Heinz Lycklama

UNIX Technology Advisor, 1988-89

UNIX TECHNOLOGY ADVISOR

NETWORKING

New Solution Recreates Old Problem

By Rob Gurwitz

Remember the days of time-sharing a PDP-11 or VAX-based UNIX system with ten or twenty other users? Invariably, at times of peak load, system response would slow to a crawl or the system would crash. People would congregate in hallways, taking an enforced coffee break because "the system was down".

At many sites, the advent of single-user workstations has changed that picture. No longer is a whole department dependent on a single timeshared machine with its load bottlenecks and single points of failure. The decentralization of computing resources concentrated more power in the hands of individual users and allowed more flexibility and insulation from hardware failures.

Add to that the advantages of large, bitmapped displays with the ability to support multiple windows on screen, and the result is a true revolution in individual productivity.

New System Demands

Of course this revolution has come with its own costs: new demands for fast communication, the need to share decentralized data, and the increased administrative burden of maintaining multiple machines where one served before. These needs have increased the burden on the operating system to provide transparent data sharing through new mechanisms. Distributed file systems, of which Sun's Network File System is the most widely used example in the UNIX community, extend the concepts of centralized file management of timesharing days to a networked system of workstations.

With NFS, filesystems, sections of the UNIX file naming hierarchy associated with a logical partition of a disk, can be shared by multiple workstations on a network. An NFS client workstation can "mount" a filesystem which physically resides on an NFS server workstation disk and make the filesystem part of its local naming tree.

The "mount" mechanism for remote filesystems is analogous to the UNIX mechanism for associating filesystems on local disks with a naming hierarchy. Files mounted remotely in this way are then accessible through normal UNIX commands, just as if they were on a physical disk attached directly to the workstation. This type of access is known as "transparent", since no special commands are needed to access the remote files once the filesystem is mounted.

How "Transparent Access" Works

The system calls which normal UNIX file commands (*cat, ls,* etc.) issue are converted in the UNIX kernel to corresponding operations across the network to the server system where they physically reside. A write system call on a remote file gets translated to a remote write operation directed at a process on the server system which actually does a write to the file on its disk. For performance sake, many operations such as reads and writes are not issued one-to-one for every client system call, but the concept is the same.

The distributed file system is only really transparent to the user if the performance of remote access is comparable — it doesn't have to be equal — to local access, and if the server machine where the files are located is operating correctly. The decentralization of the file's users (clients) from the machine where it physically resides (the server) opens up more possibili-

(over, please)

ALSO IN THIS ISSUE . . .

ties for error than when the client and server are the same machine.

In the old timesharing model, the disk could generate an error, or the CPU could crash due to a hardware or software error, but aside from those occurrences a file operation was pretty safe. It either worked, or, in the odd crash, it didn't.

In the new model, the possibilities for failure are multiplied and the failure modes are more complex. Instead of one system, at least two are involved, with a network and a lot of complex software in between. If any link in the chain fails, the operation fails.

With a network and many machines involved, the failures can be frequent and they can be transitory. Even worse, as we shall see, the failure of one system can adversely affect another, even if it is not directly the client in a particular operation.

If At First

In NFS, for example, the requirements for maintaining the communications context (e.g., which files are open, what machines they are located on, how to communicate with those machines, etc.) of the distributed file system are the client system's responsibility. The server system is only responsible for completing the individual file operations coming in over the network. The server maintains no state about the remote clients of a file.

The implication of this scheme is that the client is responsible for retrying an operation if the server does not respond when overloaded or down. This simplifies the design of the server software, but it also affects the error handling strategies on the client and can result in problems for the user. Note that there are other ways to build distributed file systems that place more responsibility on the server. But because of the distributed nature of the system, there are always more chances for errors if the file is remote than if it is local.

With NFS, there are two ways of dealing with server failures. Either the client can retry an operation some number of times and then give up, or it can continue retrying indefinitely, making the command in question hang until it is interrupted by the user or until the server is again available to handle the operation. The infinite retry strategy may seem excessive at first, but imagine running a long, expensive job. Do you really want it to bomb out halfway through its run just because the server crashed and was unavailable for a short time?

For Example ...

My organization uses UNIX workstations as our main computing resource. We have about 30 networked together, plus five large file servers with a total of over 10 Gbytes of storage. Every engineer has a workstation

(continued, p. 4)

The UNIX Technology Advisor is published monthly by MYOB, Inc. It is a publication dedicated to the needs of technical managers working in the UNIX/ open systems environment. POSTMASTER: please send address changes to The UNIX Technology Advisor, POB 955, Hollis, NH 03049. Entire contents copyright 1989 by MYOB, Inc., unless otherwise noted. All rights reserved; no materials may be reproduced in whole or in part without prior written permission from The UNIX Technology Advisor. Subscriptions to the Advisor are available in the United States for US$295 per year. Correspondence on editorial or circulation matters should be sent to POB 955, Hollis, NH 03049. Telephone: (603)465-7825. Letters to The UNIX Technology Advisor become the property of the newsletter, are assumed intended for publication, and may be used by the editors as newletter matter. Letters should include writer's full name, affiliation and telephone number. Letters may be edited to meet space or clarity requirements. Statements of fact and opinions expressed by columnists and writers are their own, and are not necessarily those of The UNIX Technology Advisor. UNIX is a registered trademark of AT&T; The UNIX Technology Advisor is not affiliated with AT&T.

WELCOME TO THE UNIX TECHNOLOGY ADVISOR

By H. Brewster Maule, Publisher

The scene is set for a revolution in computer technology. Companies are moving to integrate their systems across mainframes, minis and micros; open systems technology is the promise that makes it possible.

If you're a technical manager for a vendor or end-user, you're facing a critical juncture. Whether you're deeply involved in UNIX/open systems issues, or on the threshold of converting, you must commit to strategies that are costly in time and effort in spite of the fact that there are no clear-cut technical answers or directions.

The Advisor is intended to help you make the best of an uncertain situation. It will keep you up to date on developments you should be aware of when making decisions you have to live with.

It will feature 12 columns (listed below) that will provide close coverage of technical development efforts and a focus for commentary on, and analysis of, key technical issues facing the industry.

The distinguishing features of columns will be their narrow focus and the fact that contributions for each will be written by up to four different technical people. The writers *work* in the industry, as opposed to following events the way reporters do.

The Advisor will also feature articles by technical people on topics not regularly addressed in columns. Some will be written in response to material that has appeared in an issue. If you feel strongly about material you see here — and we expect you to! — your comments are certainly welcome. They may even reach a wide audience by appearing as articles!

I trust you'll find The UNIX Technology Advisor a thought-provoking and extremely useful aid in your daily work. I look forward to serving your technical information needs for a long time.

By Dr. Heinz Lycklama, Chairman, Editorial Board

When I was first approached by the publishers to participate in The UNIX Technology Advisor, I was immediately struck by two things: first, how controversial such a newsletter would probably be; second, how useful.

In the first meeting of the Editorial Board, we talked about the issue of controversial opinions — the "religious fervor" that's apparent in the UNIX/open systems community.

The technical people I know aren't shy about expressing their opinions on technical issues and problems. And they're likely to disagree with each other — sometimes loudly — on the best ways to proceed.

The factor they have most in common is that they're all well worth listening to. They have faced, imaginatively and effectively, the thorny technical issues that concern us all. Their opinions are valuable because they've considered them deeply and intelligently.

So the guidance you'll find in the Advisor is uniquely useful. It comes from professionals who work on the leading edge of technical developments, technical managers who are deeply committed to solving the complex problems that abound in the UNIX/open systems environment.

I've found my own involvement in the Advisor has been rewarding on both the professional and personal levels. Professionally, it opens another pipeline to the best thinking in the field. Personally, I find the other Board members and the columnists to be stimulating and entertaining as well as informative.

I believe the Advisor will become a significant force in the ongoing UNIX/open systems debate. I trust you'll find it as useful and provocative as I do.

Column Topics:

Academic R&D • Applications • Communications, Interoperability, Networking • Industry forums • Internationalization • Languages/ Software Development Tools • Mail • Operating Systems, Architecture, Hardware • Security • System Administration • Standards • User Interface, Graphics

(NETWORKING, continued from p. 2)
on his desk, with systems ranging from Sun 3/52s with 4 Mbytes of main memory and small 70-140 Mbyte SCSI disks for local storage, to Sun 3/260s with up to 32 Mbytes of main memory and lots of disk, which are used as file servers.

Our network also includes a number of Stellar GS1000s, very high performance graphics workstations with 16-128 Mbytes of main memory and lots of disk. All the systems are tied together over Ethernet and use NFS for file sharing.

In our setup, private user directories are scattered across both individual workstations and servers. The public directories (*/public/bin*, etc.) are all located on the servers. For every workstation to be able to access all the files, each must mount 50-80 filesystems across the network.

NFS requires that every filesystem on a server be mounted separately by the client in order to access all the server's files. In practice, however, only a subset of a server's files are mounted by every client. Just keeping track of the mount tables on 30+ workstations is an administrative nightmare.

To alleviate the administrative burden, Sun has recently introduced a system which automatically mounts remote filesystems on a client the first time they are referenced. Thus, if I want to access a file in */usr* on a workstation named *leo*, I can refer to it through the directory named */net/leo/usr*, which the system automatically mounts for me the first time through.

To prevent clogging a system with too many mounts, filesystems accessed in this way are automatically unmounted after a time if they are not in use. This scheme simplifies setting up the sharing, but here's where the problems start.

What Do You Take In Your Coffee?

Say I have a number of systems mounted for me in my */net* directory. When I list the files in */net* I get:

```
    altair astral bacall mercury vulcan
```

Each entry in the */net* directory corresponds to the root file system of a workstation in the system. If one of these machines happens to crash, say *altair*, and I have the system set to continuously retry operations on the files if the server doesn't respond, then a seemingly benign operation like a *pwd* in a directory on a machine that is still up, say */net/mercury/usr/rfg*, can hang. Why should this happen, since *mercury* is up and it's the seemingly uninvolved server *altair* that's down?

The answer can be seen in the algorithm for *pwd*, which tries to find the name of the current directory by successively backtracking up the naming hierarchy. *pwd* is trying to find the names of each directory component in the current directory pathname by starting at the current directory and looking for its entry in the parent directory. It does this by comparing the UNIX internal file ids (*dev*, which identifies the filesystem, and *ino*, which identifies the file in that filesystem) kept for each entry in the parent directory with that of the current directory. When it finds the right entry it saves the name of the entry, goes up one level, and repeats until it finds the root.

