CMU-ITC-88-066

"Make or Take" Decisions in Andrew

James H. Morris

Information Technology Center
Carnegie Mellon University

ABSTRACT

In creating a software system on top of a rich system like Berkeley UNIX,† one has many choices of where to start building a particular facility. In creating the Andrew system we generally chose to maximize our use of existing things in the beginning and gradually replaced components as our understanding increased. In this paper we analyze several examples of this process. Some of the areas discussed are: the programming environment, the file system, protocols, window systems, document editors, the shell, printing, and mail.

In 1982 Carnegie Mellon started the Information Technology Center, an IBM-funded development project aimed at creating a prototype university computing environment that came to be called Andrew [1,2]. I was its first director. The charter was to recruit a team of about 35 people, some from IBM, and build the system over a five year period. There were several expectations of what would be done; but we had a great deal of latitude for both the goals and the methods to be employed.

For many years I have been an admirer of F. J. Corbato who supervised the implementation of CTSS and Multics[3] and Butler Lampson who lead many systems projects, notably the Xerox Alto System [4]. Besides implementing good and useful systems they also wrote some very candid summaries of why they had made certain decisions and how they had fared [5, 6]. If more of us who implemented systems devoted our prose to telling true anecdotes the state of the art would advance faster. In any case, they made a number of observations about implementing software systems with small creative teams which I attempted to apply:

1. Don't count on a half-existent tool for your project. In Corbato's case, this consisted of committing to PL/I as a tool for Multics. Lampson codified such mistakes as "Error 33." For good technical reasons we could have decided to use Modula II as a programming language or Accent [7] as an operating system, but we resisted. I had also seen the programming languages and operating systems intended to support a product actually suck all the energy out of it.

2. While a conservative system design might be the best thing for the users, a university-based project has other constituencies to satisfy: funders, administrators, the general research community, and the development team itself. All of these groups want the project to be innovative. Corbato used this excuse to explain the elaborateness of the Multics design when it might have been more sensible initially simply to port CTSS to better hardware. Our sin of technical ambition consisted of choosing high-function workstations as the student machine rather than IBM PCs.

3. Try to use one programming environment for all aspects of the project, even if the technical requirements suggest using different ones. If one has a limited amount of talent (who doesn't), one would like to have maximum flexibility in moving people around on a project.

4. Have the programming team use the system they are developing; it helps ensure that it will be usable.

† UNIX is a trademark of Bell Laboratories.

These considerations suggested using UNIX as the operating system for the student computer and the programming environment for the project. Added to these were the compelling arguments about portability and openness. Finally, there was the rather daunting managerial task of forming a team and showing some results quickly; both external and internal pressures called for getting a system deployed at CMU within a few years. If we started designing something from scratch or depended on some less well-known technology we might have spent a few years just getting things sorted out. We felt it was better to get a system out earlier and improve it as we learned more about the real requirements of the users. So off we went into the exciting world of UNIX workstations.

Partly because of this choice, the creation of Andrew has been an exercise in software evolution. The recent IFIP congress had two interesting papers that endorse such an approach. Barry Boehm, the father of software "creationism", better known as the water-fall model, suggested that building prototypes and re-using software might be a good idea [8]. He codified this idea as the "spiral model." Fred Brooks expressed a more radically evolutionary viewpoint: "The building metaphor has outlived its usefulness...[Software] is grown, not built ... Each added function and new provision for more complex data or circumstance grows organically out of what is already there." [9]

Only an open, flexible system like UNIX permits this sort of approach. The fact that all the software is available in source form is crucial. A vital part of the software development process is *learning*; the development team needs to learn how the existing things work before they can build new, better ones. It is not sufficient simply to use a particular facility, one needs to be able to study it, modify it, and replace pieces of it. We went through this process with many pieces of UNIX; and this paper is an analysis of how it worked.

