# Porting UNIX* to Windows NT

David G. Korn (dgk@research.att.com)

*AT&T Laboratories*
*Murray Hill, N. J. 07974*

## Abstract

The Software Engineering Research department at Murray Hill writes and distributes several widely used development tools and reusable libraries that are portable across virtually all UNIX platforms.[1] To enhance reuse of these tools and libraries, we want to make them available on systems running Windows NT[2] and/or Windows 95[3]. We did not want to support multiple versions of these libraries, and we wanted to minimize the amount of conditionally compiled code.

This paper describes an effort of trying to build a UNIX interface layer on top of the Windows NT and Windows 95 operating system. The goal was to build an open environment rich enough to be both a good development environment and a suitable execution environment. This meant that the overhead needed to be small enough so that there was no incentive to program to the native operating system directly. The openness meant that the complete facilities of the native operating system were accessible through this environment.

The result of this effort is a set of libraries, headers, and utilities that we collectively refer to as UWIN. UWIN contains nearly all the X/Open Release 4[4] headers, interfaces and commands. We discuss alternative porting strategies, commercial products, design goals, problems that had to be overcome, and the current status. Some performance measurements of the current system are presented here.

## 1. INTRODUCTION

The marketplace has dictated the need for software applications to work on a variety of operating system platforms. Yet, maintaining separate source code versions and development environments creates additional expense and requires more programmer training.

One way to lower this cost is to use a middleware layer that hides the differences among the operating systems. The problem with this approach is that it forces you to program to a non-standard, and often proprietary, interface. In addition, it often limits you to the least common denominator of features of the different operating systems.

An alternative is to build a middleware layer based on existing standards. This has been the approach followed by IBM with the introduction of OpenEdition[5] for the MVS operating system, URL `http://www.s390.ibm.com/products/oe`. OpenEdition is X/Open compliant so that a large collection of existing software can be transported at little cost.

Windows NT is an operating system developed by Microsoft to fill the needs of the high-end market. It is a layered architecture, designed from the ground up, built around a microkernel that is similar to Mach.[6] One or more *subsystems* can reside on top of the microkernel which gives Windows NT the ability to run different logical operating systems simultaneously. For example, the OS/2 subsystem allows OS/2 applications to run on Windows NT. The most important subsystem that runs on Windows NT is the WIN32 subsystem. The WIN32 subsystem runs all applications that are written to the WIN32 Application Programming Interface (API)[7]. The API for the WIN32 subsystem is also provided with Windows 95, although not all of the functions are implemented. In most instances binaries compiled for Windows NT that use the WIN32 API will also run on Windows 95.

The POSIX subsystem allows applications that are strictly conforming to the IEEE POSIX 1003.1

* UNIX is a registered trademark, licensed exclusively through X/Open, Limited.

operating system standard[8] to run on Windows NT. Since the POSIX standard contains most of the standard UNIX system call interface, many UNIX utilities are simple to port to any POSIX system. Because most of our tools require only the POSIX interface, we thought that it would be sufficient to port them to the POSIX subsystem of Windows NT. We were wrong for the reasons described in the next section.

We investigated alternative strategies that would allow us to run programs on both UNIX and Windows NT based systems. After looking at all the alternatives, we decided to write our own library that would make porting to Windows NT and Windows 95 easy. We spent three months putting together the basic framework and getting some tools working. Realizing that the task was larger than a one person project, we contracted a small development team of 2 or 3 to do portions of the library, packaging, and documentation. This paper will discuss porting alternatives, the goals for our library, the issues that need to be addressed, and the implementation of our POSIX library. Finally, we present some performance results and future directions.

## 2. ALTERNATIVE STRATEGIES

Six basic strategies can be employed to port existing UNIX based applications to Windows NT. The first strategy is to rewrite the code using the WIN32 API. This strategy makes sense if there are no requirements to continue to run on a UNIX system. Otherwise, this strategy will either require two sets of source (which will most likely be too expensive to maintain) or the use of a WIN32 emulation library that runs on UNIX platforms. There are at least two vendors that have WIN32 API libraries for UNIX systems. We ruled out this approach because of the effort to rewrite the code to the WIN32 API and because the WIN32 API is more complex than the X/Open API.

The second strategy is to use the Microsoft C library. Microsoft supplies a library of routines that are similar to their UNIX counterparts. You could then make modifications to your application as necessary where the calls differ from the UNIX call. This strategy has been used by at least one commercial UNIX tools vendor to port GNU based tools to Windows NT. While this strategy is appropriate for some applications, other applications may require much work to overcome some subtle differences. In addition, the resulting code may have a large amount of conditionally compiled code that is hard to test and maintain.

A third strategy would be to rewrite the code using a *framework* which provides a virtual system interface. There are several vendors that offer object-oriented application layer interfaces that encapsulate the operating system and therefore enable applications to work on multiple systems. There are three drawbacks to this approach. First of all, it requires a large up front investment. Secondly, you will be locked into the vendors' libraries and not able to take advantage of savings that result from competition. Finally, you will likely be restricted to the intersection of features available on the underlying platforms.

A fourth strategy is to port the application to the POSIX subsystem of Windows NT. The POSIX subsystem can run any strictly conforming IEEE POSIX application program. This strategy should not require major investment, and any investment that you make should increase the portability of your application to other POSIX conforming systems. Unfortunately, this is not a viable alternative for most applications. Microsoft has made the POSIX subsystem as useless as possible by making it a closed system. There is no way to access functionality outside of the 1990 POSIX 1003.1 standard from within the POSIX subsystem, either at the library level or at the command level. Thus, you cannot even invoke the Microsoft C compiler from within the POSIX subsystem. However, since you can invoke POSIX commands from the WIN32 subsystem, it is possible to port some stand alone programs to the POSIX subsystem. For example, we ported the `pax` utility, the POSIX 1003.2[9] replacement for `cpio` and `tar`, to Windows NT, and it can be invoked from any WIN32 program. Softway System, Inc., URL `http://www.softway.com`, has an agreement with Microsoft to enhance the POSIX subsystem so that they can achieve POSIX 1003.2 conformance. Softway claims that they will open up the POSIX subsystem so that it can access WIN32 applications. Even if the POSIX subsystem on Windows NT is opened up, the POSIX subsystem is not available for Windows 95.

The fifth strategy is to use an existing POSIX or X/Open library that runs in the WIN32 subsystem. At the time that we began this effort, we were aware of two vendors that sell such libraries but as discussed later, these products were less than satisfactory. In addition, Steve Chamberlain at Cygnus has started writing a POSIX interface for

Windows NT and Windows 95, but it appears as if his goals are less ambitious than ours, URL `http://www.cygnus.com/misc/gnu-win32/`.

