Comparing UNIX with other systems

Timothy D. Chase
Corporate Computer systems, Inc.
33 West Main Street Holmdel, New Jersey

The original concept behind this article was to make a grand comparison between UNIX and several other well known systems. This was to be all encompassing and packed with vital information summarized in neat charts, tables and graphs. As the work began, the realization settled in that this was not only difficult to do, but would result in a work so boring as to be incomprehensible. The reader, faced with such a wealth of information would be lost at best. Conclusions would be difficult to draw and, in short, the result would be worthless.

After tearfully filling my waste basket with the initial efforts, I regrouped and began by asking myself why would anyone be interested in comparing UNIX with another operating system? There appears to be only two answers. First, one might hope to learn something about UNIX by analogy. If I understand the file system on MPE/V and someone tells me that UNIX is like that except for such and so, then I might be more quickly able to understand UNIX. This I felt was an unlikely motivation. After all, there are much simpler ways to learn UNIX.

Instead, the motivation for comparing UNIX to other systems must come from a need to evaluate UNIX. If we are aware of the features or short comings of other systems, then we can benefit by evaluating UNIX relative to those systems. Choosing an operating system or computer is a major decision which we can benefit from or be stuck with for a long time. So I decided that this article should discuss some of the important UNIX issues in the hope that it might help with decisions regarding the operating system. Naturally such a discussion results in comparisons with other systems, but certainly not in the way I had originally imagined.

So why are people interested in UNIX? Conventional wisdom indicates several reasons. First and foremost is because UNIX is a vendor independent operating system. Many feel that if software development dollars are invested in UNIX, they are safer because UNIX is not tied to any given vendor. If, for example, you write a program in FORTRAN for your HP3000 which makes use of the features of MPE/V, you are going to have to spend some money to move that application to another vendor's hardware. To a large extent, HP has you trapped in that you'll often choose to put up with things that make you unhappy rather than switch to another computer vendor.

The dream of UNIX is that it makes computer hardware more of a commodity than it is now. You can develop UNIX applications then shop for delivery vehicles. This may or may not actually be true, as we shall see, but it most certainly is not true for proprietary operating systems like DEC's VMS or HP's MPE. An excellent model for this approach is the IBM developed PC. Because of the PC's open architecture and generally available operating system, you can now purchase PC's from literally hundreds of hardware vendors. Currently, one can secure a 1 megabyte PC with a color CRT and a 40 megabyte hard disk for under $2,500. It has no nameplate on the front, but it runs Lotus as well as the IBM original. It's the user's dream that UNIX brings this state of affairs to the minicomputer.

The second important feature which UNIX promises is people portability. Nowadays it's not enough to get a COBOL programmer, you need to get a VAX/VMS COBOL programmer. Each proprietary operating system has its own command language, editors, compilers, file system and system services. Only the most superficial of programmers is not effected by the host operating system.

With UNIX, that changes. We have programmers working for us who don't really care what machine they're working on. All they know is that they are working on UNIX. As program managers, this is a powerful incentive to use UNIX. It's especially true for managers of internal service organizations supplying general programming talent. Rather than force the company to standardize on a given vendor or vendor's processor line, UNIX allows you to address a range of processors with a single pool of programming talent. Clearly a cost effective use of an expensive commodity.

A third reason to consider UNIX as an important system is the available software pool. Because UNIX is widely available, it behoves third party software vendors to develop products which are UNIX based. Because UNIX is installed on so many systems, it gives software houses a large potential market. This naturally results in more packages for you to choose from. In addition, competition causes the resulting packages to be high quality. Contrast this with proprietary operating systems. The HP3000 MPE system is actually too small to induce big software suppliers to make the (often extensive) changes necessary to port their packages. The smallness of the potential market associated with proprietary operating systems limits the choices you have and effects the overall quality of available third party packages.

So, our comparison of UNIX with other operating systems is motivated by the hope of evaluating the goodness of UNIX to see if it might be a useful way to gain vendor independence, people portability and access to existing software.