If the parent and child directories at any step are in the same filesystem, all we need do is compare *ino*s which are kept in the parent directory entry to find the name of the child. If they are in different file systems, then we need to find both the *ino* and the *dev*, which is not kept in the parent directory entry. To get the *dev*, we do a *stat* operation on the file named in the entry.

Back To The Future

This is where a problem arises, since if the entry corresponds to another remotely mounted directory, as it would for the entries in */net*, then we end up going across the net to do the *stat*. If, while looking for the entry which corresponds to our current directory path, we encounter a machine that is down, the command hangs — even though we aren't directly doing an operation on the down server.

This unfortunate side effect is just one example of how file sharing across a distributed filesystem like NFS can actually appear more like the centralized model of timesharing days. Here, even though both client and server are operating normally, the interaction of another server can cause operations to hang. Of course, there are solutions to this particular problem, but often the solutions require more complex operating system software mechanisms to get around the glitches.

A Complex Alternative

For example, one way of gettting around the dependence on individual server reliability to ensure availability of critical shared files is to replicate the files on multiple servers. Other distributed systems, like Locus, take this approach. If files are maintained on multiple servers, then if one goes down, others are still available to service requests. Such systems can automatically switch between servers to handle requests if one goes down.

The added complexity arises in trying to keep the replicated copies up to date, so that changes to a file on one server are propagated to the other servers for the file. Replication can greatly increase the robustness of a distributed filesystem and prevent occurrences like the one we saw above. But it does so at a cost in complexity and size of system software.

This increase in complexity and size can have severe implications for the size and usability of computers. Sun has recently released version 4.0 of its operating system. The new release is chock full of new features, like the automount facility, dynamic linking, etc. Its size is about 700 Kbytes, compared with about 600 Kbytes for an older 3.2 version of the OS.

We currently run Sun OS 4.0 on a number of Sun 3/52 workstations, which are limited to 4 MBytes of main memory. These workstations can barely run the X window system as a multiwindow bitmapped terminal with the real compute activities (compiles, edits, etc.) taking place on machines with more memory and more compute power. With the new, larger, more complex system software, these six thousand dollar workstations perform worse than the two thousand dollar X terminals which we have just begun to use.

Yet Another Twist

The X terminal is a clever twist. It is a M68000 based bitmapped terminal, with a mouse, ethernet interface and enough memory and power to run the X window system. It can support multiple X windows running over the network to jobs running on server machines that do the real work. The result is a low cost system with many of the advantages of workstations, multiwindow, bitmapped displays, but with little or no local compute power. X terminals bring us closer to the time sharing model, but give the flexibility of having jobs in different windows run on different servers. In other words, timesharing with a twist.

Conclusion

The moral of this tale is that as systems become more powerful and memory becomes more available, the functions expected of them and hence the software needed to run them become larger and more complex. This new world of functionality and performance comes at a cost.

We should be careful in designing, buying, and using these systems to make sure we don't take a giant step backwards to the days of the long, enforced coffee break!

Rob Gurwitz, Director Communications and Distributed Processing, Stellar Computer Inc.

OPERATING SYSTEMS

What's Real With Real Time UNIX Systems?

By Dr. Myron Zimmerman and Naren Nachiappan

Many remarkable features and traits have been attributed to the UNIX system. One is its uncanny knack for appearing in situations for which it was never intended. An extreme example of this is the system in the real time world. Standard UNIX systems emphasize multi-user time sharing and averaged performance, while real time is at the opposite end of the spectrum where deterministic response and time-critical performance are paramount.

The UNIX system first reared its head in the real time world in the mid to late seventies in instrumentation control and acquisition applications at academic laboratories; it has appeared recently with increasing regularity in the industrial automation and instrumentation markets.

Strategies and techniques for adapting UNIX systems to the real time world have advanced since those early days, resulting in a bounding and improvement of system response times, and the addition of many real time functions without fundamental change. This column will examine developments in this area, looking first at the system's notoriously poor *context switch latency*. Future columns will focus on issues such as scheduling algorithms, contiguous file systems, asynchronous I/O, memory management and locking, event handlers, networking, and POSIX 1003.4 standards. Unless noted otherwise, performance figures presented in this column are based upon UNIX System V Release 3.0 running on a 16 MHz Compaq DeskPro 80386 with a 80387 floating point co-processor and 4 MB of 80 ns memory.

Overview Of Context Switch Latency

Responsiveness of systems to the external world can be critical. For example, if in a chemical processing plant a sensor on a mixing tank indicates that the tank is full, then shutting off the pump with a 1 or 100 millisecond accuracy can represent the difference of a tablespoon or almost a gallon on a 500 gallon per minute pump. Or if data is arriving at 9600 baud over a RS-232 connection, then a system has about 1 millisecond to analyze each character of data before the next character arrives. There are endless examples of time-critical applications in the general communication, command, control, and acquisition areas.

One of the most commonly used performance metrics of a real time operating system's responsiveness is its *context switch latency* (CSL). The CSL is generally defined as the time interval between the generation of an interrupt by a hardware device and the execution of the first instruction of a specified user process.

Please note that the CSL is more than just the *interrupt latency*. The *interrupt latency* is the initial part of the CSL and is defined as the time interval between interrupt and execution of the first instruction of a kernel device driver. As an aside, the *interrupt latency* of standard UNIX systems is reasonable (minimum: 50 microseconds; average: 55 microseconds; worst case: 150 microseconds) if the port uses correct interrupt priorities and device drivers do not excessively lock out interrupts (a frequent problem during device error recovery and timing loops — the worst case of 150 microseconds quoted above was due to timing loops in the floppy disk driver). In fact, the early use of UNIX systems in real time applications was to implement the real time process as a device driver.

The CSL is made up of fixed and variable components. The fixed components are the same each time and include general interrupt handling overhead, kernel overhead in determining the appropriate process to execute, saving the current context and restoring the context of the target process including machine, floating point, and virtual memory registers.

These fixed components should not vary dramatically between different operating systems (assuming similar models of a process context) and are about 400 microseconds for standard UNIX systems. Note that some real time executives achieve much better times by using a different process model and providing only bare-bone system services, somewhat more akin to a UNIX kernel device driver.

Variable components of the CSL do vary significantly between different operating systems. Real time systems have a total CSL (adding all fixed and variable components together) which ranges between the fixed CSL and some reasonable upper limit or worst case. On the other hand, standard UNIX systems have an *unbounded* worst case CSL, often exceed 100 milliseconds, and are known to approach one second in some cases!

Non-Preemptable UNIX Kernel

Why does the UNIX system have an unbounded CSL? The simple answer is that the standard kernel is not *preemptable*. *Preemption* means to immediately stop the current activity or processing and switch to some different new activity. The early designers chose to implement a non-preemptive kernel because it significantly simplified the kernel code and design, allowing them to focus on other advanced system features and design.

The system effectively treated all kernel code as *critical code* (except for interrupt processing) and made all kernel system calls non-reentrant. This means that when the system is executing kernel code on behalf of a process, as a result of that process executing a system call, another process *cannot* begin execution until the prior process completes its system call or the system call blocks awaiting some event.

An examination of the *read* system call illustrates how a non-preemptive kernel can lead to very large latencies. When a process executes the *read* system call, the kernel goes through some initial setup code and then executes a loop as schematically shown below.

```
While       more data to be read
do
   translate logical file offset to physical disk
   block
   is disk block in cache ?
   if yes
            get disk block from cache
   if no
            queue request to device driver to read
            in disk block
                    sleep waiting for request to
                    complete
   copy data from disk block to user buffer
   adjust file offset and data count

done
```

It is clear that the only circumstance under which the process can be preempted is if the requested physical disk block is not in the cache. On a system with hundreds of cached blocks, if a process requests a *read* with a large amount of data, then the entire *read* system call with numerous loops of the above code could execute without once allowing a higher priority process to execute. The above situation was benchmarked and revealed a CSL of 85 milliseconds.

Rewrite Of The UNIX Kernel

One solution to the problem of the non-preemptable UNIX kernel is to totally rewrite it to be fully preemptable. In this case, system calls are reentrant and the kernel is fully preemptable except for short periods while executing critical code. Critical code occurs during manipulation of kernel data structures, where the modfication to the structure must be fully completed (a.k.a. atomic operation) before again using the structure. With care and good design, the critical code sections can be kept quite short. The longest such section represents the variable component to the worst case CSL.

Reimplementing a complex system while maintaining complete compatibility is always problematical.

Even if a UNIX-like system conforms to the explicit POSIX or SVID standards, implicit standards such as the order of error checking on system calls or the degree of atomic operation of system calls can create incompatibilities.

For example, if file *read/write* system calls are preemptable (non-atomic) and several processes are manipulating the same file, then the file can get into states which are not possible with atomic system calls. However, some UNIX-like implementations are using this approach. More discussion will be forthcoming in a future column.

Adding Preemption Points To The Standard UNIX Kernel

Another solution is to introduce discrete preemption points into the non-preemptable UNIX kernel. This reduces the variable component of the worst case CSL to the longest time between preemption points. The basic strategy is to add sufficient preemption points to achieve the desired worst case CSL. However, great care must be exercised not to insert preemption points into critical code sections. This can be a tricky business, because the standard system does not always localize critical code. Fortunately, in practice most preemption points can be inserted where the kernel is already prepared for a context switch.

This approach is simply illustrated by turning to the previous *read* system call example and adding a preemption point as shown highlighted in the schematic below.

```
While      more data to be read
do
   translate logical file offset to physical disk
   block
   is disk block in cache ?
   if yes
            get disk block from cache
   if no
            queue request to device driver to read
            in disk block
                    sleep waiting for request to
                    complete
   copy data from disk block to user buffer
   adjust file offset and data count
   is higher priority process runnable ?
   if yes
            context switch to higher priority
            process

done
```

The previous benchmark was also rerun with this simple preemption point added, reducing the latency from 85 to 3 milliseconds.

Special Note About Workstation Display Drivers

Device drivers are a common source of non-preemptable code which are often overlooked. Specifically, the culprits are drivers which do a lot of processing such as console displays on PCs or workstations. A memory-mapped display driver may require a lot of host processor time to perform certain display functions, such as scrolling (windowing systems on dumb bit-mapped displays present the most extreme cases). The one second latency mentioned earlier involved a process executing a *write* system call to the console using a IBM EGA display. The solution is to add preemption points into the driver, exercising care to deal with reentrancy and interrupt initiated activity (display output as a result of a keystroke).