A sixth and final strategy would be to write your own POSIX library using the WIN32 API. After investigating the other alternatives, this is what we decided to do. We are convinced that this was the best strategy for us, since we believe that it resulted in a better implementation than the two commercial products described later, and because it eliminates the need to pay licensing fees for each copy of each product that uses the library. The availability of source code makes it possible to provide adequate support.

## 3. GOALS

We wanted our software to work with Windows 3.1, Windows 95, and Windows NT. A summer student wrote a POSIX library for Windows 3.1 and we were able to port a number of our tools. However, the limited capabilities of Windows 3.1 made it a less than desirable platform. We instead focused our goals on Windows NT and Windows 95. We decided to use only the WIN32 API for our library so that the library would work on Windows 95 and so that all WIN32 interfaces would be available to applications.

Initially, our goal was to provide the IEEE POSIX.1 interface with a library. This would be sufficient to run `ksh` and about eighty utilities that we had written. It soon became obvious that this wasn't enough for many applications. Most real programs use facilities that are not part of this standard such as sockets or IPC.

We needed to provide a character based terminal interface so that curses based applications such as `vi` could run. After the initial set of utilities was running, we wanted to get several socket based tools working. Several projects at AT&T that became interested in using our libraries, required the System V IPC facilities. The S graphics system[10] and `ksh-93`[11] required runtime dynamic linking. As the project progressed, the need for privileged users, such as `root` on UNIX systems, surfaced. We decided that it was important to have `setuid` and `setgid` capabilities. It soon became clear that we needed full UNIX functionality and we set our goal on X/Open Release 4 conformance.

We needed to have a complete set of UNIX development tools since we didn't want to get into the business of rewriting makefiles or changing build scripts. Most code written at AT&T, including our own, uses `nmake`[12], (no relation to the Microsoft `nmake`), but we also wanted to be able to support other `make` variants. We didn't want to do manual configuration on tools that have automatic configuration scripts.

One important goal that we had from the beginning was to not require WIN32 specific changes to the source to get it to compile and execute. The reason for this is that we wanted to be able to compile and execute UNIX programs without having to understand their semantics. In addition we wanted to limit the number of new interfaces functions and environments variables that we had to add to use our library. It is difficult to manage more than one or two environment variables when installing a new package.

Another goal that we had was to provide a robust set of utilities with minimal overhead. If utilities written to the X/Open API were noticeably slower than the same utilities written to the native WIN32 API, then they were likely to be rewritten making our library unnecessary in the long run.

A final and important goal was interoperatability with the native Windows NT system. Integration with the native system not only meant that we could use headers and libraries from the native system, but that we could pass environment variables and open file descriptors to commands written with the native system. There couldn't be two unrelated sets of user ids and separate passwords. If write permission were disabled from the UNIX system, then there should be no way to write the file using facilities in the native system and vice versa.

We have not as yet achieved all of our goals, but we think that we are close. We are in the process of running the X/Open conformance tests to verify compliance with the X/Open API's. The rest of the paper will discuss some of the issues we needed to deal with and our solutions.

## 4. PROBLEMS TO SOLVE

The following problems need to be understood and dealt with in porting applications to Windows NT. These are some of the issues that need to be addressed by POSIX library implementations. Section 6 describes how UWIN solved most of these problems.

## 4.1 Windows NT File Systems

Windows NT supports three different file systems, called FAT, HPFS, and NTFS. FAT, which stands for File Access Table, is the Windows 95 file system. It is similar to the DOS file system except that it allows long file names. There is no distinction between upper and lower case although the case is preserved. HPFS, which stands for High Performance File System, was designed for OS/2. NTFS, the native NT File System, is similar to the Berkeley file system.[13] It allows long file names (up to 255 characters) and supports both upper and lower case characters. It stores file names as 16 bit Unicode names.

The file system namespace in Win32 is hierarchical as it is in UNIX and DOS. A pathname can be separated by either a `/` or a `\`. Like DOS, and unlike UNIX, disk drives are specified as a colon terminated prefix to the path name, so that the pathname `c:\home\dgk` names the file in directory `\home\dgk` on drive `c:`. Many UNIX utilities expect only `/` separated names, and expect a leading `/` for absolute pathnames. They also expect multiple `/`'s to be treated as a single separator.

Even though NTFS supports case sensitivity for file names, the WIN32 API has no support for case sensitivity for directories and minimal support for case sensitivity for files, limited to a `FILE_FLAG_POSIX_SEMANTICS` creation flag for the `CreateFile()` function. Certain characters such as `*`, `?`, `>`, `|`, `:`, `"`, and `\`, cannot be used in filenames created or accessed with the WIN32 API. The names, `aux`, `com1`, `com2`, `nul`, and filenames consisting of these names followed by any suffix, cannot be created or accessed in any directory through the WIN32 API.

Because Windows 95 doesn't support execute permission on files, it uses the `.exe` suffix to decide whether a file is an executable. Windows NT doesn't require this suffix, but some NT utilities, such as the DOS command interpreter, require the `.exe` suffix.

## 4.2 Line Delimiters

Windows NT uses the DOS convention of a two character sequence `<cr><nl>` to signify the end of each line in a text file. UNIX uses a single `<nl>` to signify end of line. The result is that file processing is more complex than it is with UNIX. There are separate modes for opening a file as text and binary with the Microsoft C library. Binary mode treats the file as a sequence of bytes. Text mode strips off each `<cr>` in front of each new-line as the file is read, and inserts a `<cr>` in front of each `<nl>` as the file is written. Because the number of characters read doesn't indicate the physical position of the underlying file, programs that keep track of characters read and use `lseek()` are likely to not work in text mode. Fortunately, many programs that run on Windows NT do not require the `<cr>` in front of each `<nl>` in order to work. This difference turned out to be less of a problem that we had originally expected.

## 4.3 Handles vs. file descriptors

The WIN32 API uses *handles* for almost all objects such as files, pipes, sockets, processes, and events, and most handles can be *duped* within a process or across process boundaries. Handles can be inherited from parent processes. Handles are analogous to file descriptors except that they are unordered, so that a per process table is needed to maintain the ordering.

Many handles, such as pipe, process, and event handles, have a synchronize attribute, and a process can wait for a change of state on any or all of an array of handles. Unfortunately, socket handles do not have this attribute. One of the few novel features of WIN32 is the ability to create a handle for a directory with the synchronize attribute. This handle changes state when any files under that directory change. This is how multiple views of a directory can be updated correctly in the presence of change.