As easy as goodness is to talk about, it is quite another matter to define it. This is because goodness is, of course, relative to how the system is going to be used. It is generally felt that UNIX lacks certain features which would make it a good real time system. Does that make UNIX bad? If you don't care about real time features then it certainly does not effect you. So, although we will evaluate UNIX relative to other systems, our conclusions will only be valid with respect to your scope of application. If you live in Oneonta, New York and there is only an HP office near by, then you may not be really interested in UNIX's vendor inde-

pendence while a customer in downtown Manhattan might find it very attractive.

Open versus Proprietary

The first UNIX feature to be compared is its openness versus other operating system's proprietary nature. In a nutshell, UNIX's openness stems from the fact that it was provided in source form to most users and is available in some incarnation on many different computers. (For security reasons, UNIX sources are no longer automatically shipped with the system. Now, in order to obtain sources one must purchase a source license.) Written primarily in a machine independent way in the C programming language, UNIX trades off wide availability against somewhat diminished performance. It should be unsurprising that a proprietary operating system executing on the computer it was designed for should out perform UNIX. Because UNIX was written with portability in mind, it cannot take advantage of the special features available on any given computer. UNIX must first abstract the feature and then implement it in such a way that it is available on many different processor designs.

In practise this is so difficult that it borders on impossible. The result is that UNIX has different flavors which address differing underlying hardware capabilities. For example, in VAX/VMS, the operating system is strictly demand page and the hardware is designed to accommodate this. UNIX, however, must cope with machines which can support demand page organization and those which cannot. As a result, UNIX may be run in either paged or swapped mode. This fact causes a difference between individual UNIX installations. Fortunately, the number of application programs which can (or must) differentiate between the two modes is small. Still, the pure concept of a universal UNIX has to give way to questions of efficiency.

Although next to impossible to verify, one confidential study I have seen indicated that a VAX/780 running UNIX could support 32 users while the same machine running VMS could support in excess of 48. The point being not a quantification of the difference in performance, but rather verification of the existence of a difference.

For many applications, however, the ability to execute on different machines unchanged is more important then a slight decrease in performance. But, there are those who don't find UNIX all that pure on the different machines it runs on. A recent issue of Datamation quotes P. J. Plauger as saying "Currently, there are so many dialects [of UNIX] that the idea that there is one UNIX is silly". Plauger, who used to work with Bell Labs during the UNIX genesis appears to know what he's talking about. A partial listing of UNIX system currently in use includes the Sixth, Seventh and Eighth Editions (sometimes called Versions 6, 7 and 8), Programmer's Work Bench, System III, System V, System V Release 2, System V Release 3, Berkeley 4.1, Berkeley 4.2, Berkeley 4.3, PC/IX, UniPlus+, Ultrix, Venix and XENIX as well as a host of UNE-alikes such as Idris (from Plauger's company Whitesmiths) and Coherent.

In addition, various companies like HP, DEC, SUN, Apollo, Plexus ModComp, etc. all offer standard UNIX with a few enhancements just to make it run better. Advertisements offer

"Standard System V with Berkeley BSD 4.2 enhancements". The net result of all of this is that, although UNIX is conceptually open, pure and portable, the local enhancements tend to make it become proprietary in subtle ways. In UNIX, Local enhancements take two different forms. Usually they are commands which have been added and are therefore available only on the enhanced system. Other, braver, users actually modify the resident operating system. This can result in a UNIX which looks normal, but behaves in distinctly abnormal ways.

Though you probably never thought of comparing UNIX with UNIX as another operating system, you should. Take, for example, standard UNIX (whatever that is) compared with HP/UX. HP's real time enhancements (just to make it run better) include the introduction of more than 10 new system calls not found in System V which were either taken from Berkeley 4.2 or invented by HP. If you use these calls in developing your application, then you'll find that UNIX can be just about as bad as a proprietary operating system when it comes to porting to another vendor's hardware.

In the face of all of this, AT&T magnanimously offered to standardize all of the UNIX implementations by introducing the "System V Interface Definition" and an appropriate set of test programs to measure any given implementation's adherence to the standard. The SVID, as it's called, was met with less than enthusiastic acceptance from AT&T's competitors who narrow-mindedly complained that AT&T was actually trying to control the UNIX marketplace. To address this complaint, the IEEE organized an alternative interface standard based on SVID given the project number P1003. This system is called POSIX; a name derived from Portable Operating System. This name is hopefully far enough from UNIX to avoid the legal wrath of AT&T for trademark infringement, but close enough so that everyone (wink, wink) knows what they're talking about.