Measured Results Using Preemption Points

A number of UNIX system vendors have used the strategy of adding preemption points into their standard kernel. Concurrent (previously MassComp) with RTU and Hewlett Packard with HP-UX have both used this strategy and bounded their worst case CSL. VenturCom's RTX/386, a modular real time extension to the standard UNIX system, and VENIX/386 implementations also use this strategy and have a benchmarked worst case CSL of 2 milliseconds (minimum: .46 ms; average: .5 ms; worst case: 2.0 ms).

The VenturCom implementation introduced only thirty preemption points into the entire UNIX kernel. The trick, of course, is where. Selecting the preemption points was based on analysis of source code and exhaustive kernel runtime profiling, looking for long threads of non-preemptable execution and inserting preemption points. These were primarily in memory management and block I/O routines. Adding even more preemption points could further reduce the worst case CSL; however, this approach is rapidly reaching the point of diminishing returns. For example, doubling the number of preemption points would not come close to halving the worst case CSL, and would start involving significant rewriting of cridical code regions.

A Hybrid Variation

The authors have researched another approach which is a hybrid of the first two and is based on the following observations. First, real time applications that demand fast response times (sub-millisecond) generally directly interface to the appropriate hardware, and do not rely on system calls or kernel services as part of the fast response because of system overhead. Each UNIX system call introduces at least .1 millisecond. Second, the standard UNIX kernel can easily be made fully preemptive if the new process does not execute a system call (or page fault, etc.).

This means that a real time process of this type could start execution immediately after an interrupt, subject only to the fixed components of the CSL, without any kernel preemption latency. However, system calls in this situation would block, and execution continue until a preemption point is reached. This hybrid approach effectively defers the kernel preemption latency from the CSL to the first system call.

This can be implemented as an event handler in a user process, similar to a normal UNIX system signal handler, where the worst case CSL to the event handler would be hundreds of microseconds. The first system call, if any, executed by this event handler would then incur additional latencies.

Conclusions

The strategy of adding preemption points into the standard UNIX kernel bounds the CSL for a previously unbounded kernel, and dramatically reduces (by a factor of 100) some of the frequently occurring latencies. This apporach has the additional advantage of introducing only minor kernel changes, allowing the rapid tracking of new kernel releases and easy implementation on different existing UNIX versions.

On the other hand, a kernel rewrite strategy should give 50% to 80% better results on a worst case CSL. However, this approach introduces a significant risk of subtle incompatibilities to application software and will require the rewriting of device drivers, representing a significant incremental cost.

In the final analysis, it all depends on the worst case CSL required by the application. The latencies achieved with current commercial versions of UNIX systems with preemption points have opened up a wide range of real time applications which were unimaginable with standard UNIX systems.

Dr. Myron Zimmerman, President, VenturCom, Inc.
Naren Nachiappan, Director of Research and Development, VenturCom, Inc.

INDUSTRY FORUM

The OSF Decision

By Dr. Ira Goldstein

Editor's note: The Open Software Foundation (OSF) announced its choice of User Interface products, referred to as the User Environment Component (UEC) of the OSF open applications environment, on December 30, 1988. The choices were made following a UEC Request for Technology issued to OSF members.

The OSF UEC will become the first OSF product offering. As such, it will have a tremendous impact on User Interface standards for the foreseeable future. The product will be a source code package that will run on any System V.3 compatible operating system.

The following material is adapted from the OSF User Environment Component Rationale Document issued January 11, 1989. It includes a brief description of the technology model used by OSF to make its decision, plus a detailed description of the actual UEC offering.

Reprinted with the permission of OSF.

Technology Framework

The OSF user interface technology framework (see Figure 1) is a conceptual model based on work from the National Institute of Standards and Technology (NIST) and X/Open. The framework assumes a client-server model as is used in the X Window System™, Version 11, where the server is a workstation or specialized terminal with a screen, keyboard, and pointing device, and the client is an application program.

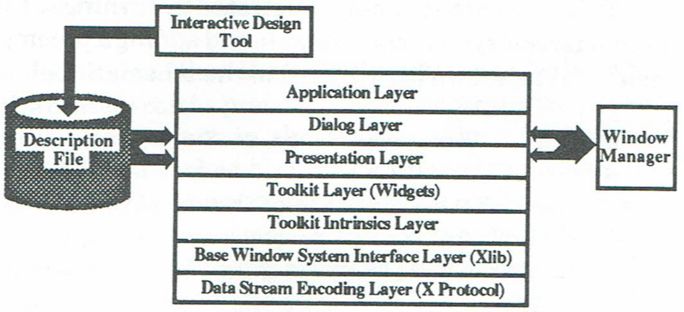


Figure 1: Technology Framework

Layers 0 through 5 of this layered model typically are implemented as library routines linked with application code, represent by layer 6.

In addition to the layers, the framework describes interactive design tools and window managers. Interactive design tools are separate applications, pictured as producing output, which drive the presentation and dialogue layers. Window managers are special applications that communicate with ordinary applications through their presentation and dialogue layers. Briefly, the layers are:

<u>Data Stream Encoding Layer (0)</u>: X11 protocol: graphics primitives, manipulation of nested and overlapped windows, specialized inter-client communica-

tion.

Base Window System Interface Layer (1): Xlib, a C interface built on top of the X11 protocol.

Toolkit Intrinsics Layer (2): Xt Intrinsics, built on top of Xlib, for construction and use of user interface objects.

Toolkit Layer (3): the OSF toolkit, built on the Xt Intrinsics, containing a set of ready-made user interface objects.

Presentation Layer (4): support for initializing or altering the presentation aspects of an interface.

Dialogue Layer (5): interaction mediator between application and user.

Application Layer (6): application code.

Interactive Design Tools: separate programs enabling the user to define an application's interface.

Window Managers: separate application implementing window management policy.

UEC Scope

The submissions to the UEC Request for Technology included X extensions, graphical toolkits, text management toolkits, user interface toolkits, presentation description languages, dialog description languages, interactive design tools, window managers, graphical desktops, and graphical shells. The OSF model of technology (see Figure 2) groups them into three concentric components, as shown below.

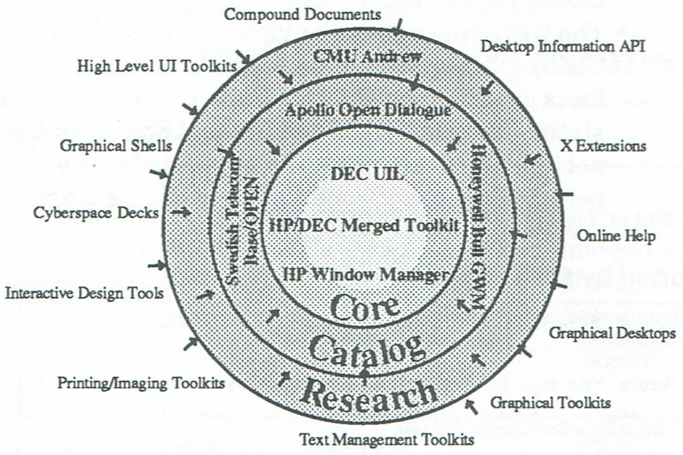


Figure 2: OSF Technology Model

At the center of this model is the core offering, containing basic technologies which OSF makes available and encourages the industry to license. The UEC core offering consists of:

- A Style Guide
- A Window Manager
- A Toolkit
- A Presentation Description Language
- Associated Documentation

The second ring of Figure 2 labelled "catalog" consists of products that are consistent with the OSF core but are supported by their suppliers. The third category consists of research prototypes and tools that are made available for industry review.

The following sections describe the components of the offering, and are excerpted directly from the Rationale Document.

Style Guide

A style guide is a multi-faceted document. At one level it is a specification of the constraints and flexibility of the window manager and toolkit behavior. The style guide specifies the appearance and behavior of these technologies from the user's perspective. On another level, a style guide is a guide to usage, a cookbook for application writers where the ingredients are the toolkit widgets.

Since OSF has selected the HP window manager, with additional features from the Digital window manager, and a merged HP and Digital toolkit, it follows that the style guide also should be a merged document based on the selected technologies. Thus the OSF style guide will be based on the corresponding documents from Digital Equipment Corporation and Hewlett-Packard/Microsoft. It will describe the PM compatible behavior inherent in the OSF window manager and toolkit, as well as the extensions to this behavior and appearance.

Window Manager

OSF has selected Hewlett-Packard's submission as the basis of the OSF window manager. In addition, the window manager will also include selected features from the Digital window manager such as the Icon Box.

The window manager supports Presentation Manager behavior and layout, and provides the 3D, or bevelled, appearance which has been chosen as the OSF reference appearance. The appearance is supported on, and takes advantage of, both black and white and color workstations. This appearance, however, is provided only as a reference. Vendors are free to modify or enhance it as they wish. The window manager is compliant with the Inter Client Communication Conventions Manual (ICCCM) from the X Consortium, and its implementation is based on the toolkit.

The window manager supports a great deal of customization in both appearance and behavior, including support for both real-estate-driven and click-to-type focus models. In addition, the colors and styles associated with the 3D appearance can be varied widely in order to adapt to different screen sizes, resolutions, and color capabilities.

The Digital Icon Box holds icons for each of the windows managed by the window manager, collecting them together in a single location, and allowing them to be moved as a group. The size of the icon box can be limited. If there are more icons than fit in the icon box, scrollbars are automatically provided. Icons are displayed even for windows which are mapped; double clicking on such an icon raises the corresponding window above other windows on the screen, providing a particularly convenient means of regaining access to obscured windows.

Toolkit

OSF has selected a merged toolkit, based on a combination of the HP/Microsoft and Digital submissions. A limited number of widgets, such as the Digital Help widget, have been excluded. In addition, modifications have been made to make the names of the final widget set internally consistent. The table below (see Figure 3) contains a list of the objects supported by the OSF toolkit.

The application program interface provided by the toolkit is that supplied by the X Intrinsics, as extended by the Digital toolkit. Additions are being made where necessary in order to support the HP toolkit functionality; however, these changes should not interfere with the application program interface (API) available to members in the initial snapshot.

The behavior of the toolkit conforms to Microsoft's Presentation Manager, as specified by the HP/Microsoft submission, with additional extensions to take advantage of the power of networked, X-based workstations. This compatibility should provide users with a simple learning transition between PC and workstation environments, as well as simplifying the documentation needs of products available on both types of systems.

Presentation Description Language

OSF has selected Digital Equipment Corporation's UIL (User Interface Language) as the basis of a presentation description language for describing the presentation aspects of user interfaces. This language can be used directly by interface designers who need not have extensive programming expertise.