1. The Programming Environment

We began with the basic UNIX programming facilities: the C compiler, *dbx*(1), EMACS, *make*(1), and RCS. We have always attempted to get facilities from elsewhere and adapt them when necessary rather than devote design effort to new ones. Shell scripts were written to make RCS work in our distributed environment; but we were able to throw them away later when our central file system began to work. We created a variant of Make that supported simultaneous construction and installation of modules for different machine architectures. We replaced DBX with the Gnu debugger, GDB [10]. Eventually, programmers started using our new editor in preference to EMACS. We have recently decided to adopt the Pmak source control system [11], but have taken it as is, without extensive modifications. It is plausible that spending some more time on our tools might have made us more productive in the long run, but there is also a chance we would still be arguing over how to design the tools.

2. The File System

While we successfully avoided the impulse to tinker with the basic operating system concepts of UNIX, the file system part seemed to call for some work. Our decision to support a central service, called VICE, for a campus of several thousand workstations demanded certain changes. Generally, however, we were very conservative about changing things that might make existing programs fail.

The Interface

It seemed obvious that the basic concept behind the UNIX file system – a hierarchy of directories and files with occasional links – was a robust and useful one. We have found it very useful in scaling up to a system supporting many users and departments, distributed over many file servers. However, it should be noted that this concept is not immediately obvious to novice users and often causes confusion.

As detailed elsewhere [12], the basic tactic we employed was to perform a minimal amount of surgery on the kernel so that open and close calls re-emerged in a user process that fetched the file from a server to the local disk cache and stored it back to the server. We didn't have the expertise or market ambitions of SUN or LOCUS who substantially reworked the kernel. On the other hand, our plans were more technically aggressive than UNIX United [13] which took a minimalist approach: linking the file systems of relatively autonomous systems.

Obeying the official semantics of the file system doesn't prevent problems, however. Unavoidably, we changed the practical effect of some kernel calls. For example, many UNIX applications fail to check the error returns from *close*, and under VICE there are many times when such an error return occurs. Generally, even though the semantics are the same one cannot employ techniques based on the assumption of fast disks are available when files are shipped over the network.

Besides supporting all the existing UNIX calls we replaced global and world file protection by a scheme based upon access lists. This was deemed important for a large campus system. This addition has been helpful, but sometimes confuses users because it works independently of the normal UNIX protection; little attempt was made to preserve UNIX semantics.

File Servers

At an early stage we decided that file servers were to be special machines, under administrative control of an operations staff. This was done as an overall simplification for the sake of reliability and security. Therefore, there was no requirement that they support general timesharing access. Thus, in principle they needn't run a UNIX operating system at all; they only need to respond appropriately to VICE requests. However, following principle number 3 we choose to adhere to UNIX.

The first version of the file server leaned very heavily on UNIX facilities. We ran a standard UNIX on each file server. Each workstation had a representative process running on any file server it dealt with. This simple structure saved our creating our own process mechanism, and allowed us to lean on some of the other process management facilities of the system. In a similar spirit, we represented the virtual file sub-tree that a file server held by an isomorphic UNIX file tree on that machine's disk. There was actually a second isomorphic set of files containing other information about files and directories necessary for VICE but not supported by UNIX, e.g. access list information and directives pointing to other file servers.

By the time the first version of VICE was operative, it was obvious that it could be greatly sped up. Several things were done to good effect as detailed in [14], including a reduction in our straight-forward use of the UNIX file system. The one-process-per-workstation scheme was dropped; this reduced process switch overhead and allowed file locking to be performed in a common process. A completely new directory scheme involving mountable volumes was implemented [15]. This allowed us to tune the directory lookup algorithms and avoid the double structure mentioned above. It also required us to implement our own processes for reconstructing the directory structures after a mishap.

In sum, this two stage implementation process worked out very satisfactorily. Trying to do it all at once would have been too hard, especially considering that the team was not particularly expert in UNIX in the beginning. We doubt that anyone could have predicted where the performance bottlenecks would be without building a prototype.

A case can be made for a third stage of implementation which divorces itself from UNIX almost entirely. Although the performance of file servers is now satisfactory, their mean time to recovery is terrible. With each server supporting about 1 GByte, the amount of time for a conventional UNIX salvage operation (*fsck*) is nearly 40 minutes. This salvage is almost always necessary, no matter how seemingly benign the crash, because of UNIX's somewhat casual write-behind method of storing files. There are many more dependable methods of running a file system.