The conclusion here is a bit cloudy. Although UNIX does represent a giant step toward a portable machine and vendor independent operating system, there is still quite a ways to go. Trivial programs will clearly port without change. More complex applications which require local enhancements to UNIX will not. In point of fact, it would be just about as easy to write the trivial class of programs to port to any operating system be it proprietary or not. As users, we must remember from all of this that when someone says "UNIX" we might need further clarification.

Some bits of history

How did UNIX get to be so varied? Didn't it all come from Bell Labs? In order to answer this, it's worth our time to understand a bit of UNIX history and compare this history with that of other operating systems.

The UNIX time line begins way back in 1965 when Bell Labs was working with MIT and General Electric on Project MAC. The goal of this effort was to develop MULTICS. A large and complex system, MULTICS never totally met its design goals. Reading about MULTICS provides an interesting insight into UNIX's conception. Many of the features one might credit to UNIX actually came from project Mac.

Bell left Project MAC in 1969 and one of the team members, Ken Thompson, started working on an operating system as well as a *personal research project* involving real time animation in a competitive setting titled Space Travel. This was for an almost forgotten PDP-7 "with good graphics capability." Thompson and Dennis Ritchie implemented the predecessor to UNIX in assembly language on the PDP-7 to enable Thompson to get the Space Travel video game working (One can only guess how history might have been changed had AT&T understood the worth of video games as well as it understood the worth of UNIX.) UNIX was then moved to the DEC PDP-11 and subsequently rewritten in the new C language developed by Ritchie. The rewrite was completed in 1973. The use of a high level language to implement an operating system was unique for the day and would later be one of UNIX's important features.

Because of Federal antitrust rulings, AT&T could not sell UNIX, but it could give it away. In a brilliant master stroke (or a lucky move) AT&T started giving UNIX to universities. The smallness of the system and the fact it was written in a high level language made it attractive for teaching purposes. The fact that hundreds of students were adding to the system and growing to become disciples didn't hurt its current and future popularity.

By 1977, UNIX was being used in over 500 sites. Also in that year, UNIX was ported to the first non-DEC machine, the Interdata 8/32. From that year until 1982, several versions of UNIX were available within the Bell System. These were ultimately coalesced into one system called System III which was offered commercially in 1982. Several new features were added during 1982 and in January of 1983 AT&T offered official support of System V. (The missing System IV was never commercially released and enjoyed fleeting popularity within the Bell System during 1982).

The boys at Berkeley, being an unruly lot, did their own UNIX developing and came up with several additional versions. These were BSD 4.1, BSD 4.2 with the most current being BSD 4.3.

This history tells a great deal about the operating system. First, its design was largely motivated by intellectual curiosity and not market pressures. Unlike proprietary operating systems, UNIX did not have to compete with other venders nor support upgrades from previous systems. Second, even though the basic design for UNIX came from only a few minds, thus insuring conceptual integrity, a great many people have had a go at the system since its beginnings. As a result, UNIX is the product of evolution as opposed to design.

When comparing UNIX to other operating systems, this shows. For example, the UNIX human interface has little uniformity to its syntax and is filled with commands which represent the various implementers pet names or individual senses of humor. Contrast this with the VAX/VMS command language which was designed as a unit, with similar formats and relevant English names for each command. It's argued that these differences are only a problem for inexperienced users, but they are still a consideration. The lack of consistent design is also apparent in the system services. There are some calls which appear to have duplication of function as well as a lack of consistency in doing things. Error condition indication is an example. Most calls return errors in the same way, but there are a few which

don't. Not that this is a serious problem, but it does tend to foster misunderstandings among new users.