Interfaces described in this language are compiled to a binary format. This binary format is *not* linked with an application program, but is read by the program at runtime. This means that changing the presentation aspects of an interface does not require that the application program be recompiled or relinked, leading to very fast turnaround in tuning an interface design. Both the description language compiler and the runtime interface manager will be based on Digital's UIL technology.

UEC Documentation

Documentation quality is critical to the success of the UEC offering. The documentation effort will emphasize easy access to accurate and useful information, by providing a cohesive overall organization, effective graphics, useful examples, and navigational tools such as high-quality indexes.

The initial documentation set will include the following types of manuals:

- A Style Guide covering compatibility with Presentation Manager behavior, and conventions for toolkit component usage.
- The UEC Applications Environment Specification (AES). This specification will define all the interfaces classified by OSF for Full or Trial-Use AES status. Interfaces defined in the AES are stable, not implementation specific, supported over a long period of time, and modified only after a lengthy review process.

Figure 3: Objects Supported by OSF Toolkit

Basic Widgets	
Drawing Area	graphics workspace
Separator	lines for separating areas
Label	static text and graphic area
Scale	slider for getting numeric values
Scrollbar	scroll control
PushButton	task activator
DrawnPushButton	pushbutton with user-drawn graphic
ArrowPushButton	pushbutton with drawn arrow graphic
ToggleButton	button with state
CascadeButton	button for cascading menu items
OptionField	field allowing an enumerated set of values
Text	text entry and editing
Command	input pad with transcript
Composite Widgets	
Board	bulletin board for arbitrary placement of objects
Form	object layout using alignment constraints
List	list of strings
SashedColumn	column with resizable subareas
RowColumn	object layout with row and column constraints
MenuBar	menu area for pulldowns
Frame	container providing 3D framing
Scrolled Widgets	
ScrolledText	scrolled text area
ScrolledList	scrolled list area
ScrolledWindow	generic scrolled area
Specialized Composite Widgets	
RadioBox	collection of ToggleButtons
SelectionBox	widget for selecting one among a list of strings
FileSelectionBox	special SelectionBox to deal with selecting files
Basic Top-level Widgets	
MainWindow	top level application window
BoardDialog	Board used as a transient dialogue box
FormDialog	Form used as a transient dialogue box
Menu	popup or pulldown menu
Specialized Dialogue Widgets	
MessageDialog	dialogue box to display a message
CautionDialog	dialogue box to display a caution
WorkDialog	dialogue box to display a work in progress notice

- Reference materials describing all programming interfaces. These will be available as on-line man pages as well as hard - copy manual format. The reference materials will document all interfaces defined in the specification, as well as any that are part of the offering but not included in the specification.
- A task - oriented programmer's guide providing how-to information about the offering.

Documentation will be part of every "snapshot" released during the development process, both in source and on-line output format.

In the coming months, OSF will be announcing Requests for Technology that extend this core offering to other areas of the user interface.

The OSF Research Institute will be working with universities to explore innovative extensions of the OSF core technology.

Dr. Ira Goldstein, Vice President, Research and Advanced Development, OSF

X Window System is a trademark of MIT

SECURITY

DAC – The Immediate Future In Trusted Systems

By Steve Sutton

The UNIX system has come a long way from the academic curiosity of the early seventies. It has emerged as one of the few national — more properly, world-wide — operating systems, positioned squarely at the center of the consumer-driven movement toward "open systems." It has also become a critical element of systems that must be "secure" in some sense of the word — whether for national defense or for high-volume electronic funds transfer.

The UNIX industry has done well in keeping abreast of "soft" security with many small fixes to common UNIX foibles. However, major feature extensions for highly-secure systems lie ahead. Driven by monstrous federal procurements, vendors can no longer avoid "trusted" systems that will provide significant new forms of security for end-users. Careful users who genuinely want more security need to know what to expect, what to look for, and how to plan for it.

Control over which users can access which data, and how they can access it, is by any measure a fundamental component of "security." One of the key features of trusted, "secure" systems, and also a feature of all standard UNIX systems, is the topic of this column: *discretionary access control* (DAC).

While certainly useful, DAC is not sufficient for environments where breaches in access control have severe consequences and where malicious users are likely. Defense-related activities have always held these environments; other federal and commercial environments are increasingly faced with this situation. As a result of this demand, vendors will strengthen DAC, and other access controls will be added to UNIX systems. Many access control additions will respect backward compatibility, and will likely be optional to the end-user.

The *Trusted Computer Systems Evaluation Criteria* (TCSEC, commonly called the "Orange Book" or, by insiders, simply "the Criteria") is the major driving force through which trusted features are being added to UNIX systems. It is the guide by which the *National Computer Security Center* (NCSC) evaluates and rates commercially available trusted systems. An NCSC rating is increasingly required by large federal procurements.

The TCSEC was formulated in the early eighties from Department of Defense (DoD) policy and practice. It contains basic security technology applied in a rational manner; its applicability extends far beyond the DoD. The current IEEE POSIX effort for trusted system standards, P1003.6, is using the TCSEC as its principal guideline. Not surprisingly, access control is a major topic of TCSEC.

This article will focus on what is likely to be implemented in the trusted UNIX system standards, like POSIX. Access controls are a major part of these trusted system features, and DAC is the one of immediate interest.

There are two other important forms of access control: *mandatory* and *integrity*. All three are complementary and all will eventually appear in highly trusted systems. *Mandatory access control* (MAC) is based on the control of all access to files by the system administrator. Integrity controls are mandatory access controls on alteration. Both MAC and integrity will be dealt with in future articles.

In general, each separate kind of access control protects data regardless of the others. All must allow access in a given mode — any dissenting vote will preclude access. Both mandatory and integrity controls cure some of the implicit problems with DAC, and at least mandatory controls will be required in most highly secure environments.

Terminology

We will use the term "access mode" for a potential manner in which access is allowed (read, write, etc.). The term "object" will refer to a container of data to which access is controlled. Fortunately, most UNIX system objects are accessible through the file system. For simplicity, the reader can generally equate "object" to "file." The term "administrator" will refer to a user with special security or operational responsibilities.

The term "trusted computing base" (TCB) is taken from the TCSEC and refers to all portions of the system (hardware and software) that must be trusted to uphold (that is, are capable of compromising) the security rules of the system. A key element of trusted operating systems is that untrusted components are freely allowed. They are precluded from violating the system security rules by the TCB.

One final note. There are a lot of acronyms in security, and you might as well get used to them. DAC, MAC, TCSEC, and TCB are the most common ones.

Discretionary Access Control

DAC is a scheme in which the owner of an object can determine which users can access the data and the modes in which that access can take place. Hence, access control is at the "discretion" of the owner. Inherent in this scheme, and in UNIX systems, is the fact that each object has an owning user, and perhaps group. There is really no intent in DAC that system administrators restrict the degree to which a user can expose owned data, although some DAC schemes have such features.

UNIX Permission Bits

Currently, UNIX systems implement a form of DAC commonly called the "permission-bit" scheme. Objects are given a set of nine *permission-bits*. Each group of three *permission bits* denotes the ability to read, write, and execute, respectively. The three groups apply to:

- The user that owns the object (which we will simply call the "owner");
- The group associated with the object;
- And all other users.

Most users see permission bits as the familiar "rwxr-x--" strings used in the *chmod* and *ls* commands. The *read*("r") and *write* ("w") modes have the usual meanings. The *execute* ("x") mode means something different for files than for directories. For files, it applies only to executable (program) files and means that the user can run the program. For directories, it means that the user may "enter" the directory in pathname searches; this "search access" does not by itself allow reading or writing of files within the directory. This is illustrated by the following figure:

owner	group	others
r w x	r w -	r - -
1 2 3	4 5 6	7 8 9

In this figure, the owner *(jjones)* is given read, write, and execute permission, the group *(tellers)* read and write, and all other users read only. These would be entered as "rwxrw-r- -" in the *chmod* command.

The ACL

Permission bits are simple and usually sufficient. Their main shortcoming is that they do not allow a more detailed specification for access.

Suppose for the preceding example you wanted to give just one more user, *bsmith*, who is not a member of *tellers*, read access to the file. The only way is to ask the system administrator to include *bsmith* in the group *tellers*; this may inappropriately allow *bsmith* access to all files restricted to group *tellers*.

The *access control list* (ACL) is the DAC scheme that is the most straightforward extension of permission-bits. It associates an ACL with the object. The ACL is a series of entries, where each entry designates access to a particular user, a particular group, or to all users:

user:	*jjones*	rwx
group:	*tellers*	r w -
others:		r - -

This example has one user entry, *jjones*, one group entry, *tellers*, and the entry for "others." It allows the same access as the permission-bit example above.

user:	*jjones*	r w -
user:	*bsmith*	r w -
user:	*oddman*	- - -
group:	*tellers*	r - -

Here, users *jjones* and *bsmith* are given read and write access. Group *tellers* is given read only access. However, note that *oddman* is given no access. Since *oddman* is listed before the group, *oddman* is given no access even if *oddman* is a member of *tellers*. In effect, all members of *tellers* are given read access except *oddman*, who has no access. This example begins to illustrate the power of ACLs.

There are many variants on this scheme. One popular extension is the ability to specify access to a particular user within a particular group.

Restrictions on the order of the ACL entries vary between schemes. Some allow the user to order the entries arbitrarily. These are flexible but can lead to confusion. Many require the most specific entries to come first, i.e., specific user entries, followed by specific group entries, followed by the "other" entry, if present. Most schemes search the entries from the top and stop when they first find a match, as in our ACL example above.

A number of other DAC schemes have been proposed in the security literature. However, since ACLs are a direct extension of standard UNIX system permission bits, and wording in the TCSEC is directly targeted toward ACLs (although other forms are not precluded), it is unlikely that any other scheme will become the UNIX standard. A few UNIX system vendors provide ACLs and no significant UNIX versions provide other forms of DAC.

Default DAC

The default DAC is the information placed on a newly-created object. This is quite important since few users want to manually set the DAC time after time. UNIX systems set the permission bits based on a property of the program called the *umask*, which limits the bits that are given as default. A user typically sets the *umask* on login and all files created will be given the corresponding defaults.

For ACLs, defaults are a little more complicated. There are two basic opinions. One is that the default should be based primarily on the directory in which the file is created. The other is that it should be based on a property (a default ACL) of the program. Which is correct and which will prevail only time will tell. However, default ACLs are a difficult issue of which both vendors and end users need to be aware.

The Problem with ACLs-Backward Compatibility

ACLs are the popular choice for extending UNIX system DAC. If they could simply replace the permission bits, then a simple, "clean" form of ACL would suffice. However, the vendors that participate in the standards forums, such as POSIX, demand backward compatibility. In particular, current programs and shell-scripts that contain permission bit operations should function unchanged when applied to objects that have ACLs. Moreover, the security intent of these programs should carry over.