## 4.4 Inconsistent Interfaces

The WIN32 API handle interface is often inconsistent. Failures from functions that return handles return either 0 or -1 depending on the function. The `CloseHandle()` function does not work with directory handles. The WIN32 API is also inconsistent with respect to calls that take pathname arguments and calls that take handles. Some functions require the pathname and others require the handle. In some instances, both calls exist, but they behave a little differently.

## 4.5 Chop Sticks Only

The WIN32 subsystem does not have an equivalent for `fork()` or an equivalent for the `exec*()` family. There is a single primitive, named `CreateProcess()` that takes 10 arguments, yet still cannot perform the simple operation of overlaying the current process with a new program as `execve()` requires.

## 4.6 Parent/Child Relationships

The WIN32 subsystem does not support parent/child relationships between processes. The process that calls `CreateProcess()` can be thought of as the parent, but there is no way for a child to determine its parent. Most resources, such as files and processes, have handles that can be inherited by child processes and passed to unrelated processes. Any process can wait for another process to complete if it has an open handle to that process. There is a limited concept of process group that affects the distribution of keyboard signals, and a process can be placed in a new group at startup or can inherit the group of the parent process. There is no way to get or set the process group of an existing process.

## 4.7 Signals

The WIN32 API provides a structured mechanism for exception handling. Also, signals generated from within a process are supported by the API. However, signals generated by another process have no direct method of implementation. In addition to being able to interrupt processing at any point, a signal handler might perform a `longjmp` and never return.

## 4.8 Ids and Permissions

Windows NT uses *subject identifiers* to identify users and groups. A subject identifier consists of an array of numbers that identify the administrative authority and sub-authorities associated with a given user. A UNIX user or group id is a single number that uniquely identifies a user or group only within a single system. Information about users is kept in the a registry database which is accessible via the WIN32 API and the LAN manager API.

Windows NT uses an *access control list*, ACL, on each file or object to control the access of the file or object for each user. UNIX uses a set of permission bits associated with the three classes of users; the owner of the object, the group that the object belongs to, and everyone else. While it is possible to construct an access control list that more or less corresponds to a given UNIX permission, it is not always possible to represent a given access control list with UNIX permissions.

Windows NT has separate permissions for writing a file, deleting a file, and for changing the permission on a file. The write bit on UNIX systems determines all three. Thus, it is possible to encounter files that have partial write capability.

UNIX processes have real and effective user and group id's that control access to resources. Windows NT assigns each process a *security token* that defines the set of privileges that it has. UNIX systems use setuid/setgid to delegate privileges to processes. Windows NT uses a technique called *impersonation* to carry out commands on behalf of a given user. There is no user that has unlimited privileges as the *root* user does with UNIX. Instead the special privileges of root have been broken apart into separate privileges that can be given to one or more users. One of the biggest challenges we faced was providing the UNIX model of setuid/setgid on top of the WIN 32 interface.

The implementation of WIN32 for Windows 95 does not support the NT security model and calls return a *not implemented* error.

## 4.9 Terminal Interface

Windows NT and Windows 95 allow each character based application to be associated with a *console* which is similar to an `xterm` window. Consoles support echo and no echo mode, and line at a time or character at a time input mode, but lack many of the other features of the POSIX `termios` interface. There is no support for processing escape sequences that are sent to the console window. In echo mode, characters are echoed to the console when a read call is pending, not while they are typed. There are separate console handles for reading from the keyboard and writing to the screen.

## 4.10 Special Files

The WIN32 API supports unnamed pipes with the UNIX semantics. Named pipes are also supported but have different semantics than fifos and occupy a separate name space. There is no `/dev` directory to name special files such as `/dev/tty` and `/dev/null`. The WIN32 does support special names of the form `\\.\`*PhysicalDrive* for disk drives and tape drive devices.

Windows NT supports hard links to files, but there is no WIN32 API call to create these links. They do not support symbolic links in the file system directly, but on Windows 95 and on Windows NT 4.0, the file browser does support *short cuts* which are very similar to symbolic links.

## 4.11 Shared libraries

The WIN32 API supports the linking of shared libraries at program invocation and at run time. The libraries are called dynamically linked libraries or DLL's and are represented by two separate files.

One file provides the interface and is needed at compile time to satisfy external references. The second file contains the implementation as is needed at run time.

There are some restrictions on DLL's that are not found on UNIX system shared library implementations. One restriction is that you cannot override a function called by a DLL by providing your own version of the function. Thus, supplying your own `malloc()` and `free()` functions will not override the calls to `malloc()` and `free()` made by other DLL's. Secondly, the library can only contain pointers to data, not data itself. Thus, making a symbol such as `errno` part of a DLL is impossible. Even making symbols such as `stdin` point to data in a DLL invites trouble since it is not possible to compile code that uses

```
     static FILE *myfile = stdin;
```

### 4.12 Compilers and libraries

Microsoft sells the Visual C/C++ compiler for Windows NT and Windows 95. This compiler has both a graphical and command line interface. Microsoft also sells a software developers kit (SDK) that contains tools, including the Microsoft `nmake`. The compiler and linker use a different set of flags than standard UNIX compilers, and C files produce `.obj` files by default, rather than `.o` files. Fortunately, the linker can handle both `.obj` and `.o` files. The linker has options to choose a starting address and to specify whether the application is a console application, a GUI application, a POSIX application, or a dynamically linked library.

### 4.13 Environment Variables

The WIN32 API supports the creation and export of environment variables in much the same way that UNIX systems do. Some environment variables, such as `PATH` are used by both WIN32 and by UNIX, yet have different formats. UNIX uses a `:` separated list of pathnames; WIN32 uses a `;` separated list.

## 5. COMMERCIAL POSIX LIBRARY INTERFACES

We purchased software from the two commercial vendors that we were aware of that sell POSIX libraries for Windows NT that run under the WIN32 subsystem. Each offers a software development kit containing include files and libraries, and each offers a set of UNIX utilities. Both of these vendors require a license to use their libraries in products. We used earlier versions of their products but based on their web pages at the time this paper was written, the following description still applies. Both of these vendors supply `cc` commands that invoke the underlying Microsoft Visual C/C++ compiler. Neither of these products support symbolic links, job control and fifos. Both appear to have implemented the `exec*()` family incorrectly in that the process that does the `exec` does not terminate until the child process completes. A process that repeatedly `exec`s itself will eventually cause the operating system to run out of processes. It is not clear from their home pages whether either of these products work with Windows 95.

