Wux: Unix Tools under Windows[*][†]

Diomidis Spinellis
Department of Computing
Imperial College, London

Abstract

Wux is a port of Unix tools to the Microsoft Windows environment. It is based on a library providing a Unix-compatible set of system calls on top of Windows. Unix-derived tools run in parallel, communicating using the Unix pipe abstraction. All processes are run within an application template that gives them basic Windows compatibility such as input and output windows and an icon. The performance of the system is comparable to that of Unix ports to the PC architecture.

1 Motivation

The Unix operating system offers a wide variety of tools that can be used as building blocks or autonomous units for application development. Although these and similar tools can be ported to other environments [KP76, Gor93], their usability is often impaired by the lack of the glue elements available under Unix: multitasking and interprocess communication using pipes. Specifically, tool ports to the widely used MS-DOS system have to rely on serialised process execution and pipes implemented using inefficient intermediate files. The advent of the Microsoft Windows[1] system provided an accessible and widely used platform on which a more serious, complete, and efficient porting effort could be based.

The main objectives of our porting project were the following:

- Port a number of useful Unix-derived tools to the widely used Windows platform. As Windows applications have to be structured around a message loop, this exercise is far from trivial.
- Utilise the features provided by the Windows system to provide an environment that would be closer to Unix. Windows runs (in its 3.1 incarnation) on top of MS-DOS offering a number of advantages. The features important for our work were:
  - non-preemptive (co-operative) multitasking,
  - dynamically linked libraries with a data segment shared between processes,
  - memory management including virtual memory, and
  - a graphical user interface (GUI).

  We hoped that the first three capabilities could be used to provide Unix-like functionality, while the last one would enhance the usability of the resulting system.
- Experiment with the integration of these utilities in the Windows environment. We wanted to test the possible synergy between the traditional Unix tools and Windows concepts like the mandatory message loop structure, the clipboard, and OLE (object linking and embedding).

[*] In *Proceedings of the Winter 1994 Usenix Conference*, pages 325–336, San Francisco, USA, January 1994. Usenix Association.

[†] This is a machine-readable rendering of a working paper draft that led to a publication. The publication should always be cited in preference to this draft using the reference in the previous footnote. This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.

[1] From here on "Windows" is used to refer to Microsoft Windows.

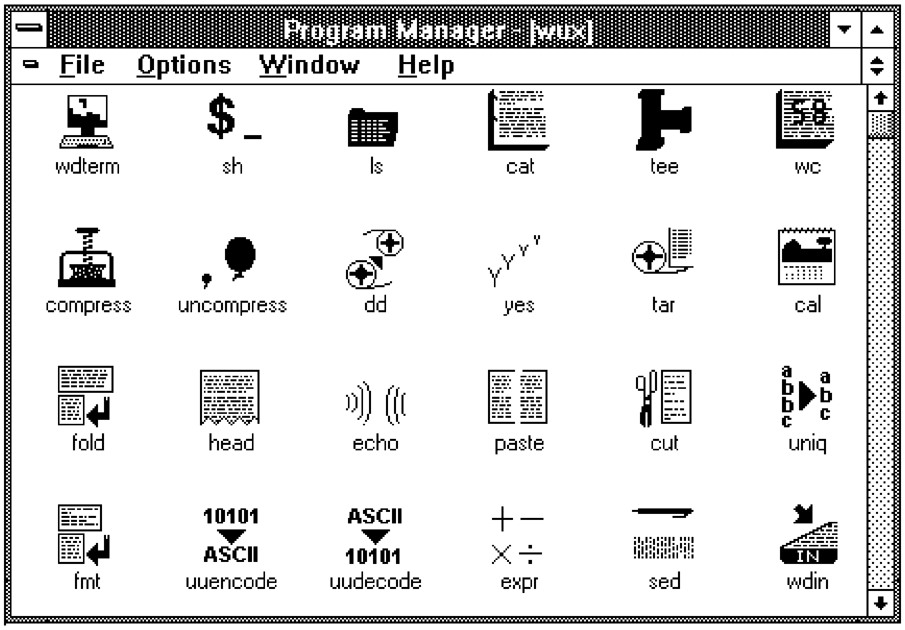


Figure 1: Program manager group of *wux* applications.