Unfortunately, the kind of protection intended by the permission-bit operations cannot be unambiguously applied to ACLs, particularly large or complex ACLs. So the strategy of the standards bodies has been to define an ACL scheme that is a compromise between simplicity and backward compatibility.

One proposal is for both permission bits and an optional ACL to exist, and for the permission bits to interact with the ACL. The group permission bits would serve as a "mask" (limitation) on the ACL. If the group permission bits were r-- (read only), then the ACL could at most give read access – and no others.

The advantage (which is probably far from obvious) is that a user or shell-script that blindly applied "***r--***" to the ACL-protected object would do approximately the right thing.

Further details of these proposals are beyond this article. The main point is that the standard ACL scheme will be more complicated due to commitment to backward compatibility.

The Prognosis for Extended DAC in UNIX Systems

What can you, the UNIX developer, expect to have to support for extended DAC in trusted UNIX systems? First, we should note that ACL-like extensions to the permission-bits are a commonly requested feature for UNIX systems, even from users that would not consider themselves as having demanding security requirements. Second, along with security auditing, ACLs will be one of the first security enhancements to appear.

The good news is that applications that rely on permission-bits will continue to function on objects with the new ACLs. The bad news is that the ACL schemes will be more complex than if compatibility could be ignored.

Conformance to the standards is virtually required. Vendors will likely consider two additional kinds of features:

- Since the standard ACL format may be less than simple, user-friendly interfaces will become highly desirable. Many casual UNIX system users do not understand permission bits — ACLs will only be harder.

- More features to add access granularity to the standard ACL scheme are possible. For example, modes other than "rwx" could be added, such as delete or the ability to define the DAC for an object.

While the former are attractive, sellable features, the latter are specialized and should probably be avoided by the general vendor.

As one last issue, does this mean that all files need ACLs? If so, then pre-existing disk pack formats would not be allowed, since they would have only permission-bits. Fortunately, even the TCSEC will allow access to pre-existing permission-bit only disk packs. The ACL will be a feature that would normally be provided for "new" packs.

The Problem with DAC

The problem with DAC is that any program running on behalf of the user is allowed (by the operating system kernel through system calls) to change the DAC of any object owned by that user. The kernel, which can be assumed "trusted," has no way of knowing if the user actually intended the change to be made, or whether the program itself made the decision unknown to the user.

Programs with malicious code can therefore "give away" data unknown to owners who would normally want to control access. Such malicious programs are called "Trojan Horses" because the malicious intent is masked by an overall benign appearance.

No one wants such programs on their system. However, for the foreseeable future one cannot avoid them in practice. Most UNIX sites need many applications from diverse sources, including internally developed programs. To limit one's selection to the few that have gone through a rigorous testing program is rarely practical.

Does this mean that DAC is useless? No. It simply means that DAC is inherently susceptible to a common penetration scheme. This is precisely why the other forms of access control (MAC and integrity) are important; they are not susceptible to this problem.

As an illustration, a mandatory access control is one wherein the owner of an object is not allowed to arbitrarily specify which other users can access the object. Instead, access is based on overall controls established and maintained only by system administrators. Since the user can never relax a mandatory control (i.e., allow more access than would be allowed by the administrative control scheme), neither can Trojan Horse progams. Hence, the principal security susceptibility in DAC is not present in MAC.

For Implementers Only

Discretionary access controls are basically a feature-level extension. In the main, they do not require any fundamental reorganization of the underlying system, and can be surprisingly easy to implement. In general it is more difficult to define the features than to implement them. Given that at least one popular standards group (POSIX) is defining interfaces for these features, the easier part is left to implementers.

Backward compatibility can be reasonably maintained, and this is certainly the commitment by the standards groups. The different access control schemes (discretionary, mandatory, etc.) are essentially independent and can thus be bundled in any combination.

The major implementation issue is that the file system format will have to be augmented to hold additional access control information. This puts yet one more burden on the file system designer and can be a moderately large effort. The second issue — probably the harder of the two — is to be able to present end users with genuinely useful and friendly interfaces to the mechanisms.

For Programmers Only

What does the programmer need to know about enhanced forms of DAC?

The biggest problem is programs in existence today that manipulate the permission bits. When they deal with objects that have ACLs (or other extended DAC information) they may not quite do the security function that was intended. Some tips on how to avoid this:

- Write programs that do access control from a high level. Use a standard of routines like "block all access," "block access to all but owner," or "grant public read." As DAC extensions emerge, only these routines will need to change — not the programs that call them.

- Scan old shell-scripts for *chmod* statements. These will be a source of problems.

- Don't use *stat* and *chmod* to gain temporary exclusive use of files. Use a real locking mechanism if possible.

- Give some thought to the places where you really need more detailed access controls – where ACLs will be welcome – and where you don't.

For Buyers Only

When you buy a trusted system, ask the following questions about DAC extensions:

- Can you either deny or grant access to specific users and also to specific groups? You should be able to.

- How well do permission-bit commands (like *hmod*) work on files with extended DAC?

- Has the vendor made it easy to use the extensions?

Most ACL schemes will be a compromise between backward compatibility and cleanliness, and it is possible to provide sets of commands that work with "virtual images" of the ACLs that make them seem cleaner.

- How do default mechanisms work? Are defaults based on the directory in which files are created, or on attributes of the creating program (like the UNIX *umask*)?

When you start to get systems with extended DAC, think about the following:

- Standing between the devil of permission bit and the deep blue sea of ACLs may be difficult. Decide where you need more detailed access control.

- Consider a set of personal or site-specific shell scripts for common operations, like those above (e.g., "block all access"). With any luck, your vendor will give you these — eventually.

- Consider living only in the new world of ACLs by using only ACL commands, even if they only pro vide protection synonymous with permission bits. Permission bits are not terribly clear to today's casual users. ACLs may turn out to be clearer.

Summary

No single access control mechanism amenable to standardization in UNIX systems is sufficient for highly secure environments. Instead, a combination of discretionary controls, which UNIX systems have, and mandatory controls, which are not available in UNIX systems, will be required.

Today's standard UNIX systems have a usable form of DAC, the permission-bits, that will likely be augmented with ACLs. The form of the ACL will be a compromise between simplicity and the preservation of backward compatibility. While there will be some opportunity for vendors to add DAC features beyond the standard, they are small.

ACLs are regarded as a useful feature in almost all environments and represent the most immediate access control expansion for UNIX systems. However, the demand from the federal sector (specifically, systems evaluated according to the TCSEC) will cause MAC features to appear soon. As evidence, the current POSIX effort (1003.6, due in late 1990) is targeting both ACLs and MAC for the first version of trusted features extensions.

The Last Word

Perhaps the only certainty is that:

• Vendors who do not choose to offer TCSEC security will find themselves with an ever-dwindling segment of the UNIX system market;

• End-users that are not at least a little prepared for the systems will be missing a great opportunity or will be subject to more than a little trusted system culture shock.

Steve Sutton, Trusted Systems Consultant

INTERNATIONALIZATION

What's Required of a Truly International UNIX System?

By Greger Leijonhufvud

Five years ago, a UNIX environment adapted to the needs of non-English-speaking users was not a high priority for anyone.

Of course, some developers and software houses had already seen a market for their products in Europe and elsewhere. The usual solution was a new version of the software, adapted to the local requirements, using the appropriate language or other local information for each country.

Multi-national corporations faced the same problem, using one of two approaches: either localize applications, in effect creating different systems (sometimes with conversion facilities), or decide that "business is conducted in English."

The result: most local subsidiaries had their own systems, incompatible with sister subsidiaries in other countries, causing substantial support and coordination problems. An invoicing system installed in the early eighties in ten different countries usually required the development of ten different versions.

This common practice is called *localization* because it results in flavors of each product which address only the *local* environment and requirements. *Internationalization* is a new and different approach, one which aims to provide system-wide tools that allow developers to minimize the task of providing a separate local environment, with fewer development, testing, and support issues to deal with when taking advantage of foreign markets.

Today there is much more interest in development of a truly international version of UNIX systems. Several manufacturers have released products, others are still in the design and development phases. Fewer and fewer question the whole concept. Standards bodies like the ISO committees, the ANSI X3J11 (C language) and the IEEE POSIX groups have incorporated internationalization features in proposed and accepted standards.

With all this activity going on, it's time to take stock of where it might be leading us.

The Necessary Components For True Internationalization

Most of us take for granted the fact that when we communicate with a computer — through terminals, automatic bank tellers, flight reservation systems, rental car check-ins (and maybe even the car) — we can do so in English. We also take for granted additional support for the underlying concepts, such as sorting order, date formats, the "decimal point" and more.

Non-English-speaking users face a much less convenient world. The output from an *ls* is not listed alphabetically. The date is all wrong. Many can't even use their own alphabets, which may require more symbols than the 7-bit ASCII code set provides for the 52 letters, 10 digits, and 32 punctuation symbols commonly used in the U.S. environment. What an American reads as a special symbol is a letter to the French, and may be a different letter to Germans.

One obvious solution is to extend the code set to a 256-character 8-bit code. The drawback is that, even though UNIX systems have always worked with 8-bit bytes, the eighth bit is frequently used by applications as a "flag".

Some of the problems in internationalization are trivial. But others are rooted in the fundamentals of the UNIX system. In order to fulfill the main goal of internationalizaton — to provide the international program developer and user with an efficient, easy, and familiar environment — an international UNIX system must provide the necessary tools to support different local languages and local customs with the same version of language-independent software.

And it must be possible to do so independently of program code.

What Will It Take?

There are five main components of international UNIX systems:

1. Character Sets and Code Sets
2. Compilers
3. Commands
4. Libraries
5. The Announcement Mechanism

All five elements must work together to support a local environment.

Character Sets and Code Sets

Good programs are data driven. But good international programs cannot afford to be code set driven, as the traditional UNIX system is.

Since our goal is to be able to develop programs that work in many languages, the current character set must be expanded to support many additional symbols.

A *character set* is the sum of all the symbols (also known as "glyphs") we use to express ourselves in writing. The familiar character set is sufficient for the English-speaker, but other languages require more symbols — for instance, accented letters in French, or the tilde in Spanish.

There are two types of alphabetic character sets: Latin-derived and non-Latin derived. The Latin-derived sets are typically small (26-50 letters), use only two character forms (upper and lower case) and are seldom syntax-sensitive. Many non-Latin sets, such as Arabic, have four or more forms of each "letter"; the form is syntax sensitive, and may be written right-to-left. In addition, Latin symbols such as digits are often mixed in.

