AUUGN

Australian Unix systems User Group Newsletter

AUUG 88 Conference Issue

Volume 9 - Number 4

September 1988

Registered by Australia Post Publication No. NBG6524

The Australian UNIX* systems User Group Newsletter

Volume 9 Number 4

September 1988

CONTENTS

* UNIX is a registered trademark of AT&T in the USA and other countries.

AUUG General Information

Memberships and Subscriptions

Membership, Change of Address, and Subscription forms can be found at the end of this issue.

All correspondence concerning membership of the AUUG should be addressed to:-

The AUUG Membership Secretary,
P.O. Box 366,
Kensington, N.S.W. 2033.
AUSTRALIA

General Correspondence

All other correspondence for the AUUG should be addressed to:-

The AUUG Secretary,
P.O. Box 366,
Kensington, N.S.W. 2033.
AUSTRALIA

AUUG Executive

President	**Greg Rose** *greg@softway.sw.oz* Softway Pty. Ltd., New South Wales	Secretary	**Tim Roper** *timr@labtam.oz* Labtam Limited, Victoria
Treasurer	**Michael Tuke** *no net address* Edge Computer, Victoria		
Committee Members	**Frank Crawford** *frank@teti.qhtours.oz* Q.H. Tours, New South Wales		**Richard Burridge** *richb@sunaus.aus.oz* Sun Microsystems Austrlia New South Wales
	Chris Maltby *chris@softway.sw.oz* Softway Pty. Ltd., New South Wales		**Tim Segall** *tim@hpausla.aso.hp.oz* Hewlett Packard Australia, Victoria

Next AUUG Meeting

Regional Meetings will be held during February 1989, and a major Conference and Exhibition will be held in Sydney in September 1989. Futher details will be provided in the next issue.

Editorial

Welcome to the AUUG 88 Conference issue of AUUGN.

This Conference has attracted many interesing speakers and papers most of which you will find reproduced in the following pages. I hope you will find time to enjoy reading them.

I am now entering my third year as Newsletter Editor, and think that the Newsletter could still be improved. I cannot do this task ALONE. I would like the Newsletter improve its ability to attract papers during the rest of the year when there is not the incentive for the authors to present their paper at an AUUG Conference. I intend to continue the policy of reprinting the US and European User Group Newsletters as this adds a world perspective to the Newsletter.

The Newsletter is only as good as the material that it gather from its sources and I think it needs greater input from our members.

I will be attending the Conference and I would be more than pleased if you approached me concerning any ideas you may have for future Newsletters.

I wish to thank Tim Roper and his programme committee for their efforts in assembling the papers. Also I want to thank Katharine Ching and Peter Nankivell of Monash University, and my colleagues at Webster Computer Corporation Corporation for their assistance in producing the issue.

Again, I hope you enjoy this issue and please feel free to contribute an article soon.

REMEMBER, if the mailing label that comes with this issue is highlighted, it is time to renew your AUUG membership.

AUUGN Correspondence

All correspondence reguarding the AUUGN should be addressed to:-

John Carey
AUUGN Editor
Webster Computer Corporation
1270 Ferntree Gully Road
Scoresby, Victoria 3179
AUSTRALIA

ACSnet: *john@wcc.oz*

Phone: +61 3 764 1100

Contributions

The Newsletter is published approximately every two months. The deadline for contributions for the next issue is Friday the 14th of October 1988.

Contributions should be sent to the Editor at the above address.

I prefer documents sent to me by via electronic mail and formatted using *troff -mm* and my footer macros, troff using any of the standard macro and preprocessor packages (-ms, -me, -mm, pic, tbl, eqn) as well TeX, and LaTeX will be accepted.

Hardcopy submissions should be on A4 with 35 mm left at the top and bottom so that the AUUGN footers can be pasted on to the page. Small page numbers printed in the footer area would help.

Advertising

Advertisements for the AUUG are welcome. They must be submitted on an A4 page. No partial page advertisements will be accepted. The current rate is AUD$ 200 dollars per page.