Protocols

One of the major advantages of UNIX for us was that it had an implementation of TCP/IP, the most popular protocol at Carnegie Mellon. Our first pass at the system used TCP sockets for transferring files. We built various remote procedure call (RPC) protocols on top of TCP/IP to support the file system and other activities.

After the first implementation of VICE we discarded the TCP layer and implemented two successive versions of RPC directly on IP. This was made necessary by limitations on the number of TCP connections a process could have. Also we had some of our own ideas of how to make file transfers faster and were able to double the speed.

At the beginning of the project I had hoped that the VICE file system would present a simple, easy to implement, interface on the wire, so that nearly any machine could connect to VICE. There are currently many more IBM PC's and Macintoshes available to students than UNIX workstations, so linking in non-UNIX machines somehow is essential. However, depending upon the established communications mechanisms of UNIX, especially sockets, prevented us from accomplishing this directly. Instead we needed to install various intermediate server machines which ran UNIX, linked into VICE, and spoke a special protocol to the non-UNIX machine. This has been accomplished for PC's. We are still working on a similar solution for Macintoshes.

3. The User Interface

"But, UNIX is not a user-friendly system for general campus use," said the Vice Provost of Computing. "Not to worry," said I, "We're only going to use it as the operating system and will paper over all the nasties." The idea was firm in my mind, if not everyone else's that UNIX and its various tools were, a mere transitional stage on the way to a truly user-friendly system. This plan was even enshrined in the name of our user interface sub-system, VIRTUE, standing for "Virtue is reached through UNIX and EMACS."

The Window System

Given a large bit-map display and UNIX, the first step is fairly obvious: let each application process talk to the user through a different window on the screen. This natural idea has served well because it made things more intuitive for the expert and novice alike. The expert, who already understands processes, immediately knows the purpose and use of a window. On the other hand, the novice has a visual manifestation of the otherwise mysterious thing called a process. For example, the idea that one process might be stalled or broken even while others are operative is easily reflected in the fact that its window is non-reactive or even disappears. Various process-related control features – priorities, kill operations, etc. – can be associated with windows and save the user learning the various shell arcana.

In later versions we employed some processes that controlled multiple windows and found the users became confused, asking questions like "Why can't I read my next message while the send window is sending a reply to the last one?" or "Why does zapping one window make two disappear?" We would avoid this mode if UNIX processes were not so expensive.

The next choice we made was less obvious: make the window system itself a user process. The basic task of virtualizing the display is obviously an operating system function, so one can argue that the proper thing to do is add the new facilities to the kernel as one would any other driver that gives shared access to a device. However, we were reluctant to fiddle with the kernel for all the usual reasons, including not having access to the sources on certain machines we used. Also, the recently released 4.2 socket mechanism offered the chance to coordinate processes closely. The general idea is that an application writes to the screen by sending commands down a socket and receives keyboard and mouse input by reading a socket. Subsequently the X [16], NeWS [17], and GMW [18] window systems pursued the same strategy with some success.

The obvious objection to the scheme is that it limits the speed with which one can paint the screen. However, that problem was overcome by batching commands. All the commands that the application process sends to the display are actually saved up in its process memory until it issues a command that demands a response from the window system. It turned out that even a 1 MIP machine had sufficient power to produce respectable results using this scheme for text and line drawings. The hardest things to deal with are complete raster images, but even these can be painted fairly quickly on faster machines, especially if one cheats a little by introducing sockets that allow the software equivalent of DMA; i.e., transferring large blocks of memory between processes on the same machine.

A major problem with this approach has been mouse tracking. Since a scheduled process is responsible for moving the cursor on the screen when the user moves the mouse, the cursor occasionally freezes. This destroys the user's illusion that the mouse is an effective pointing device and is very frustrating. Of course, it would be possible to put basic mouse support into the kernel; but we have not done so for the the powerful reasons listed above. Also, we have created elaborate conventions for changing the cursor shape based on its region that a self-respecting kernel might not support. Machines without cursor hardware flash the cursor when things are drawn under it. If the window system is de-scheduled when the cursor is out, this can be a problem.