UNIX's family history also points out another important problem. Everyone who's ever taken an operating system course knows something about UNIX and there are a great many people who know an awful lot. The result of this is that security on the UNIX system is difficult to control. Contrast this with something like the MPE/V system. Either by plan or carelessness, it's downright difficult to get a detailed picture of what's going on inside the 3000. This makes security a lot easier because there is already a confusion factor about what's happening. With UNIX's public privates, this is much harder. This problem is compounded by the fact that UNIX used to come with the sources right out there for all to see; just pleading to be modified, studied and tampered with.

The security problems with UNIX are horrendous even in UNIX heartland. At Bell Labs in New Jersey there has been a mini-crisis with unauthorized access to internal UNIX systems. Some, in darkened rooms with the backs to bright windows, have even admitted that there is fear for the master sources kept in Short Hills. It would be possible, some theorize, for hackers to subtly modify the system which is shipped thus enabling them to get in any derived UNIX system. This might go unnoticed for long periods with predictably disastrous results. Bell is taking herculean steps to correct the problem, but the fundamental reason for security difficulties remains the basic philosophy of UNIX and its general exposure compounded by UNIX's roots being in educational institutions. Further standardization activities by AT&T and the IEEE may actually worsen the problem.

So the conclusion is that unlike most proprietary operating systems, UNIX has grown by evolution often at the hands of research types. The result is that it lacks an overall consistency. In addition, the fact that UNIX's sources are generally available and well understood by many smart people introduces security problems which are unique to UNIX.

Something has to be missing

When comparing UNIX with other operating systems one thing should quickly strike you as odd. The basic UNIX kernel was implemented over several years by essentially two people. Take this in contrast to something like System/360 from IBM. This (monster) operating system took man centuries to implement as compared to a man decade for UNIX. We can conclude from this that either the boys at Bell are pretty smart when compared with the people from IBM, or that something is missing from UNIX which others thought important to include in System/360. Throughout the UNIX literature the *small is beautiful* theme reoccurs. Admittedly one of the basic system design goals was to provide a minimum amount of kernel function. Non-kernel user programs would be left with the job of providing the real sophistication.

This philosophy has several results. First, it is not true that small and easy to understand necessarily lead to optimum machine usage. Much of the complexity of other operating systems stems from the fact that they are giving the user a wealth of choices which offer various degrees of optimality for different programming situations. A good example of this is the

difference between the process scheduling algorithms found in VAX/VMS and UNIX.

VMS offers the user with a complex set of scheduling techniques clearly breaking the UNIX small is beautiful rule. In fact, the VMS scheduler offers two distinct scheduling principles. UNIX, on the other hand, only offers one. It's much easier to use and, for the most part, transparent to the programmer. The problem is that a real time application has different scheduling needs than an interactive application. VMS offers solutions to both problems at the same time, UNIX offers only the interactive solution.

A second result of the "UNIX philosophy" is that UNIX users tend to view rolling your own as a normal way of dealing with operating system deficiencies. The file system doesn't perform the way you want? Just write your own. Does memory management miss the mark on important features? Just write your own. You either view this aspect of UNIX as great (because you can modify UNIX so easily) or terrible (because you have to modify UNIX so often). I know of few people who get in there and tinker with MPE, VMS, RTE or VM and even fewer who expect that they will have to.

This ability to tinker, often coupled with a real requirement to introduce new operating system features, tends to create a new class of programmer in your organization -- the UNIX OS Guru. Unfortunately, this is just the sort of thing that you were hoping to avoid by going to UNIX. Now, all of a sudden, you have pockets of unique specialized knowledge in your organization. No one really knows what changes were made. You are held captive by subordinates who have the keys to the kingdom. At least with a proprietary operating system you are being held captive by another (large) company with, hopefully, well understood motivations.

The list of features missing from UNIX is often long and may be important depending on your individual application. Most UNIX advocates dismiss "missing features" by telling you that there is some Berkeley version or third party package available which solves precisely those problems. This answer, to me anyway, is admission to a larger problem and that is the rapid proliferation of nonstandard UNIX systems. Among the missing features are the following.

A classic file system.

The UNIX file system is unique among operating systems. It reflects UNIX's original text processing application in the Bell Labs patent department. A file in UNIX is a string of characters with no record boundaries. A file consists of a number of blocks which are hung off of a master block called an inode. The inode block contains pointers to the other blocks. To extend the potential size of files, UNIX provides pointers from the inode block to subordinate blocks which, in turn, contain pointers to the actual data blocks. This results in the odd situation where the first part of a file is slightly faster to access than the last part of the file (last part requires occasional double reads).