### 5.1 NuTCracker from DataFocus

NuTCracker, by Datafocus, URL `http://www.datafocus.com`, makes an attempt to support UNIX conventions. It maps Windows NT file names to and from UNIX file names, and adjusts the `PATH` environment variable accordingly. For example, it maps the Windows NT file name `d:\bin` to the UNIX filename `/d=/bin` and handles the special names `/dev/null` and `/dev/tty`. The `=` is a poor choice because the POSIX.2 standard for the shell language leaves the behavior of commands that have an `=` in their name unspecified.

NuTCracker ships the MKS Toolkit as the utilities. The MKS Toolkit is a completely independent implementation that does not use the NuTCracker libraries. We view this as a serious deficiency since the behavior or the utilities is no guide as to the correctness or functionality of the NuTCracker library.

The NuTCracker library lacks some functions not defined by POSIX or ANSI C that are available on UNIX systems such as `hsearch()` and `cuserid()`.

In addition to the above deficiencies, NuTCracker does not support filename case distinction.

NuTCracker supports a Motif library for porting X11 based applications including a version that offers a Windows look and feel.

### 5.2 Portage from Consensys

The other product that we purchased is named Portage and is sold by Consensys Systems, URL `http://www.consensys.com`. The source is based on System V, Release 4, which makes it the more suitable for most AT&T products. Their utilities were built from the System V source, but it was clear that changes were made in order to port them to Windows NT.

Portage Version 1.0 does not map Windows NT file name into UNIX names. They have modified some tools such as `ksh` to recognize **;** as the **PATH** delimiter in place of **:**. Version 1.0 did not support case distinction, but their home page indicates that they now do.

In terms of functionality, the NuTCracker suite is more complete than Portage.

## 6. UWIN DESIGN AND IMPLEMENTATION

We started work on writing our own POSIX library at the beginning of 1995 after being frustrated with the existing commercial products. We were able to put together a useful subset of functions in about 3 months. However, to be successful, it was necessary to provide as complete a package as possible. The library needed to handle console and serial line support, sockets, UNIX permissions, and other commonly used mechanisms such as memory mapping, IPC, and dynamic linking. In addition, to be useful, the libraries had to be documented and supported. This put the scope of the project outside of the reach of a small research department such as ours.

We subcontracted some of the development to Wipro in India to help complete this project. We jointly designed the terminal interface and the group in India implemented it. They also worked on completing the sockets library. They packaged the software for installation and are providing documentation. This section describes the UWIN implementation and how we solved many of the problems described in Section 4.

### 6.1 UWIN Architecture

The current implementation of UWIN consists of two dynamically linked libraries named `posix.dll` and `ast.dll` that more or less implement the functions documented respectively in section 2 and section 3 of UNIX manuals. In addition, a server process named UMS runs as `Administrator` (the closest thing to `root`). UMS generates security tokens for setuid/setgid programs as needed. It also is responsible for keeping the `/etc/passwd` and `/etc/group` files consistent with the registry database. The Architecture for UWIN is illustrated in Figure 1. The UMS server does not exist for Windows 95.

The `posix.dll` library maintains an open file table that is shared by all the currently active UNIX processes in a memory mapped region. This region is writable by all processes so that an ill-behaved process could affect another process. Even though all processes have read and write access to the shared segment, secure access to kernel objects in Windows NT is not compromised by this model because a process must have access rights to an object to use it; knowing its address or value doesn't give additional access rights. Some initial measurements indicated that the alternative of having a server process update the shared memory region, would have had a performance penalty that we did not believe was worth the cost. However, this is an area for future investigation.

The open file table is an array of structures of type `Pfd_t` as illustrated in Table 1.

| Pfd_t | |
|---|---|
| long | refcount |
| int | oflag |
| char | type |
| short | extra |

**TABLE 1.** File Table Structure

The `refcount` field is used to keep track of free entries in this table. The Win32 `InterlockedIncremenet()` and `InterlockedDecremenet()` functions are used to maintain this count so that concurrent access by different processes will work correctly. The `oflag` field stores the open flags for the file. The `type` field indicates what type of file, regular, pipe, socket, or special file. The function that is used read from or to write to the file depend on the value of `type`. For certain types, the `extra` field stores an index into a type-specific table that stores additional information about this file.

The `posix.dll` library also maintains a per process structure, `Pproc_t`. The per process structure contains information required by UNIX processes that is not required by Win32 processes such as parent process id, process group id, signal masks, and process state as illustrated in Table 2.

Like the open file table, the process table maintains a reference count so that process slots can be allocated without creating a critical region. The meaning of most of the fields in the process structure can be deduced by its name. The `Psig_t` structure contains the bit mask for ignored, blocked and

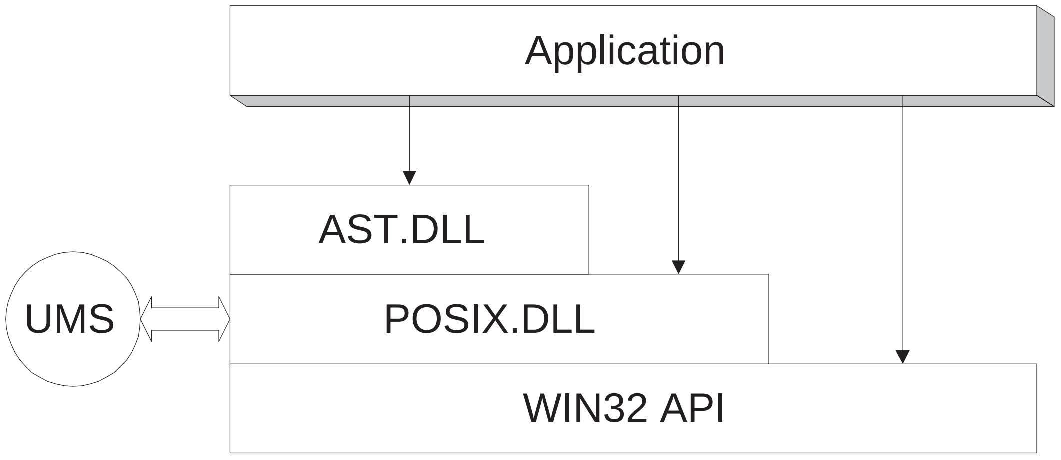

**Figure 1.** – UWIN Architecture

pending signals. When the first child process is invoked by a process, a thread is created that waits for this and subsequent processes to complete. The `waitevent` field contains an event this thread also waits on so that additional children can be added to the list of children to wait for.

| Pproc_t | |
|---|---|
| long | refcount |
| HANDLE | proc,thread |
| HANDLE | sigevent |
| HANDLE | waitevent |
| HANDLE | etok,rtok |
| ulong | ntpid |
| pid_t | pid,ppid,pgrp,sid |
| id_t | uid,gid |
| Psig_t | siginfo |
| mode_t | umask |
| ulong | alarmremain |
| int | flags |
| time_t | cutime,cstime |
| Pprocfd_t | fdtab[OPEN_MAX] |

**TABLE 2.** Process Table Structure

The process structure contains an array of up to `OPEN_MAX` structures of type `Pprocfd_t` that is indexed by file descriptor. The `Pprocfd_t` structure contains the close-on-exec bit, the index of the file in the open file table, and the corresponding handle or handles as illustrated in Table 3.