A more general form of this problem is represented by the kinds of things a Macintosh can do easily. For example, MacPaint allows one to snip out an arbitrary shape and mouse it around the screen smoothly. We have yet to support continuous scrolling in which text rolls by until the mouse button is released. Because people assumed rubber-banding would be slow, it was two years before anyone tried it in the Andrew window system. Once someone figured it out, however, several applications did it. Such challenges can often be met with some work by a clever person who can understand UNIX signals and how the window system deals with them, but the number of such people is vanishingly small. Window systems that allow one to download programs make such stunts somewhat easier.

A significant, much heralded advantage of these window systems is that they allow one to execute the application process on a different machine from the one with the user's screen. This is a direct benefit of using the 4.2 socket mechanism. We have used this feature a great deal in Andrew because we have a large network of machines and a common file system. It is very common for people to use two or three machines at once [19].

Another fact we occasionally ponder is that the UNIX process scheduler still thinks its running a timesharing system and might be improved for workstation use. However, we don't have any clear notions of how it should change.

While virtual memory and multi-processing may have intrinsic value for the user, they present difficult challenges to the user interface designer. Those features take control of performance away from the him. He may get his application running perfectly, but then someone will try to use it an overloaded workstation and he can't stop them. It's a little like a movie director trying to design a film for which every local projectionist can change the speed of the projector capriciously. Even on an unloaded workstations, the raw process-switching time for UNIX processes precludes our achieving certain effects. Choosing to make the window system itself a scheduled process exacerbates the problem.

The Shell

Naturally, we began by making the shell into a window. Creating more shells created more windows. Thus most of the fiddling around one does with multiple shells became more intuitive for novices. We actually overlaid the shell with another process that maintains the command/response history as an editable text document and thereby provides some of the advantages of paper teletypes as well as the cut and paste operations. Of course, these features already existed in the shell facilities embedded in EMACS.

The more interesting question is why we never moved beyond the shell to provide an icon driven interface as found on the Macintosh Finder. We made two attempts at Finder-like systems, but neither displaced the shell.

The first was Don Z which presented UNIX files in the a Macintosh style. Each subdirectory was treated like a Macintosh folder, and each file was represented by an icon based upon its suffix. Double clicking an icon would invoke the proper program upon it; e.g. a text file always carried the suffix ".d", had a icon like a text page, and invoked the text editor. It even went so far at to be proactive: small speech balloons occasionally arose from random icons inviting the user to activate them. It was table-driven: one could could specify the icon, the command to be invoked, the speech balloon, etc. for classes of file names based on a template. One problem was that Don Z had to deal with directories containing huge numbers of files and the iconographic representation completely broke down. The normal searching commands of the shell were more appropriate. Due to other priorities, this experimental system was never adequately developed.

The second interface, Bush, took the approach of drawing directory structures like trees without any fancy iconography. It coped better with large directories and has turned out to be a useful tool for certain directory maintenance operations. Bush does not attempt to understand the types of files.

One reason neither of these could completely displace the shell is their lack of a command script facility. In a variety of systems I have observed, point-and-click-based command drivers, no matter how attractive, have not been able to displace conventional shells because they offered no storable language.

Obviously, one can create them; but it is not as obvious as simply storing and re-executing the same language as the user types. It requires a little design work to map every icon-tweaking action into a storable, linear command. There is a deeper problem, however: if the users all get into the habit of using mouse based commands, they are less likely to learn the linear forms quickly [20].

The worst effect of our continuing to use the C-shell is its consummate weirdness. It has been reported extensively elsewhere how novice and not-so-novice users have gotten totally confused and lost significant amounts of work because a mistyped character invoked an obscure feature of the shell. One doesn't have to iconify things to overcome this problem. The command languages of MS-DOS and TOPS20 show that a relatively safe command processor is possible.

Given a finite amount of talent and time the question of what gets built is one of priorities. We generally favored doing things that added functionality to the system. After our initial work on a window system we decided to adopt the X window system and ignore the command executive problem. We chose to devote much more of our effort to the creation of user interface toolkit and the document editor it supports.

Document Editors

Naturally we began the project using EMACS as the basic text editor. It was in universal use at Carnegie Mellon. It does as well as one can do for character based displays like VT100's and H19's. The only compatible extension one can make is to allow positioning with the mouse. It didn't handle multi-font text or things like drawings and rasters at all; and it wasn't obvious how to extend it gracefully, so we began work on a completely new document editor early.