Mailing Lists

For the purchase of the AUUGN mailing list, please contact Tim Roper.

Acknowledgement

This Newsletter was produced with the kind assistance and equipment provided by Webster Computer Corporation.

Disclaimer

Opinions expressed by authors and reviewers are not necessarily those of the Australian UNIX systems User Group, its Newsletter or its editorial committee.

Adelaide UNIX Users Group

The Adelaide UNIX Users Group has been meeting on a formal basis for 12 months. Meetings are held on the third Wednesday of each month. To date, all meetings have been held at the University of Adelaide. However, it was recently decided to change the meeting time from noon to 6pm. This has necessitated a change of venue, and, as from April, meetings will be held at the offices of Olivetti Australia.

In addition to disseminating information about new products and network status, time is allocated at each meeting for the raising of specific UNIX related problems and for a brief (15-20 minute) presentation on an area of interest. Listed below is a sampling of recent talks.

D. Jarvis	"The UNIX Literature"
K. Maciunas	"Security"
R. Lamacraft	"UNIX on Micros"
W. Hosking	"Office Automation"
P. Cheney	"Commercial Applications of UNIX"
J. Jarvis	"troff/ditroff"

The mailing list currently numbers 34, with a healthy representation (40%) from commercial enterprises. For further information, contact Dennis Jarvis (dhj@aegir.dmt.oz) on (08) 268 0156.

Dennis Jarvis,
Secretary, AdUUG.

Dennis Jarvis, CSIRO, PO Box 4, Woodville, S.A. 5011, Australia.

PHONE: +61 8 268 0156

UUCP: {decvax,pesnta,vax135}!mulga!aegir.dmt.oz!dhj
ARPA: dhj%aegir.dmt.oz!dhj@seismo.arpa
CSNET: dhj@aegir.dmt.oz

WAUG
Western Australian UNIX systems Group
PO Box 877, WEST PERTH 6005

Western Australian Unix systems Group

The Western Australian UNIX systems Group (**WAUG**) was formed in late 1984, but floundered until after the 1986 AUUG meeting in Perth. Spurred on by the AUUG publicity and greater commercial interest and acceptability of UNIX systems, the group reformed and has grown to over 70 members, including 16 corporate members.

A major activity of the group are monthly meetings. Invited speakers address the group on topics including new hardware, software packages and technical dissertations. After the meeting, we gather for refreshments, and an opportunity to informally discuss any points of interest. Formal business is kept to a minimum.

Meetings are held on the third Wednesday of each month, at 6pm. The (nominal) venue is "University House" at the University of Western Australia, although this often varies to take advantage of corporate sponsorship and facilities provided by the speakers.

The group also produces a periodic Newsletter, **YAUN** (Yet Another UNIX Newsletter), containing members contributions and extracts from various UNIX Newsletters and extensive network news services. **YAUN** provides members with some of the latest news and information available.

For further information contact the Secretary, Skipton Ryper on (09) 222 1438, or Glenn Huxtable (glenn@wacsvax.uwa.oz) on (09) 380 2878.

Glenn Huxtable,
Membership Secretary, **WAUG**

Comments from the Programme Committee Chair

AUUG88

Timothy Roper

Labtam Limited

Such a choice selection of guest speakers immediately made my task a pleasure. The theme of *Networking: Linking the UNIX† World* is certainly topical but having **Michael Lesk** to deliver the keynote address makes a delightful combination. As originator of *uucp*, the first and biggest network of UNIX systems, an early worker in computer typesetting and a ringer at novel applications and user interfaces, he will provoke a lot of thought about modern approaches to computer-based human communication systems.

John Mashey is no stranger to UNIX systems having been employed by Bell Laboratories in 1973 along with its first PDP 11/45. We look forward to benefiting from his expertise in hardware and software architectures and his knowledge of their current trends.

There is no doubt that the efforts of **Mike Karels** and his team have contributed greatly to the pervasiveness of networking software on many brands of UNIX system. What he has to describe about current and future developments in the Berkeley Software Distributions is sure to be seen in the marketplace very soon.