2 Functional Description

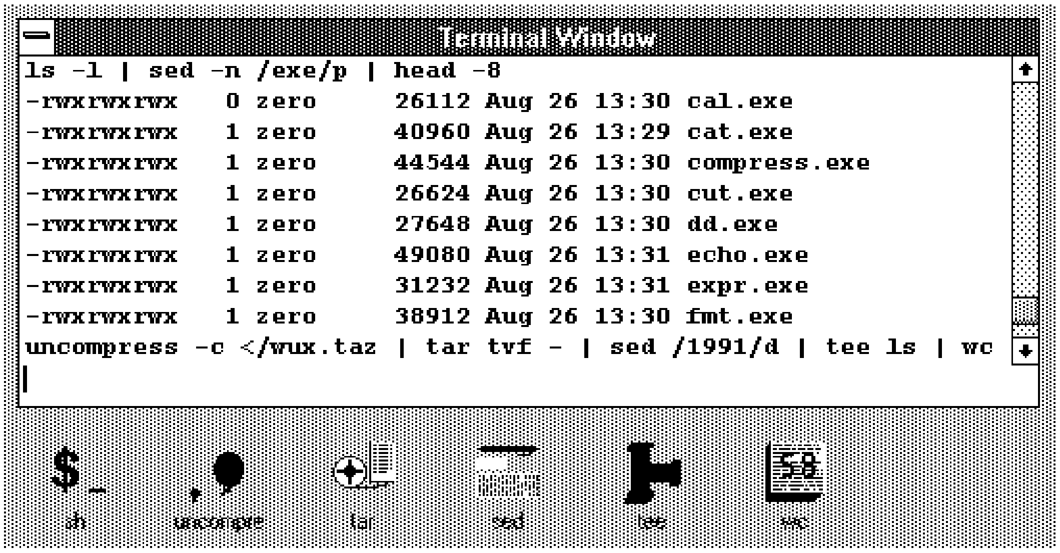
Wux consists of a set of native Windows executable files, each one of them corresponding to a Unix tool (Figure 1). The programs can be used in two different ways:

1. they can be executed from a shell-like window using the traditional Unix argument parsing and redirection functionality including pipes, or
2. they can be executed as stand-alone processes within the Windows environment using the Unix argument parsing and redirection conventions including redirection to file descriptors. To this second mode of execution we plan to add a graphical interface for argument specification and input/output file specification.

The programs run as native Windows processes, and can therefore utilise all the facilities provided by Windows, such as the ability to allocate extended and virtual memory, inter-process communication, and non-preemptive multi-tasking. Running processes are represented by their icon displayed at the bottom of the Windows screen (Figure 2). Whenever a process requests input from the standard input device or produces output to the standard output device and no redirection has been specified a suitable data source or sink window is created to provide that service. Data can be typed-in on a line-by-line basis in the *wdin* window which also provides an editable history of previous lines. Similarly, generated data can be displayed in the *wdout* window which provides a scrollbar to examine previous output. The standard error output is also redirected in a *wdout* window. Applications that are of interactive nature (such as editors and shells) can be executed with both input and output connected to *wdterm*, a terminal emulation window, akin to the X-Windows [SG86] *xterm* application.

3 *Wux* Implementation

In the following sections we describe the *wux* implementation. We first provide a short technical description of the Windows environment, then analyse the difficulties of porting Unix applications to Windows, proceed with an overview of the *wux* architecture and its components, and end with a section detailing how Unix applications are ported to *wux*, and some performance figures.



```
ls -l | sed -n /exe/p | head -8
-rwxrwxrwx   0 zero     26112 Aug 26 13:30 cal.exe
-rwxrwxrwx   1 zero     40960 Aug 26 13:29 cat.exe
-rwxrwxrwx   1 zero     44544 Aug 26 13:30 compress.exe
-rwxrwxrwx   1 zero     26624 Aug 26 13:30 cut.exe
-rwxrwxrwx   1 zero     27648 Aug 26 13:30 dd.exe
-rwxrwxrwx   1 zero     49080 Aug 26 13:31 echo.exe
-rwxrwxrwx   1 zero     31232 Aug 26 13:31 expr.exe
-rwxrwxrwx   1 zero     38912 Aug 26 13:30 fmt.exe
uncompress -c </wux.taz | tar tvf - | sed /1991/d | tee ls | wc
```

Figure 2: Windows screen running wux.

3.1 Technical Description of the Windows Environment

Windows 3.1 runs on top of the Microsoft MS-DOS operating system. In addition to a windowing graphical user interface, it provides many functions (see section 1) that are normally provided by an operating system. In contrast to the MS-DOS operating system it runs in protected CPU mode, and can thus isolate processes from disruptive interference and provide paged virtual memory.