The first version was called EditText and dealt only with multi-font text, justified in various ways. Menu commands and mouse based-selection were used. There were also facilities for sub-dividing documents into panels. The major accomplishment was simply to paint text on the screen quickly. We used the basic underlying text model of EMACS – a document is a long sequence of characters -- but added in-line control characters to signify the beginnings and endings of various styles, e.g. bold. EditText, like Bravo [4], is a sort of mock WYSIWYG editor: the document on the screen contains multiple fonts and formatting similar to what one would see on paper but the appearance is not strictly tied to its eventual printed form. It uses a built-in algorithm to display text on the screen, wrapping lines at window boundaries, and placing text on a screen-pixel basis so as to improve its readability on the screen. The document is unpaginated, like a galley proof. Printing is accomplished by generating *troff* and using the UNIX printing software as is. Thus page breaks, hyphenation, and other things are left to troff. It is possible to view the troff output on the screen, but it can't be modified in that form.

In the process of creating EditText, some popular features of EMACS were omitted. Multiple buffers seemed unnecessary since one could use multiple windows. The programming language, MockLISP, was not implemented. It might have been simple to replicate MockLISP as long as we were dealing only with text, but we were looking forward to more variegated document elements and didn't see how to meaningfully extend the language. However, EditText did support most of the popular EMACS keyboard commands. Most of the common commands could be invoked by menu, but many more could be invoked by keyboard.

The one-process/one-window model actually to led to severe confusion when people used two different editor processes to edit the same document. Possibly because of prior experience with EMACS, they naturally expected edits on either window to effect the same document and were rudely surprised when only the changes to one were saved.

EditText was successful as a document producer of the MacWrite *genre*. Many people used it to produce nice looking documents. However, it was not deemed capable enough to displace existing tools. It could not replace EMACS for programmers and other power-typists because it was a little slower and was not finely tuned for program development. The programmers had become quite fond of a MockLISP package, Electric C, that aided with various syntactic details of the C language; because there was no MockLISP it could not be imported. There was never any attempt to replace troff, Scribe, TeX, or anyone else's favorite document compiler. However, because it used troff as a back end and allowed one to pass through troff directives directly, it seemed like a good tool for troff users; they could use the WYSIWYG features of EditText as far as they went and supplement them with raw troff as needed..

Several specialized editors for drawings, equations, and rasters were also created in the first few years – there were obviously none available in UNIX. The drawing and equation editors used *pic* and *eqn* to print things in the same way as troff.

A second editor, EZ, was created based upon our experience [21]. It was based on a new, more general object-oriented toolkit that allows arbitrary nesting of different document elements. Currently, there are document elements for text, drawings, tables, equations, spread sheets, rasters, animations, and C program text. It added the EMACS multiple buffer features and allowed entirely separate windows to be run from the same process. Aside from being a major advance in terms of multi-media documents, it is preferred to EMACS by many C programmers.

Thus far, the original troff back end printing strategy has been retained. This has been a seductive path of least resistance, but has been unfortunate for a number of reasons. First, the limitations on what troff *et al.* can print has limited our aspirations. For example, thick lines and filled regions are not permitted in any of our drawing editors. Second, the reliability of printing has been adversely affected. Added to the expected problem of dealing with many types of printers we have the difficulty of coping with the foibles of troff as distributed by three different vendors. Also, the programmers generating the printing directives are not entirely familiar with the many features of the document processors so sometimes fall into traps; e.g., a document with a single quote in it recently caused problems. Furthermore, a naive user will occasionally stumble into troff to his regret – often by copying part of a document from someone else who might have known what they were doing. Tracking down printing bugs is a nightmare when there are so many poorly understood steps in the process. Thirdly, depending on UNIX printing software has given us an excuse to avoid the significant problems associated with creating a more faithful WYSIWYG editor – something that many users strongly desire. As long as a black box like troff controls the printed output there is no way to present a faithful interactive rendering of a printed representation.

A General Problem

The managerial problem is crystallized by the following dialog I recently had about our editor:

Q: How do I do a global replace of "UNIX" by " UNIX"?