I know that the AUUG Management Committee is delighted to have **Ken Thompson** attending the conference and is further pleased that he has agreed to speak.

The Programme Committee is confident that this programme will provide something of particular interest, and a lot of general interest to most people. The *Future Directions* session on Wednesday morning promises to be a topical and lively discussion of interest to everybody. We are grateful to Pyramid Technology Corporation, AT&T Unix Pacific Co Ltd Japan, Hewlett-Packard Australia Limited and Sun Microsystems Australia Pty Ltd for sponsoring contributions to this debate.

On behalf of the Programme Committee I am pleased to congratulate **Rex di Bona** on the paper for which he has earned the AUUG prize for the best paper submitted by a full time student. In *Distributed Trouble: The University of Sydney Experience with Networked Workstations* he gives a timely and detailed insight of the increased need for attention to security issues in the context of local area networks.

I would like to thank the Programme Committee for their assistance, the authors for their cooperation, John Carey for making these proceedings a reality, and my employer for the latitude and resources necessary to coordinate those individual efforts.

It is my hope that you will go away feeling inspired and refreshed by the offerings of these three days. Please remember that it is not too early to start thinking about *your* paper for the next AUUG Conference and Exhibition.

† UNIX is a trademark of Bell Laboratories.

AUUG 88

Australian UNIX* systems User Group
1988 Conference and Exhibition

Tuesday 13th to Thursday 15th September, 1988
Southern Cross Hotel, Melbourne

PROGRAMME

DAY 1 - TUESDAY 13 SEPTEMBER

0900-1000 Registration

1000-1100 KEYNOTE ADDRESS
Unix Networks: Why Bottom-Up Design Beats Top-Down
Michael Lesk
Bell Communications Research

1100-1130 Coffee & Exhibition Viewing

1130-1200 *Future Telecommunication Developments*
Steve Jenkin, Softway Pty Limited

1200-1230 *An Overview of the Sydney University Network version IV*
P. R. Dick-Lauder and R. J. Kummerfeld
Sydney University

1230-1400 Lunch

1400-1430 *NeWS Programming*
Tim Long
Information Concepts

1430-1500 *An X11 Overview*
Tim Segall
Hewlett-Packard Australian Software Operation

1500-1530 *Hows a Windowing System*
Greg Rose, Softway Pty Limited

1530-1600 Coffee & Exhibition Viewing

1600-1630 *An Implementation of FTAM*
Andrew Worsley
CSIRO Division of Information Technology

1630-1700 *ISO/OSI under Berkeley UNIX*
Mike Karels, Computer Systems Research Group,
University of California at Berkeley

1730-1900 COCKTAIL RECEPTION

DAY 2 - WEDNESDAY 14 SEPTEMBER

0830-0900 Registration

0900-0930 *The Future of UNIX Software:*
The Open Computing Platform for the 1990s
Larry Crume, AT&T Unix Pacific Co Ltd, Japan

0930-1000 *Future Berkeley UNIX Developments,*
Mike Karels, Computer Systems Research Group,
University of California at Berkeley

1000-1030 *Unified UNIX*
John Young, Sun Microsystems Australia Pty Ltd

1030-1100 Coffee & Exhibition Viewing

1100-1130 *Open Software*
Tom Daniel, Hewlett-Packard Australia Limited

1130-1200 *ABI and OSF: Technology and Future Impact on UNIX*
Ross Bott, Pyramid Technology Corporation

1200-1230 PANEL SESSION
Future Directions of UNIX
Ross Bott, Larry Crume, Tom Daniel,
Mike Karels, John Young

1230-1400 Lunch

1400-1430 *Distributed Trouble: The University of Sydney experience with networked workstations.*
Rex di Bona, University of Sydney

1430-1500 *Installing and Operating NFS on a 4.2 BSD VAX*
Jim Reid, Strathclyde University, Glasgow

1500-1530 *Networks: The Legal and Social Ramifications*
James Watt, Griffith University