The `posix.dll` library implements the `malloc()`, `realloc()`, and `free()` interface using the `Vmalloc` library written by Kiem-Phong Vo[14]. The `Vmalloc` library provides an interface to walk over all memory segments that are allocated which is needed for the `fork()` implementation described later.

| Pproc_t | |
|---|---|
| short | index |
| char | close_exec |
| HANDLE | primary |
| HANDLE | secondary |

**TABLE 3.** Process file structure

The `ast.dll` library provides a portable application programming interface that is used by all of our utilities. The interface to this library is named `libast.a`, for compatibility with its name on UNIX systems. `libast.a` provides C library functions that are not present on all systems so that application code doesn't require `#ifdefs` to handle system dependencies. `libast.a` is built using the `iffe` command [15] to feature test the host system and determine what interfaces do not exist in the native system.

libast.a relies on the Microsoft C library for most of the ANSI-C functionality. The most significant exception to this, other than malloc() which is provided by posix.dll, is the stdio library. libast.a provides its own version of the stdio library based on calls to Sfio[16]. The Sfio library makes calls to posix.dll rather than making direct calls to the WIN32 API as the Microsoft C library does so that pathnames are correctly mapped.

The use of Sfio also provides a simple solution to the <cr><nl> problem. When a file is explicitly opened for reading as a text file, an Sfio *discipline* for read() and lseek() can be inserted on the stream to change all <cr><nl> sequences are to <nl>. The lseek() discipline uses logical offsets so that the removal of <cr> characters is transparent. We did not provide a discipline to change <nl> to <cr><nl> since we discovered that most Windows 95 and Windows NT utilities worked without the <cr>s. The <cr>s could be inserted by a filter such as sed if required.

## 6.2 Files

The posix.dll library performs the mapping between handles and file descriptors. Usually, each file descriptor has one handle associated with it. In some cases, two handles may be associated with a file descriptor. An example of this is a console that is open for reading and writing which uses separate handles for reading and writing.

The posix.dll library handles the mapping between UNIX pathnames and WIN32 pathnames. Many UNIX programs assume that pathnames that do not begin with a / are relative pathnames. In addition, only / is recognized as a delimiter. There is only a single root directory; the operation of changing to another drive does not change the root directory. The posix.dll library maps all file names it encounters. If the file name begins with a / and the first component is a single letter, then this letter is taken as the drive letter. Thus, the UNIX filename /d/bin/date gets translated to d:\bin\date. The file name mapping routine also recognizes special file names such as /dev/tty and /dev/null. A / not followed by a drive letter is mapped to the drive that UWIN has been installed on so that programs that embed absolute pathnames for files in /bin, /tmp, /dev, and /etc work without modification.

Finally, the path search algorithm was modified to look for .exe and .bat suffices.

One problem introduced by the pathname mapping is that passing file name arguments to native NT utilities is more difficult since it understands DOS style names, not UNIX names. A library routine was added to return a DOS name given a UNIX name.

The posix.dll library pathname mapping function also takes care of exact case matching on file systems that require it. One of the most troublesome aspects of the WIN32 API is its lack of support for pathname case distinction. It is not uncommon to have files named Makefile and makefile in the same directory in UNIX. UWIN handles case distinction by calling the WIN32 CreateFile() function both with and without the FILE_FLAG_POSIX_SEMANTICS function. If they compare equal, it executes the function internally, otherwise it spawns a POSIX subsystem process to carry out the task.

## 6.3 fork/exec

The fork() system call was implemented by creating a new process with the same startup information as the current process. Before executing main(), it copies the data and stack of the parent process into itself. Handles that were closed when the new process was created are duplicated into the new process. The exec*() family of functions was much harder to implement. The problem is that there is no way to overlay the calling process. Portage and NuTCracker have the current process wait for the child process to complete and then exit. There are two problems with this approach. First, a process that *exec*s repeatedly will fill up the process table. More importantly, resources from the parent process are not released. Our method causes the child process to be reparented to the grandparent and the process that calls exec*() to exit. The process id returned by the getpid() function will be the process id of the process that invoked the exec*() function. In other cases, it will be the same process id as the WIN32 uses. To prevent that process id from being used again by WIN32, a handle to the process is kept by the grandparent process.

Even though we implemented fork() and the exec*() family of functions, our code rarely uses them. Because the CreateProcess() function doesn't have an overlay flag, two processes need to be created in order to do both fork() and exec*(). libast provides a spawn*() family of functions that combines the functionality of fork()/exec*() on systems that don't have the spawn*() family. All functions in libast that create processes such as system() and popen()

are programmed with this interface. On most UNIX systems, the `spawn*()` family is written using `fork()` or `vfork()` and `exec*()`. We implemented `spawn*()` in our `posix.dll` library to call `CreateProcess()` directly.

## 6.4 Signals

Signals are handled by having each process run a thread that waits on an event. To send a signal to a process, the bit corresponding to the given signal number is set in the receiving process's process block, and then its signal thread event is set. The signal thread then wakes up and looks for signals. It is important for the signal handler to be executed in the primary thread of the process, since the handler may contain a `longjmp()` out of the handler function. Prior to calling `main()`, an exception filter is added to the primary thread that checks for signals. The signal thread does this by suspending the primary thread raising an exception that will active the exception filter of the primary thread, and then resuming the primary thread.

## 6.5 Terminals

The POSIX `termios` interface is implemented by creating two threads; one for processing keyboard input events, and the other for processing output events and escape sequences. These threads are connected to the read and write file descriptors of the process by pipes. The same architecture is used for socket based terminals and serial I/O lines. Initially, these threads run in the process that created the console and make it the controlling terminal. These threads service all processes that share the controlling terminal. New threads will be created if the process that owns the threads terminates and another process is sharing the console. When a process is created, these threads are suspended and the console handles are passed down to the child. This enables a native application to run with its standard input and output as console handles. If the application has been linked with the `posix.dll`, then these threads are resumed before `main()` is called so that UNIX style terminal processing takes place. The result is that UNIX processes will echo characters as they are typed and respond to special keys specified by `stty`, whereas native WIN32 applications will only echo characters when they are read and will use Control-C as the interrupt character.