All applications running under Windows must be structured around an event message loop. That loop typically retrieves messages from the per-process event queue and dispatches them to registered callback procedures that are part of each application. Messages can either be generated by the user (e.g. menu commands, mouse movement) or by the system (e.g. a window redraw request). During calls to the message retrieval functions Windows can pass control to other applications; this is how co-operative multitasking is implemented[2].

C programs can be compiled using the MS-DOS Intel x86 family compilers, but must be linked with a Windows-compatible library which provides a different startup code which calls *WinMain* instead of *main*. In addition, after the linking phase, a "resource compiler" attaches to the executable image of the program entities such as icons, string tables, menus, and dialog boxes. These can therefore be modified without re-compilation or relinking. Windows provides a facility of dynamic link libraries (DLLs). DLLs are shared libraries, that are loaded and linked whenever a function that resides in them is first called. Their code is executed in the context of the calling application, but their data segment is shared between all processes.

3.2 Difficulties in Porting Unix Applications to Windows

Many are the difficulties in porting Unix applications to the Windows environment. We can divide them into:

- the difficulties which arise from the requirements imposed upon conforming GUI Windows applications (startup sequence, need for an event loop),
- the differences between the Unix operating system and the "Windows over MS-DOS environment", and
- machine portability problems.

[2]Windows runs a number of virtual machines using a preemptive scheduler. These are however used to implement the MS-DOS compatibility boxes; only one of them (the system VM) runs the Windows GUI.

In the following paragraphs we provide an overview each of the three problem areas. In our discussion we assume that the Windows development is done using one of the standard software development kits and C compilers.

3.2.1 Windows-Specific Requirements

Programs that run as native Windows applications must provide a graphical user interface. At the very minimum, this should consist of an icon representing the program and a system menu that allows for the program's termination. For these to be provided, the program on startup must register a class and an event handling procedure. It must then continually retrieve and dispatch messages from the message queue in order to provide a functioning user interface and assure that other applications will be scheduled. On termination all resources allocated must be freed. It is very difficult to adapt existing non-GUI programs to this centralised control structure.

In addition, interactive character-based applications can not be directly executed, as Windows only provides a graphical front-end to native applications. The MS-DOS window application that provides a character screen, restricts programs to MS-DOS only functionality, and to a coarse grained and inefficient scheduler.

3.2.2 Operating System Differences

We will examine differences between the Unix and the "Windows under MS-DOS" environments from the perspective of the requirements for Unix compatibility. For this discussion we have used as a rough guide the POSIX.1 standard [ISO90]. Many of the programs that run on the Unix environment do not conform to this standard, and therefore our examination contains many additional pragmatic and practical problems, that would not exist in an ideal world. A thorough treatment of C program portability can be found in [Hor90]. The problematic areas can be divided among those related to the environment in which the applications are compiled, the environment in which processes execute, and the application programming interface.

Almost all Unix applications assume that they are to be compiled under the Unix environment; Windows is the platform of choice for compiling Windows applications. We will therefore combine our examination of the portability problems that arise from the differences in the compilation and execution environments. The problems related to the environment in which the processes execute are the following:

Missing or different executable files A number of Unix programs invoke other programs. Many programs rely on the Bourne shell [Bou86] to execute other programs with I/O redirection, or to perform wild-card expansion, others use one of the paging programs such as *more* or *pg* to pipe their output. Other examples are *rcs* which can execute *mail*, *crypt* which gratuitously executes *makekey*, and enhanced versions of *tar* which pipe through *dd*, *compress*, and *rsh*.

During the compilation phase applications sometimes rely on other tools such as *yacc*, *lex* or simpler text processing tools. Even when these tools exist under Windows they may be incompatible in the options they take, or the output they generate, or have lower processing size limits. Unix programs sometimes make assumptions about the C compiler used and contain *#pragma* commands, use non-portable pre-processor tricks, or expect certain variables to be placed in specific registers.

File names Many programs have source files, create temporary files, or need data files whose names are illegal for the MS-DOS file system (which is the one used by Windows 3.1). The problems can be filename size (only 11 characters are allowed), the use of one of the illegal characters ", = + < > ; ? : ", the differences in case sensitivity, or the use of MS-DOS device names.

File system semantics A common convention for Unix programs is to delete temporary files after opening them, so that they will be automatically removed from the file system when the file is closed. This trick will destroy the integrity of the MS-DOS file system.