Non-alphabetic character sets, such as the Japanese Kanji and the Chinese Hanzi, require several thousand separate symbols.

Character sets must be distinguished from code sets. A *code set* is a representation of a character set. Many different code sets have been used to represent the Anglo-Saxon character set: BCD, XS3, Fieldata, EBCDIC, and ASCII, to name just a few.

In the early years, non-English character sets were represented in the U.S.-derived code sets by replacing less frequent special symbols by the local characters. There are several "national" variants of the ISO 646 standard: the "|" (vertical bar) symbol in the U.S. variant (also known as "ASCII") is used in Spain for the ~ symbol and in Germany for the : symbol. Modern code sets tend to leave the "basic" character set alone and add symbols.

Japanese and Chinese code sets must cover thousands of characters. The Japanese JIS 6226 code set defines over 7,000 characters. An 8-bit byte is not enough for such needs; the JIS standard uses 16 bits per symbol.

As a user, I'm concerned that "my" character set is supported. That may not be the same character set as that required by another user on the same system; a Swiss UNIX system must support at least four different languages.

Developers are concerned about how these character sets are represented, about "code sets." Unless I can use a "universal" code set that covers all character sets, I may have to mix code sets. The popular ISO 8859 code set is not one, but more than nine code sets. 8859.1 covers Western Europe, 8859.2 Eastern Europe, and so on. Some encodings are incompatible — ISO 8859 cannot easily coexist with IBM Extended ASCII.

So international UNIX systems and applications have to be capable of functioning with many different character and code sets at the same time.

Compilers

Most program development requires compilers. The degree of internationalization required here is still a matter of lively debate, but agreement is fairly broad that at least the following are required:

1. Constants in the user's language
2. Comments in the programmer's language
3. A data type capable of representing characters regardless of how many bytes are required; for example, a Japanese "character" does not fit in a byte and the C *char* is defined as a byte wide

In addition, some advocate that programmers should be able to define identifiers and even keywords in their own language.

Commands

Any UNIX command or utility that *interprets* or *processes* data (display, editing, sorting, formatting) must be modified to remove code set dependencies, including:

1. Use of the eighth bit as a flag
2. Sign extension on size conversions
3. Array dimensioning based on code set size
4. String truncation, since multi-byte symbols cannot be truncated in the middle

They must also be modified to adapt to the local conventions:

1. The collation (sorting) order
2. The local date and time formats
3. The decimal delimiter

And they should also communicate with the user in the user's language.

Libraries

Library functions are the basic tool of the program development system. The same changes required for commands are also required for the system-wide functions used in almost all programs. Some changes such as case conversion may require a new design when the process is more than a simple reciprocal substitution.

New library functions may also be required. The X/Open Portability Guide (XPG) specifies a message catalog facility, enabling the developer to separate messages from code, thus making local *language* adaptation much easier.

Announcement

With the capacity to support multiple character sets (and languages), the software now requires a mechanism that can define, modify and report on the environment, including:

1. Names of months and weekdays
2. Local format of date and time
3. Decimal delimiter (radix) and currency symbols
4. Collation order
5. Yes/No representation
6. Message language

Conceptually, this is similar to the terminal set-up mechanism. It is controlled by the end user and transparent to most programs.

When character and code sets, compilers, commands, libraries, and the announcement mechanism are all in order, we have an international UNIX product. It is not yet *adapted* for the local environment, but all the tools are there. It should now be a much easier task to develop an application that can be self-adapting and that often will not have to know the user's language or preferred decimal delimiter.

To support worldwide applications portability, some aspects of the five elements must be common between implementations.

Standards Efforts

ANSI X3J11 is defining the syntax and behavior of the C language (compiler and library routines). The current proposal contains several internationalization features, broken into three groups:

1. *Modified C functions.*
Character handling (*ctype*) functions are extended to handle any 8-bit character set, and printing and editing (*printf/scanf*) functions support decimal delimiters other than periods.
2. *New C functions*
Selection of a specific "environment" (e.g., radix character or alphabet definition) is handled through *setlocale*, an alternate routine for date/time fetching is defined, and non-ASCII collation is supported through two new string manipulation routines.
3. *Multi-byte character sets*
The current draft contains support for "wide" (multi-byte) characters.

The POSIX standard 1003.1 defines operating system interfaces and behavior, how the C language internationalization features can be expressed in a POSIX environment, and also some things (such as time zone support) that are outside the scope of the C language.

The P1003.2 committee is working on commands and utilities, and has been working closely with the */usr/group* subcommittee to add such things as collation and regular expression handling for international needs.

Although not a standards group, the X/Open Consortium exerts a strong influence through the *X/Open Portability Guide* (XPG). Adherence to the XPG is a "marketing must" in Europe.

The */usr/group* Internationalization Subcommittee has been working on the topic for over two years. Recently the amount of interest in and commitment to their effort has grown considerably. The group meets more than 4 times per year and has wide representation from major manufacturers as well as software houses. In addition to providing input and comments to X3J11 on the internationalization extensions, the Subcommittee is currently working on messages and code sets.

What's Available Now?

Almost all major computer manufacturers have long since offered *localized* versions of their proprietary systems. Foremost of the *localized* systems are those offered in Japan; both Japanese and American manufacturers have offered Japanese language and character set support for many years.

Some manufacturers also have *internationalized* offerings. In the UNIX arena, the leading maufacturers in internationalization efforts, with available products, are:

> **AT&T**: System V.3.1 provides the first internationalization features from AT&T. In addition to full support for eight-bit code sets it also supports user-specified date/time formats and character classification. The System V.3.1 implementation does not follow POSIX or X/Open.
>
> Release V.4 promises to provide POSIX compatibility, and extends the internationalization to collation, multiple co-existing code sets, and messages.
>
> In Japan, AT&T also offers a Japanese UNIX supplement called Japanese Applications Environment (JAE). In Europe, AT&T UNIX/Europe Ltd. supports a French and a German Applications Environment which is X/Open compatible.
>
> **Bull**: Offers support for Western character sets and is X/OPEN compatible.
>
> **Hewlett-Packard**: Provides fully X/OPEN compliant systems, both for the Western (8-bit) and Asian (16-bit) language areas.
>
> **IBM**: AIX provides full support for internationalization both in Europe and in Asia. Only the message system is currently X/Open compliant, but IBM is committed to both POSIX and X/Open for future releases.
>
> **Siemens**: Supports Western character sets and is X/OPEN compatible.

Increasingly, applications are being developed using these interfaces.

Conclusion

In the past, internationalization of the UNIX system was often the result of immediate marketing pressure and was undertaken without much consideration of long-term effects. Existing implementations provided incompatible interfaces. Porting an application often required a re-write.

With the advent of X/Open, and the growing realization in the standards groups that internationalization must be supported, the picture changed. To be viable, internationalization must be portable. By cooperating and agreeing on a set of common interfaces and facilities, the UNIX industry is casting a solid foundation for a truly international software industry.

That means that the software packages of tomorrow will run equally well in Beijing, Belgrad, and Boston, on UNIX systems from IBM, ICL, or INTERACTIVE.

How will this affect the average UNIX installation? In future columns, we'll try to answer some of those questions.

Greger Leijonhufvud, Principal Member of Technical Staff, INTERACTIVE Systems Corporation

STANDARDS

Programming For The POSIX Era: A Case History

By Shane P. McCarron

In September of 1988, the IEEE Standards Board approved the first major programming standard for the UNIX Operating System — POSIX. This base level standard promises to change the way in which people manufacture, purchase, and use UNIX-based machines. It, along with its sister standards now being specified, describes a portable interface for application development that could make it much easier to write software for UNIX systems.

Unfortunately, there is a hitch. For each software developer with an existing application there is an existing customer base whose systems may never conform to these new standards. And even though there are few conformant systems around today, as they become available and potential customers buy them, developers will need to support this new, standard environment in addition to all the environments they have supported in the past.

This gives rise to a question: Is there a way to use these standards environments, universally, today? In this article, I will present some methods that may assist software developers using the new interfaces that are being standardized in this, the POSIX Era.

Instead of focusing on software engineering issues, I will address ways in which you can program for portability — writing software that will execute in many environments under a variety of configurations.

The Problem

So, what makes it a problem to write software that will function on a wide variety of UNIX-based machines? The obvious answer is that there are many different vendors, and each of them has their own flavor of UNIX system.

This answer is correct as far as it goes. To elaborate, many of the machines have different format tapes, diskettes, serial interfaces, devices names, compiler features, etc. Then there are the more basic software oriented problems, like different library interfaces, different system calls, and of course different bugs.

Those are vendor created problems. End users create their own set of headaches by each having a different favorite terminal, none of which emulate very well whatever it is they are trying to emulate. Portable software needs to address all these issues, and still attempt to do whatever was supposed to get done in the first place.

On the software developer side you have an additional group of problems. First on the list is that no one is ever given the time to do it right. It is almost impossible to explain to a vice president somewhere why you need twenty man months to develop a set of software engineering standards and portable programming guides. It is well and good to say that this herculean effort will make it easier to develop software when it's done, but in the interim none of the customers are getting product.

A Solution?

For the last few years I have looked to the standards world to solve my problems for me. Having standards interfaces for all aspects of the system would naturally make my job easier. Conscious portability (placing explicit references in code for specific systems) would no longer be an issue.

Unfortunately, the UNIX interface standards defined thus far are base level standards. They provide for the simplest functionality of a system — read, write, and the like. Any moderately complex application needs more than is defined by any of the standards around today.

Real world applications need facilities like windowing interfaces, databases, and networking. There are no formal standards with defined UNIX interfaces for any of these, yet.

Further, the standards that are under development are so immature that their final form cannot be predicted. Put simply, there are two fundamental problems facing the software developer who wants to use standard interfaces: implementations don't exist for defined standards, and standards don't exist for the interfaces that are really needed.

An Approach

All is not lost! The IEEE Std 1003.1-1989 (POSIX) is well defined, and ANSI X3.159 (ANSI C) is about to be finalized. Sure, these are only base level standards, but at least they will (probably) be on every major implementation in the not-too-distant future. They will provide a good foundation for portable application development.

Recently, a group of us decided to try to implement the POSIX and ANSI C interfaces on top of existing implementations. While this has proven to be very challenging, it is not totally impossible. The interfaces described in POSIX and ANSI C can be built, and can provide most of the functionality described in those documents — <u>without access to system source code</u>.

This was the key to our implementation, and guided every tough decision and compromise we had to make. If anything required system source modification, it wasn't acceptable. Most application developers do not have access to the system sources — source code licenses tend to be very expensive, restrictive, and hard to come by. This was not the path we wanted to take.