## 6.6 Ids and Permissions

Permissions for files are only available on Windows NT. Calls to get an set permissions return *not implemented* errors on Windows 95. Creating a Windows NT ACL that closely corresponds to UNIX permissions isn't very difficult. The ACL needs three entries; one for owner, one for group, and one that represents the group that contains all users. Windows NT allows separate permission to delete a file and to change its security attribute. These permissions are give to the owner of a file. The UNIX `umask()` command sets the default ACL so that native applications that are run by UWIN will create files with UNIX type permissions.

Mapping of subject identifiers to and from user and group ids is more complex. UWIN maintains a table of subject identifier prefixes, and constructs the user id and group id by a combination of the index in this table and the last component of the subject identifier. The number of subject identifier prefixes that are likely to be encountered on a given machine is much smaller than the number of accounts so that this table is easier to maintain.

## 6.7 Special files and Links

Special files such as fifos and symbolic links require `stat()` information that is not kept by the NT or FAT file systems. Also, the file system does not store the `setuid` and `setgid` permission bits. With the NT file system, this extra information has been stored by using a poorly documented feature called *multiple data streams* that allows a file to have multiple individually named parts. A separate data stream is created to hold additional information about the file. The `SYSTEM` attribute is put on any file or directory that has an additional data stream so that they can be identified quickly with minimal overhead during pathname mapping.

Using multiple data streams requires the NT file system. On other file systems, fifos and symbolic links are implemented by storing the information in the file itself. The setuid, setgid functionality is not supported on these file systems.

UWIN treats Windows 95 and Windows NT 4.0 *short cuts* as if they were symbolic links. However, these links can be created with any of the UWIN interfaces. This was done by reverse engineering the format of a short cut file and finding where the pathname of the file that it referred to was stored.

Fifos are implemented by using WIN32 named pipes. A name is selected based on the creation date of the fifo file. Only the first reader and the first writer on the fifo create and connect to the named pipe. All other instances duplicate the handle of either the reader or the writer. This way all writers to a fifo use the same handle as required by fifo semantics.

A POSIX subsystem command is also invoked to create hard links since there is no WIN32 API function to do this. Hard links fail for files in the FAT file system.

## 6.8 Sockets

Sockets are implemented as a layer on top of `WINSOCK`, the Microsoft API for BSD sockets. Most functions were straight forward to implement. The `select()` function proved more difficult than we had anticipated because socket handles could not be used for synchronization, and because the Microsoft `select()` call only worked with socket handles. The `posix.dll select()` function allows different types of file descriptors to be waited for.

Our first implementation of `select()` created a separate thread that used the Microsoft `select()` to wait for socket handles, and created an event for the main thread to add to the list of handles to wait for. Our second implementation used a library routine to convert input/output events on sockets to windows messages and then waited for both windows messages and handle events simultaneously. This method had the added advantages that it was possible to implement `SIGIO` and that it was easy to add a pseudo file device named `/dev/windows` that could be used to listen for windows messages. Adding this pseudo device made it possible to use the UNIX implementation of `tcl` to port `tksh`[17] applications to Windows NT.

## 6.9 Invocation

When UWIN invokes a process, it does not know whether the process is a UWIN process or a native process. It modifies the `PATH` variable so that it uses the **;** separated DOS format. It also passes open files in the same manner that the Microsoft C library does so that programs that are compiled with this library should correctly inherit open files from UWIN programs. The initialization function also sees whether a security token has been placed in its address space by the UMS server, and if so, it impersonates this token.

The POSIX library has an initialization routine that sets up file descriptors and assigns the controlling terminal starting the terminal emulation threads as required. The `posix.lib` library also supplies a `WinMain()` function that is called when the program begins. This function initializes the `stdin`, `stdout`, and `stderr` functions and then calls a `posix.dll` function passing the address of another `posix.lib` function that actually invokes `main()`. The `posix.dll` function starts up up the signal thread and sets the exception filter for signal processing as described above. The reason for this complexity is so that UNIX programs will start with the correct environment, and so that `argv[0]` will have UNIX syntax without the trailing `.exe` since many programs use `argv[0]`. Much of the complexity occurs inside the `posix.dll` part because programs do no require recompilation when changes are added there.

## 7. CURRENT STATUS

At the time of this writing, most interfaces required by the X/Open Release 4 standard have been written and work as described in the standard. The X/Open standard requires full ANSI C functionality as well. In addition, interfaces for the curses library, the sockets library, the dynamic linking library, are also working.

A C/C++ compiler wrapper has been written that calls either the Microsoft Visual C/C++ 2.x or 4.x compiler. This compiler supports the most commonly used UNIX conventions and implicitly sets default include files and libraries. In addition it has an added hook for specifying native compiler and linker options. Applications compiled with our `cc` command can be debugged with native debuggers such as the Visual C/C++ debugger. Several auto configuration programs use the output of the C preprocessor to probe the features of the system. The output format of the Microsoft C compiler caused some of the configuration programs to fail. To overcome this, a filter is inserted when running the compiler to generate preprocessor output so that existing configuration programs work. Our compiler wrapper can be invoked as `cc` for ANSI-C compilation, as CC for C++ compilation, and as `pcc` to build POSIX subsystem applications.

Our compiler wrapper follows the normal UNIX defaults for suffixes rather than using the Microsoft conventions; `.o`'s rather than `.obj`'s. The `.exe` suffix is not required for Windows NT since it uses permission bits to distinguish executables. However, since we also want binaries to run on Windows 95, the `.exe` suffix is added to the name of the output file if no suffix is supplied when the compiler is invoked as `cc` or CC.

The lastest version of `ksh`, `ksh-93` was ported. The implementation supports all features of `ksh-93` including job control and dynamic linking of built-in commands at run time. While no changes to the code should have been necessary, changes were made to `ksh` specifically for NT. The hostname mapping attribute, `typeset -H`, which has no effect on UNIX systems, was modified to call the `posix.dll` function that returns the WIN32 pathname corresponding to a given UNIX pathname. The ability to do case insensitive matching for file expansion was also added. A compile time option to allow <cr><nl> in place of <nl> was added to the shell grammar to avoid the overhead of text file processing.

About 150 UNIX tools have been ported to Windows NT, the vast majority required no changes. Common software development tools such as `yacc`, `lex`, `make` and `nmake` have also been ported. Most of the utilities are versions that we have written at AT&T over the last ten years and are easily portable to all UNIX platforms. Other utilities, such as `make`, `bc`, and `gzip` we compiled from the GNU source using `autoconfig` to generate headers and makefiles. The `yacc` and `less` utilities and the new `vi` program were ported from freely available BSD source code. In most cases, no changes were made to the original source code.