Missing or different files, directories and devices A number of files and directories that are standard on Unix systems (such as */etc/passwd*, */tmp*, and */dev/null*) either do not exist or are different under MS-DOS. Unix programs that try to read executable, object, or library files will not run, as these formats are different under Windows.

Format of text files Lines in text files of Unix systems are delimited by a single line-feed (LF) character. The equivalent MS-DOS convention is a carriage-return (CR) followed by an LF. Some C libraries map CR/LF pairs to LF, but require the specification of the file type (binary or text) when the file is opened.

Environment variables Common Unix environment variables (*SHELL*, *HOME*) are not guaranteed to be defined under MS-DOS.

Command line size limits The program command line size limit under MS-DOS is only 127 characters. This can severely limit many applications, including the possibility of filename expansion and backquote substitution by the shell.

The third area of operating system differences encompasses the interface to the operating system. Since the code libraries are often part of the operating system in the following list we will also include library differences. The areas where the application programming interface using a Windows software development kit differs significantly from the environment found under Unix are the following:

Process creation, management, and termination There are no *fork* and *wait* system calls; the *exit* function can not be used.

Process environment There are no notions of user and group ids; the relevant system calls are therefore missing.

Exception conditions and handling Signal handling is different from the POSIX specification. Berkeley enhancements are also missing. Some signals commonly defined in Unix systems do not appear in *signal.h*.

File and Directory operations Some file status bits are not defined. *Pipe*, *select*, *link*, *symlink*, and *ftruncate/chsize* are missing.

File protection mechanisms The *umask* is not defined.

Record and file locking mechanism File locking uses a different set of operations.

Device specific functions There are no terminal controls, and I/O processing modes under Windows. The *fcntl* interface is completely different, and all constants and structures used by Unix programs are not available under Windows.

User and group database information These databases are not implemented under Windows and no functions are provided to access them.

Networking Although there is a sockets compatible API specification for Windows [HTA+93], implementations supporting it are not part of the standard Windows distribution.

Accounting No resource accounting is available under Windows and therefore no system and library calls are provided to access the relevant information.

System management As might be expected system management under Windows it totally different from Unix and therefore the relevant system calls (e.g. *reboot*) are not provided.

C Library differences The Unix C libraries are richer than the libraries normally available for development under Windows. Regular expression handling, directory access, screen handling (*curses*), and multiple precision arithmetic functions are not provided. The C locale specific functions are not integrated with the locale facilities available under Windows.

Many of the problems described above should cease to be issues when developing for the Windows-NT operating system which offers POSIX compatibility.

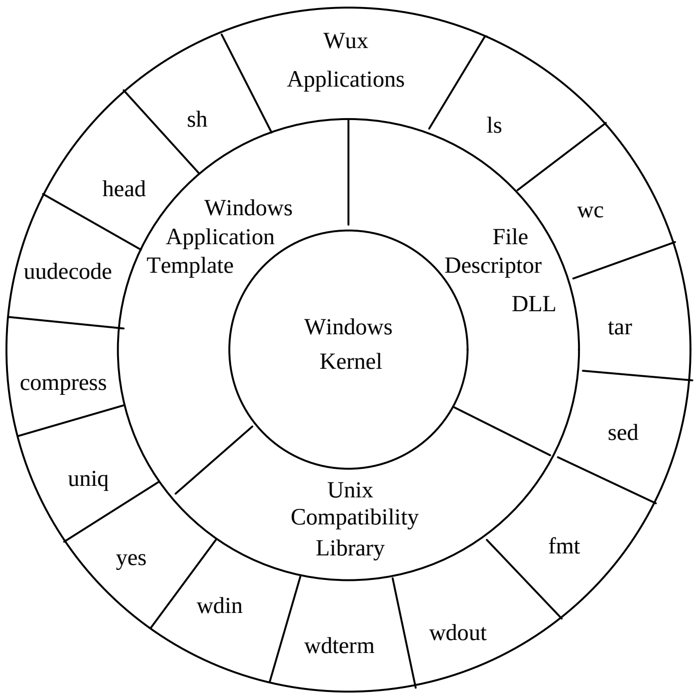


Figure 3: *Wux* structure.