A: The best I can think of is to save the file and run the following commands on it:

```
sed -e "s/UNIX/\smaller{UNIX}/g" file.d >foo
mv foo file.d
```

This answer is perfectly reasonable and satisfactory if one is a UNIX person, but it isn't considered a good answer for the freshman History major. Given that our goal is to make an editor that serves the needs of novices, I would have been (strangely) happier with the answer "You can't do it today," because then someone might someday spontaneously add a facility to the editor to allow it. There are many of examples similar to this one. The very openness and richness of UNIX means that clever people can find many ways to solve a given problem, so the hope of tricking such clever people into providing solutions for less clever ones is vain. "Why don't they just learn UNIX," has been muttered more than once.

4. Mail

Mail delivery in a workstation environment is problematical. Sending mail directly to the addressee's personal workstation often fails because it is turned off or in a bad state. Because of this, many communities still seem to use their timesharing systems for reading or sending mail long after other activities move to workstations. The timesharing system is usually up, has an established address, and has familiar software.

We started off using a common machine for mail activities, but as soon as we had a common file system we were able to do mail processing on local machines. We began, like most other UNIX groups, using *sendmail* as the routing tool. Every machine ran a copy of it although the controlling file as well as all the spooling directories were kept in VICE. However, we found this solution extremely unsatisfactory. The program was extremely unreliable and consumed significant processing power to do simple things. When anomalies occurred it was hard to reproduce or diagnose them because they occurred on machines remote from the maintainers. Furthermore, since a large percentage of mail sent from Andrew workstations could be expected to stay within Andrew, elaborate and flexible routing methods were not needed so often. Also, we wanted to try some different ways to handle bulletin boards and distribution lists. Finally, the rigid UNIX addressing rules and cumbersome error reporting for mail were irksome. In general, UNIX's mail facilities were considered too primitive for the purposes of Andrew, so there was never any question of retaining them.

Over a period of a few years we made several staged changes to our mail transport system, virtually eliminating the usual UNIX mail facilities. Today it works approximately as follows: [22]

Sending

1. A user invokes a mail sending operation from a document editor.
2. A preliminary screening of the addressees is done using a data base called the White Pages. If any address is implausible or ambiguous based on local knowledge the user is helped to correct it immediately.
3. The message is stored in his personal out basket directory.
4. A background process on the sender's workstation examines each addressee for the message. If an addressee is an Andrew user, the message is written directly to his mailbox; otherwise, it is put into a queuing directory to be examined by a post-office machine.

Receiving

A special indicator window tells a user if he has mail waiting in his mail box. When he invokes the the *new mail* operation in the mail reading application, the message is moved from the mailbox to his mail database.

A post office machine's major duty is importing and exporting mail from Andrew. It reads the queuing directory mentioned above and uses sendmail to direct mail to all the various machines on the networks CMU participates in: DARPANet, BITNET, and UUCP. It has the mail address *andrew.cmu.edu* for incoming mail and distributes mail to mail boxes, using the White Pages to identify directories. A secondary job for post office machines is to serve as a fall back delivery system for local Andrew mail. If a VICE file server is down, preventing quick delivery, a mail queue on an up server is found, and the post office machine handles it later. If things get really bad, e.g. the workstation can't reach any file server at all, mail is queued on the sender's workstation and the background process retrys delivery later.

Multiple processes have been exploited to good effect by the mail system. The mail preparation and reading programs can be run in foreground while the delivery mechanism runs in background. More significantly, the basic mail data base process can run on UNIX workstations and be accessed by various mail interface programs on PC's and Macintoshes.

Conclusions

The evolutionary, incremental approach to creating the Andrew system has generally worked out very well. We deployed a system to a small but diverse group of about 50 faculty at CMU in January of 1985. It included the initial versions of VICE, the window system, the first document editor, and the sendmail-based version of the mail system. The various enhancements and replacements discussed above have been occurring steadily since then. Today there are nearly 500 workstations and over 6,000 registered users of Andrew at CMU. Growing the system on UNIX has given us great flexibility and leverage.

However, here are some cautionary notes related to the four precepts of the introduction:

1. UNIX, its utilities, and other things we depend on, have some bugs, but our programming team is ill-equipped to fix them, especially on each of the three different machine architectures we employ. If you are responsible for delivering the total system and choose to build an existing thing, you

inherit responsibility for it but might not have the expertise to cope.