The X Windows code has two parts, the client and the server. The server had already been ported to Windows NT and Windows 95 by commercial vendors and there was no need to build UWIN version for it. In addition, the server is often running on a UNIX host. The most difficult part of porting the X Windows client code was the fact that it had `#ifdefs` for `WIN32` that selected native WIN32 calls, bypassing the UWIN calls. Once this was straightened out, the compilation was straightforward.

## 8. PERFORMANCE

There are two issues to consider with respect to performance. The first is how UWIN performs compared to using the WIN32 API and/or Microsoft C library directly. The comparison of UWIN to native performance measures one of the costs involved in using UWIN as opposed to using an alternative strategy such as rewriting to the WIN32 API.

The second is how Windows NT performs relative to other UNIX systems. The performance of UNIX operating systems on the Pentium processor was investigated by Keven Lai and Mary Baker[18], and showed that except for networking, the Linux system, URL `http://www.linux.org`, performs the best of the UNIX systems. The comparison to a UNIX system may be important in deciding whether to choose a UNIX platform or a Windows NT platform, although other considerations often dictate this choice.

All performance comparisons were made on an Micron computer with 133MZ Pentium processor and 32M-bytes of memory. The WIN32 measurements were made using the NTFS file system on Windows NT 4.0 operating system. The UNIX measurements were make on Linux version 2.0.18 on the same hardware.

There are four sets of tests. The first set of tests, shown in Table 4, are the same ones used in the 1991 Usenix `Sfio` paper. The implementation of stdio under UWIN uses `Sfio` rather than the Microsoft implementation since the Microsoft implementation makes WIN32 calls directly rather than going through the `read()/write()` UNIX interface. The Linux tests were run with an `Sfio` implementation rather than using the native implementation to make it easier to compare results. The results show that applications that are dominated by calls to `Stdio` or `Sfio` are likely to perform at least as well when run under UWIN.

The second set of tests, summarized in Table 5, measures the performance of certain systems calls; for example the time to open and close files, to read and write data, the time to create and delete files, and the time to open and read directories. The tests are as follows:

1. Open and close a file 10000 times.
2. Create and delete a file 10000 times.
3. Open and read a directory containing two files 10000 times.
4. Open and read a directory containing 500 files 10000 times.
5. Run `system("/bin/echo")` 100 times.

The tests were run five times and the middle three times were averaged. The first four tests report the sum of user+system time. The last test uses only elapsed time because of the difficulty of obtaining accumulated times for processes using `CreateProcess()` call.

These tests show that creating and deleting files in UWIN is much slower than with Linux. Much of the time difference is due to the way UWIN deletes files to provide UNIX semantics. With UNIX it is possible to delete a file while it is open, and then create a file of the same name. Clearly a more efficient mechanism for doing this is needed.

While reading small directories is slower than with Linux, the tests show that large directories are actually faster with NT. The `system()` test shows that Linux is quite a bit faster in launching processes than NT. The NT native test, unlike the UWIN test, executes `/bin/echo` directly without running the shell so that the results are better than they might otherwise be.

The third set of benchmarks is called the Modified Andrew Benchmarks[19]. These benchmarks measure the elapsed time to perform a set of tasks such as copying files, doing recursive walks, and compiling code. The original set of Andrew Benchmarks used the native C compiler; the Modified Andrew Benchmarks come with source code for a stripped down version of `gcc` so that the differences in compilers can be eliminated. It look little effort to modify the makefiles and build the compiler. The time to run the Modified Andrew Benchmark was 110 seconds under UWIN. The time was a mere 18 seconds under Linux. This benchmark shows the effect of the slower file and process creation times. The UWIN times could be improved by using the `spawn` family of functions in place of `fork/exec` to execute the components of the compiler as the UWIN `cc` command does. In both cases the test failed to complete near the final step; creating the archive because the native archiver was unable to handle the format produced by the generated `gcc` compiler.

The final set of benchmarks that we tried to run was the benchmark suite named `lmbench`, written by Larry McVoy and presented at the 1996 USENIX conference[20]. We were able to run only a portion of these benchmarks because many of the benchmarks require the `rpc` library which hasn't been ported to UWIN yet. In addition, we omitted tests that were covered by earlier benchmarks. The results of this benchmark are presented in Table 6. While it confirms that the I/O bandwidth under UWIN is quite good, it also shows that other aspects such as pipe latency is quite large. We did not investigate this discrepancy.

## 9. FUTURE

We are in the process of running X/OPEN conformance tests on UWIN and see how close we have come to being compliant. In addition, we are trying to decide how to port the *n* dimensional file system, *n*-`DFS`[21], to Windows NT. *n*-`DFS` provides a mechanism to add file system services such as viewpathing and versioning. The difficulty in porting *n*-`DFS` is that it must also capture native WIN32 API calls to provide a transparent interface.

The current version of UWIN does not handle may of the internationalization issues well. The current implementation has been compiled for ASCII rather than UNICODE. We plan to use UFT8 encoding of UNICODE for the system call interface, and to convert to UNICODE on the NT file system. This way we do not need to build separate binaries for UNICODE.

The current version of UWIN does not support files larger than two gigabytes because the size of `off_t` is stored as a 32 bit integer. The underlying NTFS file system supports 64 bit file offsets. Since the next version of `Sfio` supports 64 bit file offsets, we plan to support large files in a future version of UWIN.

Another issue worth investigating is whether it is possible to run Linux binaries under UWIN. This would only make sense for dynamically linked programs.

Finally there are some WIN32 interfaces that could be handled through the file system interface such as the Windows NT registry and the clipboard.

## 10. CONCLUSIONS

There appear to be few if any technical reasons to move from UNIX to Windows NT. The performance of Linux exceeds that of NT 4.0 and Linux appears to be more reliable. On three occasions NT 4.0 crashed when running the performance tests. There were no crashes with Linux. However, if you want to or need to move an application to Windows 95 or Windows NT, we believe the POSIX library we developed to be superior to any of the existing commercial libraries. While in many cases the performance loss using UWIN is minimal, the performance tests show that UWIN needs improvement.

The code for the `posix.dll` library is fairly small, about 10K lines including the terminal emulator. This library runs in the WIN32 subsystem using the WIN32 API and runs under Windows 95 as well.