3.2.3 Machine Portability Problems

The third area of problems that arise in trying to port Unix applications to the Windows environment are general machine portability problems. Some are triggered by the "memory model" chosen for compiling the application. The memory model describes the scheme used to get around the addressing problems caused by the 16 bit address segmented architecture of the earlier Intel processors (from the 8088 up to and including the 80286). Different memory models offer different tradeoffs between code or data maximum size, and time or space efficiency. Since the choice of a memory model determines the properties of C pointers, some memory models reflect better the assumptions made by some Unix programs than others. The problems most commonly exhibited are:

Integer size Windows development is still mostly done using 16 bit compilers in order to maintain compatibility with 80286 based platforms. Many Unix programs are coded with the assumption that the integer size is 32 bits and misbehave when compiled for Windows. Under certain memory models the size of pointers is 32 bits. Some programs assume that pointers and integers are of the same size, and therefore fail to run correctly.

Memory size Some memory models allow only 64K of data or code. Many Unix programs assume that a large (often virtual) memory pool is available and terminate with a memory allocation error. Furthermore, unless a severe drop in efficiency can be tolerated, it is not possible to create individual data objects (e.g. a structure, or an array) larger than 64K. Functions whose code would be larger than 64K can not be compiled.

Machine dependencies Some older Unix programs depend heavily on a particular machine, containing VAX assembly instructions, or native code generators.

The integer and memory size problems will be solved once development shifts to the newer 32 bit compilers that produce code for the 80386 and newer processors.

3.3 *Wux* Architecture

The *wux* Unix tools run on top of a set of support libraries that act as intermediaries between the tools and the Windows kernel (Figure 3). The support libraries consist of a statically linked library that provides Unix compatibility

int wux_open(char *fname, int flags, int mode);
int wux_creat(char *fname, int mode);
int wux_pipe(int fd[]);
int wux_read(int fd, void *buffer, int count);
int wux_write(int fd, void *buffer, int count);
int wux_readv(int d, struct iovec *iov, int iovcnt);
int wux_writev(int d, struct iovec *iov, int iovcnt);
int wux_close(int fd);
int wux_lseek(int fd, long offset, int whence);
int wux_fstat(int fd, void *buf);
int wux_isatty(int fd);
int wux_getdtablesize(void);

Table 1: Unix compatible *wux* functions

Function	Description
int wux_fdref(int fd);	Increment the file descriptor reference counter.
int wux_clmark(int fd);	Mark file descriptor to start up a *wdin/wdout* process if used for I/O (lazy process startup).
int wux_fileno(int fd);	Return the underlying MS-DOS file descriptor (if fd does not refer to a pipe).
int wux_setexitenv(int fd, jmp_buf env);	Associate a file descriptor with a *longjmp* stack environment for process termination.

Table 2: New *wux* functions.

replacing or adding a number of functions to the standard C library, and a DLL that provides the file descriptor interface including support for the *pipe* system call. In addition, a standard application template is provided that is linked with all *wux* applications to provide the initialisation, message processing, and termination functions required by the Windows environment.

The *wdin*, *wdout* and *wdterm* windows described in section 2 are implemented as separate Windows applications that respectively generate output lines from the user interaction, display input lines on the screen, or combine the two functions.

3.4 The File Descriptor Compatibility Library

The *wux* file descriptor library is layered on top of the compiler C library to provide functionality similar to that of the Unix system on top of Windows. All calls to the C library are intercepted and handled by the *wux* library. The function substitution is implemented by a set of *#define*s in a header file which is included by all other header files. Table 1 contains the Unix system calls handled by *wux*, while table 2 contains the new function calls that have been introduced. *Wux* maintains in shared memory an internal table of file descriptors which contains the information needed to implement pipes. Calls referring to real files are passed down to the C library, while calls that refer to pipes are handled internally. Inter-process communication is handled using shared global memory. Pipes are implemented in a manner analogous to the implementation described in [Bac86, pp. 113-116]. Processes sleeping on empty or full pipes loop in a Windows *PeekMessage/GetMessage* loop in order to handle process messages and allow the Windows non-preemptive multitasking to execute other processes. We decided to program pipes at a low level instead of relying on the IPC mechanisms provided by Windows in order to avoid unwanted interactions between the Windows user interface messages and the IPC messages.

File descriptors are indexes to a fixed sized array of structures containing information for every file descriptor. The structure contains bits that indicate whether a particular file descriptor is used, closed, refers to a pipe, and its read/write mode. In addition, every structure contains a reference count which is used to mark file descriptors as unused. For file descriptors that refer to operating system files, the structure contains a handle to that file, whereas for file descriptors that refer to pipes the structure contains a pointer to a buffer residing in global (shared) memory, an offset into that (circular) buffer, the number of bytes residing in the buffer, and a *longjmp* environment used for termination message handling (discussed in section 3.7).