Operating under this restriction meant that it was impossible to provide all the functionality of these standards on all existing implementations. Some aspects of POSIX are going to require massive kernel changes on the part of implementers, and emulating the behavior within an application just can't be done.

On the other hand, all of these interfaces can be provided, and routines placed behind them to perform much of the functionality as reasonably as possible.

The Choices

But let's get specific. The following is a summary of those issues in POSIX that were particularly difficult on traditional flavors of UNIX systems.

Job Control

POSIX defines functions and structures that permit user control of processes through some well known mechanisms. This is an optional feature of POSIX, and is unfortunately impossible to emulate on systems that do not have some form of Job Control to start with.

Since it is optional, and since there is rarely a need to use any of the features from within an application (almost never from within a portable application), we decided not to implement it.

It would be possible to provide almost all of the functionality on top of existing implementations that have the BSD-style job control. Unfortunately, POSIX has added a new level of abstraction to the traditional model — the session.

Providing all of the POSIX required functionality for session, process group control, and finally specific process control cannot be completely implemented without kernel modification.

Signals

The traditional UNIX process signaling system has been in its current state for many years. POSIX provides a new set of signal manipulation routines that allow for more robust handling of these events. The new signal interface has been dubbed "reliable signals" because it requires some actions to be atomic that formerly had vulnerable windows.

Specifically, the routine *sigsuspend* is guaranteed to cause the process to wait until a signal in a user defined set is delivered. Formerly this had to be done using several calls to *signal* followed by a call to *pause*. During this time it was possible for the signal you really wanted to be delivered without your current thread realizing it, thus causing the process to wait forever.

While the functions that are atomic under POSIX cannot be made so without kernel modifications, the functionality can be emulated using a complex signal handling system that captures all signals as they arrive, and delivers them to user provided routines only when requested to do so.

In addition to being very ugly, this solution is also slow. However, it does get the job done.

The disadvantage of this solution is that it is very difficult to satisfy the POSIX requirement of having a child process (a process that was created by the current process) know about the current process signal information. We have discovered no good solution for this.

Terminal Interface

One major difference between the most popular UNIX implementations has been in the low level terminal interfaces. I don't mean windowing systems, although that is also an important issue not addressed by either of these standards.

I am referring to the functions that allow manipulation of things like the character size in bits, speed of a serial port, flow control, etc. POSIX specifies a new group of routines that provide all of the traditional functionality, and can be implemented on top of existing implementations.

The difficult part of this emulation occurs on BSD systems. POSIX requires that a process be able to specify the minimum number of characters that a *read* will get from the terminal before giving them back to the application. On BSD systems you can wait for one or many.

POSIX also requires that the application be able to specify a time period that the system will wait for data from a terminal before returning. On BSD systems there is no similar facility.

Again, there doesn't appear to be any good solution for these problems.

Directory Operations

BSD has had routines that permit retrieval of the list of files in a directory for several years. AT&T provided a similar interface in System V.3. The standard interface for these functions, as defined by POSIX, is slightly different from either of these, but is very closely related.

It is important to note that the POSIX interface does not have the functions *seekdir* and *telldir*. These functions are found in BSD systems, but were not felt to be implementable in a portable fashion by the P1003.1 working group.

Instead of implementing these routines ourselves, we used a public domain library that works on BSD, System V, and many others. You can find a lot of portable code already implemented and placed in the public domain.

Atomic File Operations

POSIX requires the functions *rename* and *rmdir*. These functions allow the atomic renaming of a file and the removal of a directory, respectively.

These routines have been on BSD systems, but not on AT&T based implementations. To implement the functionality on those systems, the action must be performed as a series of calls – thus making it non-atomic.

Specifically, the BSD *rename* function could be implemented using the sequence *link*, *unlink*.

What We Did

The strategy we used was to write a group of headers (a file that is referenced by an application, and defines some standard system information) and library routines that provide all of the interfaces. POSIX requires that certain information be in the headers that already exist on most implementations, as well as some new headers.

ANSI C also has requirements on what information should be provided by traditional headers, and also requires some new ones.

The best way to provide for these requirements would be to augment the existing system files. However, as most people are (naturally) wary of modifying their vendor provided headers and libraries, we chose a less destructive method of installation.

During the installation process the existing headers are examined, and the important information from them is built into the new headers. These new headers are then installed in their own hierarchy of directories under */usr/include*. The libraries are compiled using these new headers, and then are installed as an additional library that must be actively linked to an application wishing to use the portability routines.

This project is now about 90% complete. Certainly it is far enough along that we are using the routines in applications (this has turned out to be the only way to debug them effectively).

Problem Revisited

Unfortunately, during the development of the portability libraries I came up against the second problem. My application required a user interface, and should really have had something like the traditional UNIX "curses" package under it for portability. (The "curses" package is a set of routines that permit an application to perform screen-oriented functions, and have those actions translated to the appropriate output sequences for virtually any terminal.)

This interface is not being worked on by any standards committee that I know of, but no other interface is either. I also needed a C language interface for a database, and a consistent interface for cross-network communication.

Since there are no formal UNIX standards for these yet, we took a different tack. If we know what requirements we have for a facility, and we know what traditional interfaces are available to do it, then we should be able to define our own "portable" interfaces which run on those traditional libraries. When a standard becomes available, we will just build our interfaces on top of that, and the application will continue to function.

Once a decision had been made about exactly what functionality was needed in our application world, we were able to implement it — the user interface on top of *curses / termcap / terminfo*, the database on top of a popular C database engine, and the networking stuff on top of both BSD4.2 and BSD4.1 socket libraries.

We tried to keep the code structured in such a way that new additional interfaces could be hooked in as necessary, while still having the code be fairly optimized. After all, a slow user interface is one that will not be used.

Boon Or Baggage?

What we ended up building is a minimalist's portability system. We have a set of libraries and headers that permit us to write code that should conform to 1003.1 and ANSI C, and we have additional headers and routines that provide a simple interface to some pretty complex facilities.

It will certainly be a long time before any set of international standards solves all of the portability problems associated with UNIX systems. Until that time, you can look at the components of that total solution in one of two ways: either as a boon that will make your software future more compatible, or as more baggage that software engineers need to carry around so that they can make their applications function on machines that have yet another, different interface.

By creating underlying libraries, these interfaces become the only thing a programmer has to use, all of the time. This should decrease software development time, while increasing software portability and readibilty.

When all is said and done, that is what software standards are all about. Sure it's a fantasy, but it's a good one.

Shane P. McCarron, Systems Analyst, NAPS International

USER INTERFACE

Is The X Window System A Standard?

By Martin R.M. Dunsmuir

With the move towards windowing systems in both the UNIX and PC worlds, many opportunities have arisen for the development of new interactive applications. For those UNIX Independent Software Vendors (ISVs) who are now considering their future product development strategy, the need to develop Graphical Applications is compelling.

Although the Xlib interface from the MIT X Consortium has emerged as the foundation Applications Program Interface (API) for UNIX Graphical Applications, many higher level issues remain undecided. Different vendors offer different programming and user interfaces. This diversity has left the ISV with so many choices that it is a major obstacle to serious applications becoming widely available.

This article looks at the reasons for the popularity of the X Window System and examines the question of whether, in spite of its apparent acceptance, it really is a standard that can be relied on.

The X Window Philosophy and Its Implications

What does the X Window System offer which makes it so interesting? The basic concept embodied in X which has been directly responsible for its widespread adoption is that it promises the ability to start and interact with applications from anywhere in a heterogeneous computer network. Its central function is to provide a standard graphical terminal emulator capability.

With almost all windowing systems, apart from X, some application code executes on the user's workstation. By contrast, X isolates all applications code on the so-called client machine. Client applications perform all I/O operations via remote procedure calls passed over the network to a server program controlling the screen and input devices on the user's workstation.

The advantage of this latter approach is that it allows applications to be run from any workstation that may or may not support a windowing system. The cornerstone of the system is the application level protocol which encodes communication between client and server — the X protocol (see Figure 1 below).

This prospect of freeing the user's choice of workstation while tying him to centralized "applications servers" has some interesting implications. Unless ISVs are very careful in implementing products based on X, they may find themselves pinned increasingly to proprietary systems instead of participating in the emergence of an Open Systems environment.

The reason I say this is that X is only part of the problem. Though for the moment there is a standard base set of services emerging under the X appellation, applications depend on more than a standard set of user interface services. They also rely upon the underlying OS services which are used as a means to implement the majority of their functionality.

In order to ensure the emergence and adoption of a true standard, all applications' interfaces need to be standardized. It is not enough to standardize the User Interface without standardizing the other operating system interfaces.

It is in the interests of ISVs to work for increased standardization of the X client services at all levels. They should avoid dependence on proprietary extensions, since these will tie them to a particular platform and subvert the reliability of the X system as a standard.

Elements of the X Environment

The key elements of the X environment which need to be standardized are application program interfaces (APIs), Applications Style, and Window Manager Behavior.

APIs are those interfaces provided to the programmer on the client side. It is important that these interfaces provide a standard set of functions to allow source level portability across multiple X implementations.