Because UNIX pre-reads blocks into a buffer cache, reading a file sequentially results in fast access. As you are processing one block, the system is getting another for you. There

are some problems, however.

Although a conceptually simple file system, UNIX gets into some trouble because of it. In fact, the file system is usually the first thing serious UNIX users begin to modify (just to make it run better). The usual file system problems include:

Too simple a file model. There are times when the classic record oriented file system is just what the doctor ordered. UNIX files have interesting properties, but they can miss the boat when it comes to large efficient data base applications. This criticism stems from the fact that the record model gives the user control over some of the physical attributes of a file which are important to performance tuning. In addition, many operating systems, give important system support to data base functions rather than force them to completely reside in applications programs as does UNIX.

Scattered blocks. UNIX provides little control over the allocation of files on the physical disk. The result is that data blocks are scattered all over the place. This causes multiple seeks when accessing the file. There is no way to cluster the file's data blocks to minimize the size of the seeks either. Contrast this with MPE, RTE, VMS, and other classic file systems. Files, or at least file extents, can be allocated in physically adjacent regions thus minimizing disk head movement. In fact, some operating systems even give you control over file placement with respect to physical disk cylinders so that head switching may be used instead of head movement. UNIX offers no such help. Some program which may be run after the fact allow you to reorganize the disk to provide more optimum file block locations.

No pre-allocation. UNIX gives you disk space as you use it. This means that when writing to a new file, disk allocation overhead will be intermixed with write overhead. There is no way to pre-allocate a file so that the allocation overhead is concentrated in one spot in your application. This problem is especially vexing to real time applications which may need to write data quickly and can't stand the allocation overhead at any particular moment. Because UNIX allows files to have holes in them, you can't simply position to the last byte of the file and write it causing the operating system to allocate all the other bytes. Doing so would make a file with only the last block in it and no others.

Other operating systems provide the capabilities to pre-allocate either all or some of the file when it is created. MPE's preallocation scheme provides a compromise solution which enables the user to anticipate the file's ultimate size and then determine how much should actually be allocated at file creation time.

Small data block size. The block size used for UNIX is either 512 or 1024 bytes. This is fine for interactive applications and text storage, but for large data transfers, it limits data throughput. Other operating systems don't have this problem, because they offer the user the concept of the record as a user controlled block size. User's may define big records which more efficiently accommodate larger individual /O requests.

Asynchronous disk writes. UNIX always performs disk writes to a disk cache. Because of this, the application program is never really sure that the /O has been performed to the physi-

cal disk. In transaction applications which wish to implement bullet proof error recover techniques, this is a difficult situation. Transactions are typically considered complete only when everyone is safe back home on the disk. UNIX prevents this information from getting to applications. Again, other operating systems, have extra features which either disable cacheing or allow the requesting program to specify synchronous writes.

Super block. Important parts of the file system are in memory in a UNIX computer. If the system crashes, the disk and the memory are out of synchronization. This causes damage to the file system. One especially vulnerable structure is the super block. The super block is an in-memory data structure which maintains information about the disk space managed by the system. There is only one super block for each file system (a disk may, however, be partitioned into multiple file systems which are then linked together). This is a dangerous design for systems which must have highly reliable data bases.

Sophisticated process scheduling

UNIX offers only one flavor of process scheduling. This technique was designed to give good performance to terminal jobs in an interactive environment (program development, text processing). Processes are given a time quanta and a priority. Processes are preempted on the basis of priority, but the priority level is dynamic and is automatically readjusted by the operating system when a process is suspended. High priority is given to new processes to insure quick initial response. The priority decays as the process runs (This type of process scheduling is difficult to simulate. The result is that UNIX is hard to model for system response time studies.)

Contrast this scheme with the priority scheme found in the HP RTE operating system. Typical of real time systems, the RTE provides simple priority preemption without priority adjustment. This gives the system the ability to insure processes fixed amounts of non-preempted CPU time when responding to events.