3.5 The Unix Compatibility Library

The Unix compatibility library provides a number of functions found in the Unix C library, but not available under MS-DOS, such as regular expression matching and directory access, together with the Berkeley 4.4 BSD *stdio* library compiled to call the *wux* functions instead of the C library functions. We had to provide our own version of the *stdio* library, because the *wux* function substitution (e.g. calls to *read* mapped to *wux_read*) is performed at compile time.

3.6 The Windows Application Template

The Windows application template provides a standard wrapper for implementing the Unix tools. It is a Windows program that handles window creation, argument parsing and I/O redirection, and creates a process exit environment to be used by the *exit* function and the window kill message handling code. It then invokes the *main* function of the tool that was linked to it. Input/output redirection has to be handled in the context of the process using the files, as file descriptors are not inherited under Windows.

3.7 Control handling in *wux* applications

Handling of control in *wux* applications is complicated, as the normal control flow of the Unix tool must be modified to accommodate Windows message handling. In addition, the C *exit* function is not compatible with the Windows termination sequence, because Windows programs are expected to terminate by destroying the application window and exit by returning from the Windows entry point procedure — which normally contains the application event loop. Figure 4 illustrates the control flow between Windows, the *wux* support libraries, and a *wux* application. On application startup, Windows calls the application entry procedure *WinMain*, which in our case resides in the application template. That procedure performs the startup processing described on section 3.6, and then calls the C *main* function to start the execution of the tool proper. Whenever the tool calls a *read* or *write* function, and these have to wait (due to an empty or full pipe buffer), a *sleep* function is called which loops around a Windows message loop getting messages from the application queue and dispatching them to the appropriate handler function (*WndProc*). It is inside that loop that Windows gets a chance to reschedule other applications as runnable. The handler function in our case also resides in the application template code, and does nothing more than check for application termination, and pass control back to the default Windows handler function (*DefWindowProc*) which does the proper things for most messages (window dragging, iconisation etc.). If the message loop detects a `WM_QUIT` message (which can be generated when the user destroys the application's window, or selects *terminate* from the application's system menu) it terminates the application by transferring control using *longjmp* to the end of the application template main procedure which returns to Windows. *Longjmp* with the same target address is also called whenever the *exit* function is invoked. The two cases are however dissimilar, as the first one corresponds under Unix to the termination of a program by a signal whereas the other corresponds to a normal program exit.

3.8 Implementation of the shell and the *wdterm* applications

The shell and *wdterm* applications are special, because they interact with the Windows environment, and are based on the *wux* implementation.

Wdterm is a Windows application that displays its standard input in a window, and sends user text input on its standard output. When its standard input and output are connected to a shell it forms an interface similar to that of a terminal running that shell. It is implemented using a Windows code template similar to that used for all other *wux* tools, but modified to contain two additional elements:

- **a scrollable list region** which is used to display the user input and command output, and
- **an edit line** which is used to enter new commands or text.

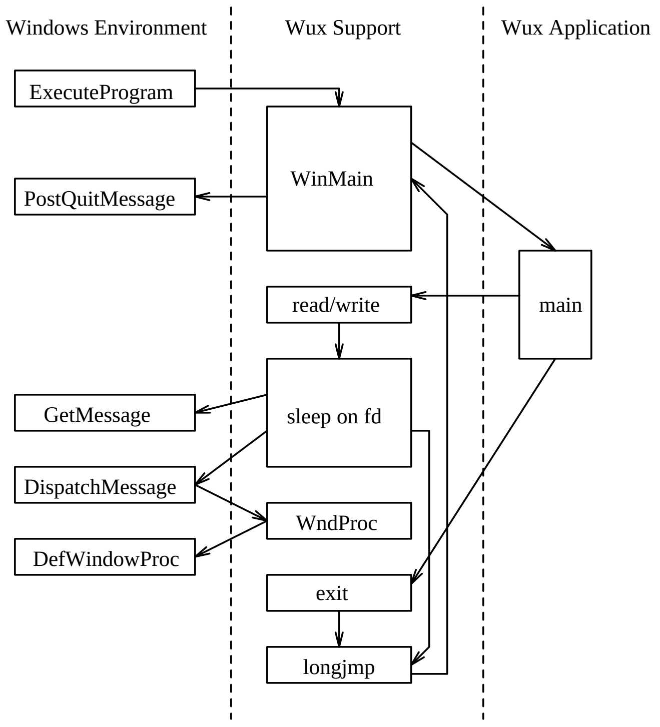


Figure 4: Control flow between Windows, the *wux* support libraries, and a *wux* application.