2. Our commitment to high-function workstations, multi-processing, and virtual memory has placed us outside the main stream of personal computer software development. The market power of 11 million installed MS-DOS systems can make good software of certain types appear faster than can any machine feature or programming tool.

3. Having a single programming environment for all parts of the system obviously limits things. Ideally, we would be supplying a transaction-based file system and a Macintosh-like user interface.

4. The development team using the system they develop works perfectly if they are the only ones to ever use it, but when their needs and skills diverge from the intended users trouble can develop.

Finally, there is the old but useful aphorism, "The best is the enemy of the good." UNIX presented us with many good solutions to users needs. If one has higher aspirations, however, it takes force of will to undertake a completely new implementation.

Acknowledgments

Richard Cohn, Fred Hansen, Christine Neuwirth, David Rosenthal, Mahadev Satyanarayanan, and Zalman Stern made some very helpful comments on an early draft of this paper.

References

[1] Morris, et al., "Andrew: A Distributed Personal Computing Environment", *Communications of the ACM*, March 1986, 29 (1)

[2] Morris, et al, "Andrew: Carnegie Mellon's Computing System", *Proceedings* of the *IFIP Congress*, September 1986.

[3] Corbato, F. J., et al., "Multics – The First Seven Years," *Proceedings of the AFIPS Spring Joint Computer Conference*, 1972, pp 571-583.

[4] Lampson, B. W. and Taft E., eds., *Alto Users' Handbook*, Xerox Corp., Sept. 1979

[5] Corbato, F. J. and C. T. Clingen, "A Managerial View of the Multics System Development," in *Research Directions in Software Technology*, P. Wegner, Ed., MIT Press, 1978, pp. 139-158. Also reprinted in *Software Management* (3rd Edition), Donald J. Reifer, ed., IEEE Computer Society, 1986.

[6] Lampson, B. W., "Hints for Computer System Design", *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, November 1985.

[7] Rashid, R. and Robertson, G.G., "Accent: a communication oriented network operating system kernel", *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, December 1981.

[8] Boehm, B. W., "Understanding and Controlling Software Costs", *Proceedings of the IFIP 10th World Congress*, 1986, pp. 703-714

[9] Brooks, F. P. Jr., "No Silver Bullets – Essence and Accidents of Software Engineering", *Proceedings of the IFIP 10th World Congress*, 1986, pp. 1069-1076.

[10] Stallman, R. M., *GDB Manual, the GNU Source-Level Debugger*, October, 1986.

[11] Tilbrook, D.M. and Place P.R.H, "Tools for the Maintenance and Installation of a Large Software Distribution," *Proceedings of the USENIX Technical Conference*, June, 1986

[12] Satyanarayanan, M. *et al.*, "The ITC Distributed File System: Principles and Design", *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, December 1985, pp. 35-50

[13] Brownbridge, D. R. et al, "The Newcastle Connection", *Software Practice and Experience*, 12:1147-1162, 1982

[14] Howard, John, et. al, "Scale and Performance in a Distributed File System", *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, November 1987, pp.1-2

[15] Sidebotham, R.N., "Volumes: The Andrew File System Data Structuring Primitive", *European Unix User Group Conference Proceedings*, August 1986.

[16] Scheifler, R. W., and Gettys, J., "The X Window System," *ACM Transactions on Graphics*, Vol 5, No. 2, April 1986, pp79-109

[17] Sun Microsystems Inc., "NeWS Manual" 800-1632-10, March 1987

[18] Hagiya, M., "Introduction to the GMW Window System", Research Institute for Mathematical Sciences, Kyoto University, Preprint RIMS-566, 1987.

[19] Nichols, David, "Using Idle Workstations in a Shared Computing Environment", *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, November 1987, pp. 5-12

[20] Cohn, Richard, Ph.D. Thesis CMU CS Department, in preparation.

[21] Palay et al., "The Andrew Toolkit: an Overview", *Proceedings of the USENIX Technical Conference*, February, 1988. (this volume)

[22] Borenstein, et. al, "A Multi-media Message System for Andrew", *Proceedings of the USENIX Technical Conference*, February, 1988. (this volume)