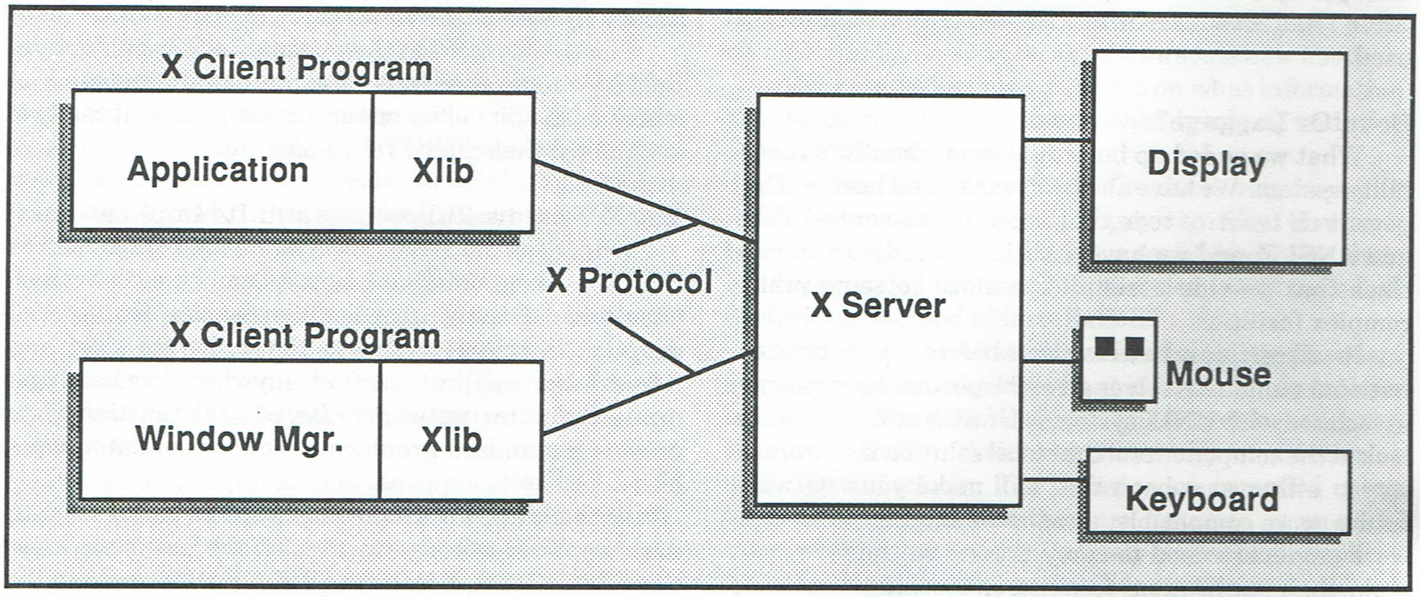
Applications Style and Window Manager Behavior are combined into what is termed the Feel of the User Interface. Standardized Applications Style governs the control elements (menus, dialogs and scroll bars, etc.) which are used to interface with the user. Window Manager functions, when standardized across different X implementations, offer users a consistent paradigm for manipulating active applications (e.g., moving, sizing, and iconifying).

The importance of standardized behavior cannot be overstressed. Without it, users have difficulty moving between different vendors' products, and the effort required to become proficient with new applications is significantly increased. Standardized behavior is a key feature of the popular windowing systems.

I do not consider the Look (or Appearance) of different implementations to be as significant as the Feel because it is quite possible to move easily between systems with different Looks but the same Feel. An example of this is Hewlett Packard's three-dimensional implementation of Presentation Manager Behavior, recently embodied in their proposal to OSF (called the Common X Interface or CXI). Users can readily use CXI and OS/2 PM interchangeably.

Let us look at the different elements of X systems

Figure 1: X Client – Server Model



and see how they relate to the model I have proposed (see Figure 2, below).

Standardized Elements of the X Client Architecture

In order to realize the full promise of X, namely the ability of any user to run any application available on the network, it is necessary to ensure that all X implementations adhere to a standard low level protocol. To achieve this, the MIT X Consortium, who are responsible for reference implementation and specification of the current X Window System, have defined and published the X protocol.

All vendors have signed up to the base X protocol and the C language binding to it, called Xlib. Xlib makes the base functionality of X available to applications, it embodies drawing functions, window creation and manipulation functions, and functions for control of variable features such as font and color selection.

The Xlib interface also defines standard events delivered to applications across a well defined interface. These events are sent by the server and other clients to communicate mouse and keyboard input and to initiate redraw and sizing operations implemented by client specific code.

The X Consortium has also agreed on a layer of functionality, built on top of Xlib, call the Xt Intrinsics, and are working to standardize conventions for the communication between different clients and, in particular, application clients and the Window Manager. This set of conventions is called the Inter Client Communication Conventions Manual (ICCCM).

The Intrinsics are significant because they implement a simple object management facility which can form the basis for higher level object oriented program interfaces. This is a common paradigm used by all popular windowing systems.

The ICCCM is important because it standardizes the advisory information which can pass between applications and the Window Manager, responsible for decorating active application windows with the elements which embody the standard behavior.

Higher Level API and Window Manager Standards

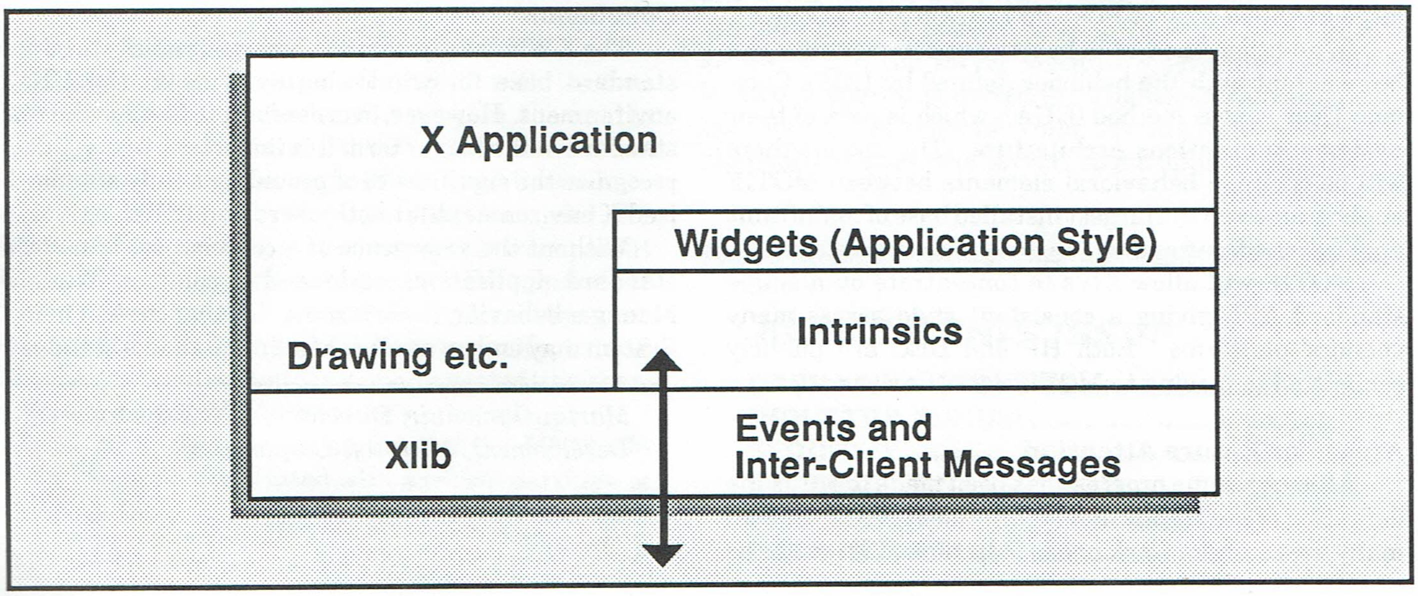
The intrinsics is a layer of software which supports the creation of window classes within an application. Using the intrinsics it is possible to define class handlers for standard and application specific windows and to have these class handlers called appropriately when events, destined for a particular class instance or window, occur. The intrinsics also allow the definition of inter-class messages and support their delivery in an analogous manner to system events such as keystrokes and resizing messages.

Typically, applications consist of a combination of application specific window classes used in conjunction with standard control classes for features such as menus, dialog boxes and scroll bars. It is these latter "controls" which implement the style of an application and allow it to share common behavior with others.

To date there have been a number of competing libraries of standard controls (known as widgets in X parlance). The most prevalent widget sets today are: DEC's XUI, implementing the DecWindows Style; AT&T and SUN's Openlook; HP's CXI; and, from the academic world, the Athena Widgets.

The widgets used define the behavior of applications

Figure 2: X Windows Client Architecture



and, because they define specific class message protocols, they represent a very important set of APIs. The widgets are in need of standardization to further solidify the specification of the environment which the programmer operates in.

In addition to the widget APIs, the other area where there is a significant level of diversity is Window Managers. Window Managers are important to users because, when combined with the widget defined application style, they define the User Interface.

They are also important because along with each Window Manager there is a set of Inter-Client communication conventions which are used to send hints to the Window Manager to control the manner in which an application is decorated with Window Manager controlled functions, such as sizing borders, iconization and system menus.

For example, applications often want to tell the Window Manager which forms of decoration they want and how they would like to behave (e.g., they won't respond to resizing below a certain size, etc.). There is yet another set of APIs associated with the delivery of these hints, and this is also an area where standardization is important.

I believe that the X Window System will become a true standard if one widget set and one Window Manager, with its inherent Inter-Client Communication Conventions, emerges as a standard for applications developers. Fortunately there have recently been some moves in this direction.

At the end of December, the Open Software Foundation (OSF) decided to adopt a standard Window Manager and standard set of widgets (based on a combination of XUI and CXI code) implementing the style of Microsoft Presentation Manager. This common behavior (called MOTIF by OSF) is an important and logical move because it promises a common style between the large installed base of PCs and UNIX workstations.

In addition, the PM behavior has commonality to a large extent with the behavior defined by IBM's Common User Access method (CUA), which is part of their Systems Applications Architecture. This means there will be common behavioral elements between MOTIF applications and the largest installed base of mainframe applications front ends.

MOTIF will allow ISVs to concentrate on a single standard API, giving a consistent style across many vendors' platforms. Both HP and DEC are publicly committed to moving to MOTIF later this year.

Areas For Future Attention

Although some progress has been made towards the definition of the standard X environment, there are still many areas where work needs to be done if X is to be as rich as other commercially available windowing systems. Particular areas which need attention are:

- The provision of standard fonts and the extension of the X protocol and APIs to make high calibre wysiwyg applications possible.

- The extension of X to provide standard support for hardcopy output. Currently the X API does not support drawing on printers and plotters. Although these functions may not require extensions to the server or protocol, it would be a significant benefit if applications could depend on a standard printer driver model and a standard set of APIs for rendering drawings. Presentation Manager, for instance, allows applications to use the same APIs for drawing on the display and on hardcopy devices. This greatly reduces the application writer's burden and leads to more consistent and simpler applications architecture.

- The UNIX shell and base tools, such as *ls*, *sort*, *grep*, etc., provide a standard environment for inter facing with and manipulating files and applications. Similar graphically oriented file management applications need to be developed so that users have a common paradigm to use for these purposes. Simple solutions, such as terminal emulators, are inadequate for this purpose in the commercial environment.

Finally, there are many extensions to X which have been developed and advertised by different vendors. These features need to be treated with care; only where they are a significant benefit and mature enough for standardization should they be adopted.

Conclusion

The X Window System has emerged rapidly to be the standard base for windowing systems in the UNIX environment. However, in order for it to become the real standard in the longer term it is important that vendors recognize the significance of providing a fully standardized X environment for both users and ISVs.

Without the emergence of a common set of APIs, a standard applications style and a common Window Manager behavior, there is some risk that the X Window System may end up as more of a dream than a standard.

Martin Dunsmuir, Director of UNIX Systems Development, Microsoft Corporation