A single function (*addline*) is used to append lines into the scrollable region. When the region grows to a certain size, every append also deletes the topmost line in order to limit the amount of memory required. The message processing function of the application is modified to provide the scrollbar functionality, the ability to re-execute commands from the scrollable region by double-clicking them, and a handler for the "return key pressed in the editable region" message. That handler just gets the user input from the editable region, appends it to the scrollable region and prints it on standard output:

```
case WM_COMMAND:
    switch (wParam) {
    case IDOK:                      /* Return key */
        (void)SendMessage(from, WM_GETTEXT, textlen + 1, (LPARAM)(LPCSTR)txt);
        addline(to, txt);
        fputs(txt, stdout);
```

The handling of input to *wdterm* is handled in an even simpler way. After the window is created, the program enters a loop that reads lines from the standard input and adds them to the scrollable region:

```
     while (fgets(buff, sizeof(buff), stdin))
             addline(hWndList, buff);
```

The *wux* message handling assures that messages (such as user input, and scrollbar manipulation) are correctly handled while that loop is executing.

The shell is implemented like all other Unix tools, but uses Windows calls and *wux* conventions to handle command execution and the setup of pipes. In the future we plan to isolate the *wux* dependencies and port one of the publicly available Unix shells to *wux*. All commands are executed asynchronously, because we have not yet implemented a *wait* function call. As file descriptors are not inherited under Windows, input/output redirection, and pipe setup can not be handled by the shell. For this reason pipes are implemented by passing the file descriptor number of the appropriate pipe end to the application's command line using the "$>$&*n*" syntax.

3.9 Porting Applications to Wux

Porting of Unix applications to the *wux* environment is relatively straightforward. The *wux* include files and libraries contain a number of definitions and functions that minimise the porting effort. In some cases absolutely no code modifications are needed. The code is simply recompiled using a standardised Makefile and definition file (a file needed by the Windows linker). A suitable icon should be created before compiling. If the application to be compiled comes with a non-trivial makefile, then the application's makefile needs to be tailored after the *wux* makefile template. Possible compatibility problems can appear:

- at compile time due to missing tools, include files, or undefined constants,
- at link time as unresolved references to unimplemented functions, and
- at run time if some aspect of the Windows environment does not match the application's expectations.

As we port more and more tools, we try to rectify these problems by integrating the solutions into the *wux* environment.

3.10 Performance

Although we expected the performance of *wux* to suffer from the layering approach and the Windows overhead, we were pleasantly surprised to find it comparable to other systems providing similar functionality. Table 3 details the times needed to copy data between two processes using a pipe. These include the context switch time. All measurements were made on the same machine and — with the exception of the MS-DOS case which used intermediate files — no disk activity. The speed of the Linux Unix system was expected since it is coded using the multitasking features of the 80386 processor family. The speed of the *wux* implementation can be explained by the fact that Windows provides only rudimentary process isolation and can therefore perform fast context switches. The slowness of MS-DOS is due to the intermediate files used to implement pipes. Although the tested systems are not directly comparable, the figures provide a rough guide on the relative performance of applications relying heavily on data transfer via pipes.

4 Related Work

The portability of tools similar in scope with those available under the Unix environment is examined in [KP76], who also detail how operating system deficiencies can be overcome. A port of the AT&T Unix tools for MS-DOS machines is commercially available in the form of the MKS toolkit [Gor93]. The offering is complete, well documented and integrated. It does not however provide multitasking and real pipes. A similar but non-commercial port of the project GNU Unix utilities for MS-DOS machines is available under the name "GNUish MSDOS" [OGH$^+$93]. These two ports can run in the Windows environment only in the MS-DOS compatibility box, suffering all the restrictions of MS-DOS. User-level Unix emulation on top of another operating system (Mach) is described in [GDFR90], while [Fra93] provides a detailed analysis and specific solutions on the portability problems that can arise between dissimilar operating systems (Oberon and the Apple Macintosh). Finally, the G shell environment [MLS88] attempts to integrate the Unix tools with the X-Window system.

	Block size in bytes		
	1	2048	4096
Linux	220	325	375
Wux	295	988	1480
NetBSD 386	678	1945	3070
MS-DOS	492	67175	86250

Table 3: Time (in μs) needed to copy a chunk of data.