We hope to be able to make version 1.1 of UWIN available in binary form on the internet. Check the internet web site `http://www.research.att.com/sw/tools` for details. We hope that this will encourage contributions of applications that have been built with UWIN.

| | | UWIN | | WIN32 | | LINUX | |
|---|---|---|---|---|---|---|---|
| test | size | seconds | Kb/s | seconds | Kb/s | seconds | Kb/s |
| fwrite | 10000K | 0.63 | 15923 | 0.49 | 20408 | 0.85 | 11709 |
| fread | 10000K | 0.28 | 36231 | 0.27 | 36764 | 1.03 | 9671 |
| revrd | 10000K | 0.31 | 32679 | 0.33 | 30303 | 0.50 | 19960 |
| fw757 | 10000K | 0.76 | 13227 | 0.94 | 10593 | 1.06 | 9416 |
| fr757 | 10000K | 0.41 | 24154 | 0.36 | 27472 | 1.09 | 9199 |
| rev757 | 10000K | 0.91 | 10989 | 1.36 | 7731 | 0.66 | 15128 |
| copy&rw | 10000K | 1.01 | 9940 | 1.00 | 9999 | 1.39 | 7173 |
| seek+rw | 2000S | 0.93 | 34566 | 0.83 | 38672 | 1.78 | 8974 |
| putc | 5000K | 0.87 | 5720 | 1.00 | 5020 | 2.09 | 2393 |
| getc | 5000K | 0.49 | 10245 | 0.50 | 9960 | 1.57 | 3194 |
| fputs | 50000L | 0.51 | 97656 | 0.59 | 84459 | 0.88 | 57102 |
| fgets | 50000L | 0.40 | 126262 | 0.65 | 77399 | 0.54 | 93283 |
| revgets | 50000L | 0.69 | 72254 | 2.37 | 21132 | 0.75 | 67114 |
| fprintf | 50000L | 5.84 | 8567 | 7.31 | 6838 | 5.58 | 8968 |
| fscanf | 50000L | 4.59 | 10888 | 4.92 | 10154 | 5.60 | 8933 |

**TABLE 4.** Stdio timings

| | | UWIN | | WIN32 | | LINUX | |
|---|---|---|---|---|---|---|---|
| test | count | seconds | #/s | seconds | #/s | seconds | #/s |
| open/close | 10000 | 2.88 | 3472 | 1.89 | 5291 | 0.45 | 22222 |
| create/delete | 10000 | 42.90 | 233 | 12.77 | 783 | 3.11 | 3215 |
| readdir-2 | 1000 | 9.62 | 1040 | 4.07 | 2457 | 2.80 | 3571 |
| readdir-500 | 1000 | 42.41 | 236 | 34.34 | 291 | 45.17 | 221 |
| system | 100 | 13.62 | 7 | 5.89 | 17 | 2.84 | 35 |

**TABLE 5.** Syscall timings

| Test | Unit | UWIN | Linux |
|---|---|---|---|
| Null syscall | us | 4 | 3 |
| Pipe latency | ms | 295 | 35 |
| Pipe bandwidth | MB/sec | 23.33 | 39.37 |
| File write bandwidth | KB/sec | 1995 | 2824 |
| File read bandwidth | MB/sec | 33.33 | 44.18 |
| Mmap read bandwidth | MB/sec | 75.00 | 86.33 |
| Memory read bandwidth | MB/sec | 90.91 | 82.28 |
| Memory write bandwidth | MB/sec | 76.92 | 83.26 |

**TABLE 6.** Selected *lmbench* results

## *REFERENCES*

1. *Practical Reusable UNIX Software,* Edited by Balanchander Krishnamurthy, John Wiley & Sons, 1995.

2. *Microsoft Win32 Programmer's Reference, Volume 2* Microsoft Press, 1993.

3. Matt Pietrek, *Windows 95 System Programming Secrets*, IDG Books, 1995.

4. *The X/Open Release 4 CAE Specification, System Interfaces and Headers*, Issue 4, Vol. 2, X/Open Co., Ltd., 1994.

5. *The OpenEdition MVS Users Guide,IBM,*1996.

6. M. Accetta et al., *Mach: A New Kernel Foundation for Unix Development,* Usenix Association Proceedings, Summer 1986.

7. Jeffrey Richter, *Advanced Windows – The Developers Guide to the Win32 API for Windows NT 3.5 and Windows 95*, Microsoft Press, 1995.

8. *POSIX – Part 1: System Application Program Interface,* IEEE Std 1003.1-1990, ISO/IEC 9945-1,1990.

9. *POSIX – Part 2: Shell and Utilities,* IEEE Std 1003.2-1992, ISO/IEC 9945-2, IEEE, 1993.

10. Richard A. Becker, John M. Chambers, and Alan R. Wilks, *The New S Language : A Programming Environment for Data Analysis and Graphics*, Wadsworth & Brooks/Cole, New Jersey, 1988.

11. Morris Bolsky and David Korn, *The New KornShell Command and Programming Language*, Prentice Hall, 1995.

12. Glenn S. Fowler, *A Case for Make,* Software – Practice and Experience, Vol. 20, No. S1, pp. 30-46, 1990.

13. M. McKusik, W. Joy, S. Leffler, and R. Farbry, *A Fast File System for UNIX*, ACM Transactions on Computer Systems, Vol. 2, No. 3, August, 1984, 181-197.

14. Kiem-Phong Vo, *Vmalloc - A General and Efficient Memory Allocator*, Software – Practice and Experience, Vol. 26, No. 3, pp 357-374, March 1996.

15. Glenn S. Fowler, David G. Korn, John J. Snyder, and Kiem-Phong Vo, *Feature Based Portability*, Proceedings of the USENIX Symposium on Very High Level Languages, 1994.

16. David Korn and Kiem-Phong Vo, *SFIO - A Safe/Fast String/File I/O,* Proceedings of the Summer Usenix, pp. 235-255, 1991.

17. Jeffrey Korn, `Tksh`*: A Tcl Library for KornShell*, Fourth Annual Tcl/Tk Workshop, Monterey, CA, July 1996, pp 149-159.

18. Kevin Lai and Marry Baker, *A Performance Comparison of UNIX Operating Systems on the Pentium*, Proceedings of the San Diego Usenix, pp. 265-278, 1996.

19. John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West, *Scale and Performance in a Distributed File System*, ACM Transactions on Computer Systems, Vol. 6, No. 1, Feb 1988 pp 51-81.

20. Larry McVoy and Carl Staelin, *lmbench: Portable Tools for Performance Analysis*, Proceedings of the San Diego Usenix, pp. 279-294, 1996.

21. Glenn Fowler, David Korn and Herman Rao, "*n*-DFS The Multiple Dimensional File System", Trends in Software – Configuration Management, pp. 135-154, 1994.