The VAX/VMS operating system, totally ignoring the "small is beautiful" credo, offers both types of scheduling. Processes with certain priority levels are scheduled with time share techniques while 16 priority levels are reserved to implement the real time technique.

Sophisticated Kernel

The kernel organization of UNIX is quite simple, but again, the missing parts are considered vital in some circles. For example, the UNIX kernel is non-preemptive (sounds pretty complex, huh?). This means that once UNIX enters its kernel, it won't interrupt out to begin another kernel task (other than /O interrupt processing) until the current task is completed or until the process which requested the operation is blocked. In English, what this means is that when a process requests a kernel service, the process begins execution in kernel mode. From that point on, the process cannot be preempted by another process in the system until the kernel mode request is either completed or until the kernel decides that the requesting process must wait for some other event to complete.

The net result of this is that UNIX processes can disregard other processes for as long as one second. Again, not something real time enthusiasts rave about in a positive way.

The HP/RTE system has the same problem, but it was written to try and be quick about getting out of the kernel. In addition, RTE has been modified to include a priority interrupt which is a way of actually interrupting the executing RTE kernel to address a more pressing need elsewhere.

VAX/VMS has an even more elaborate solution. It has an asynchronous kernel which is reentrant. It may interrupt itself and go off to process more important kernel functions while in the middle of processing a less important kernel function. This results in a larger more complex kernel, but provides far superior interrupt response and system throughput.

The realization that UNIX is a synchronous kernel, led HP to modify System V when producing HP/UX. They changed the standard kernel code to incorporate interrupt points. These are areas where the kernel may be interrupted by other kernel processes. This does not result in a truly asynchronous kernel, but it is much better than standard UNIX's strictly synchronous design.

Like many other UNIX vendors, HP has taken a hard look at the short comings of UNIX such as those mentioned above. In the case of HP, the HP/UX offering addresses each of the UNIX problems in some way. Although HP is lobbying to have their solutions accepted as "standard" the jury is still out. Every vendor who as tuned UNIX to address its problems wants their solution accepted as standard. It would appear that true standards will be long in coming. It would not be surprising to see that there will be several different standards evolving. Certainly not quite what user's have in mind when they think of portability.

Another final comment is about the richness of the kernel. Other operating systems tend to offer more in terms of kernel functionality than does UNIX. This is often explained by saying that the UNIX kernel offers the base upon which users write applications to perform the real work. This is actually what has happened. The UNIX kernel is surrounded by many man years of excellent software which is available to the UNIX terminal user. The problem is that transaction oriented systems tend to need the functionality at the process level. To have application programs providing features which might more appropriately belong in the kernel, precludes their use from other user processes. Once again, you are forced to roll your own.

And in conclusion...

Upon rereading what I have written detect a distinctly negative feeling. I don't think that this is intentional, because as a programmer, I use UNIX and like it. Perhaps I have presented some of the more negative aspects of UNIX because it is so easy to hear only good things about the system. From what you have read, I hope that you make the following conclusions.

UNIX is definitely a serious force in the market place. There are well over 200,000 UNIX installations. From that consideration alone UNIX must be doing something right.

UNIX does represent our best shot at a vendor independent operating system, but by no stretch of the imagination should you think that UNIX is standard. When you hear someone remark that they are using UNIX, you must ask a number of questions to really understand what they are saying. In addition, standardization itself, which on the surface appears to be so attractive, has its own innate problems; not the least of which is security.

UNIX evolved as an operating system instead of being designed from the start as an integrated whole. This is perhaps true of any mature operating system, but it is especially true of UNIX. As a result, there are areas of inconsistency which can be confusing.

Finally, UNIX is small. This is often confused with charming. UNIX's smallness is as much a result of missing functionality as it is the result of a good sparse design. To some, the omitted parts will not be missed, but to others, UNIX's lack of features will forever label it as a toy system still smacking of its research upbringing.

In all cases, though, it borders on silly to make blanket statements about UNIX. As with art, UNIX can only be evaluated in the context of its intended use. The potential user will gather the facts, weigh them against the application and only then conclude whether or not UNIX is the right or wrong solution.