5 Further Work

Although we already use *wux* for day-to-day work, there are many possibilities to extend it making *wux* a lot more useful. We are currently working on porting more Unix applications to *wux*. As 32 bit compilers for Windows become more mature we hope to be able to port more substantial tools like Perl [WS90] and Jef Poskanzer's portable bitmap collection and integrate them with the Windows environment. Other possibilities we are exploring are a graphical user interface for file selection and command argument specification, the integration of the Unix manual pages into the Windows hypertext help system, and the integration of the Windows clipboard. Having laid a foundation for Unix tools and interfaces we would like to experiment with using the Unix tool-based approach to work with multimedia Windows data.

6 Conclusions

The Windows multitasking features allow the realisation of pipelines of concurrently executing programs. The globaly shared memory provided by the Windows dynamic link libraries can be used as the in-core pipe buffer, and the Windows message loop as a synchronisation mechanism. A careful implementation of a compatibility system consisting of modified C headers, support libraries, and a Windows application template, makes it possible to port a number of useful Unix tools to Windows by recompiling the source code. The tools can be used as building blocks for the creation of a Unix-like working environment. We plan to extend the system providing better integration with the Windows features.

Acknowledgements

I would like to thank Stuart McRobert, Jan-Simon Pendry, Periklis Tsahageas, and the anonymous referees for their helpful comments on earlier drafts of this paper.

References

[Bac86] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, 1986.

[Bou86] S. R. Bourne. An introduction to the UNIX shell. In *UNIX Users' Supplementary Documents*. Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California 94720, April 1986. 4.3 Berkeley Software Distribution.

[Fra93] Michael Franz. Emulating an operating system on top of another. *Software: Practice & Experience*, 23(6):677–692, 1993.

[GDFR90] David Golub, Randall Dean, Alessandro Forin, and Richard Rashid. Unix as an application program. In *Proceedings of the Summer 1990 Usenix Conference*, pages 87–95, Anaheim, USA, June 1990. Usenix Association.

[Gor93] Ian E. Gorman. Building a portable programming environment. *Dr. Dobb's Journal*, 18(5):76–81, May 1993.

[Hor90] Mark R. Horton. *Portable C Software*. Prentice–Hall, 1990.

[HTA$^+$93] Martin Hall, Mark Towfiq, Geoff Arnold, David Treadwell, and Henry Sanders. *Windows Sockets: An Open Interface for Network Programming under Microsoft Windows*, version 1.1 edition, January 1993. Available via anonymous ftp from microdyne.com:/pub/winsock.

[ISO90] International Organization for Standardization, Geneva, Switzerland. *Information technology — Portable operating system interface (POSIX) — Part 1: System application programming interface (API) (C Language)*, 1990. ISO/IEC 9945-1.

[KP76] Brian W. Kernighan and P. J. Plauger. *Software Tools*. Addison-Wesley, 1976.

[MLS88] Rick Macklem, Jim Linders, and Hugh Smith. G shell environment. In *Proceedings of the Summer 1988 Usenix Conference*, pages 15–22, San Francisco, USA, June 1988. Usenix Association.

[OGH+93] Thorsten Ohl, Jean-loup Gailly, Ken Holmberg, Mark Lord, Russell Nelson, Len Reed, Stuart Phillips, Ian Stewartson, and other contributors. GNUish MSDOS. Available via anonymous ftp from wuarchive.wustl.edu:/systems/ibmpc/msdos/gnuish, January 1993.

[SG86] R. W. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[WS90] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly and Associates, Sebastopol, CA, USA, 1990.

Trademarks

UNIX, is a registered trademark of USL/Novell in the USA and some other countries.
Microsoft and MS-DOS are registered trademarks, and Windows is a trademark of Microsoft Corporation.
VAX is a trademark of Digital Equipment Corporation.
80386 is a trademark of Intel Corporation.

Biography

Diomidis Spinellis is currently designing and implementing heterogeneous environment software tools for SENA S.A. in Athens, Greece. Diomidis has an M.Eng. degree in Software Engineering from Imperial College (University of London) and is really close to completing his Ph.D. on multiparadigm programming at the same institution. In the last ten years he has provided consulting services in the areas of CAD, product localisation, and multimedia. His research interests include software tools, operating systems, and programming languages. He can be reached via surface mail at SENA S.A., Kyprou 27, 152 37 Filothei, Greece, or electronically at dspin@leon.nrcps.ariadne-t.gr.