1530-1600 Coffee & Exhibition Viewing

1600-1700 *Human Interface Issues*
Michael Lesk, Bell Communications Research

1700-1730 AUUG General Meeting

1900-2300 CONFERENCE DINNER

DAY 3 - THURSDAY 15 SEPTEMBER

0830-0900 Registration

0900-0930 AUUG ACSnetSIG Meeting

0930-1000 *Link Management and Link Protocol in SUN IV*
P. R. Dick-Lauder and R. J. Kummerfeld
Sydney University

1000-1030 *A New C Compiler*
Ken Thompson, Bell Laboratories

1030-1100 Coffee & Exhibition Viewing

1100-1130 *The Development of Distributed Processing within the Queensland Government*
Peter Waller and Tony Ashburner
CITEC, Queensland State Government
Tom Crawley and Rob Cook
CiTR, University of Queensland

1130-1200 *Time Synchronisation on a Local Area Network*
Frank Crawford, Q.H. Tours
Jagoda Crawford, Australian Nuclear Science and Technology Organisation

1200-1230 *On MICROLAN 2 – An Office Network*
Richard Lai, ICL Australia Pty Limited

1230-1400 Lunch

1400-1430 *Writing portable UNIX programs*
Stephen Frede, Softway Pty Limited

1430-1500 *Real-time disk I/O scheduling on UNIX operating systems*
Jeongbae Lee, Electronics and Telecommunications Research Institute, Korea

1500-1530 *The Process Life Cycle in STIX*
Sunil K Das and Aarron Gull
Department of Computer Science,
City University London

1530-1600 Coffee & Exhibition Viewing

1600-1700 *RISC and the Motion of Complexity*
John Mashey, MIPS Computer Systems

1700 Close

AUUG 88 Key Speaker Biographies

Ken Thompson
AT&T Bell Laboratories

Ken Thompson was born in New Orleans, Louisiana in 1943. He attended the University of California at Berkeley and received B.S. and M.S. degrees in Electrical Engineering.

In 1966 he joined Bell Laboratories where he has worked until the present. In 1975, he returned to Berkeley to teach Computer Science for a year.

Ken Thompson and Dennis M. Ritchie, the principal designers of the UNIX system have received recognition many times. In 1983 they were presented with the most prestigous award in computing, the ACM A. M. Turing Award.

Ken is also one of the principal designers of *Belle*, the former World Computer Chess Champion and five times winner of the North American Computer Chess Championship.

Michael E. Lesk
Bell Communications Research

With a prescient market instinct, Michael made text formatting accessible to the masses with the generic macros **-ms**, which were to *troff* what a compiler is to assembly language. He rounded out **-ms** with the preprocessors *tbl* for typesetting tables and *refer* for bibliographies. He also made the *lex* generator for lexical analysers. Eager to distribute his software quickly and painlessly, Michael invented *uucp*, thereby begetting a whole global network. *Uucp* gave operational meaning to the phrase "Unix community". News now travels electronically among users all over the world; and technical collaborations proceed between distant locations almost as easily as within one building. Over the years, often helped by Ruby Jane Elliot, he initiated fascinating on-line audio, textual, and graphical access to phone books, news wire *apnews*, and weather. With Brian Kernighan, he was responsible for the UNIX computer-assisted software *learn*.

Mike Karels
University of California at Berkeley

Mike Karels is one of the leaders of the Computer Systems Research Group at the University of California, Berkeley. This group is the current focal point of the world's UNIX research initiative, and has been since AT&T stopped distributing the work done by Bell Laboratories, where UNIX was originally created in the early 1970's.

During the 80's, Berkeley has created, and collected from other sources, the most advanced UNIX system currently available, overcoming many of the limitations and restrictions of the original Bell Laboratories releases. So successful has this been, that AT&T have now adopted, or are in the process of adopting, most of Berkeley's enhancements, either unaltered, or with some cosmetic variations, as is to be expected when a research effort is transformed into a commercial product. The Berkeley enhancements are already supported by most of the various UNIX vendors, and have been for some years, the offering of a UNIX product without "Berkeley enhancements" is almost unheard of today.

As one of the leaders of the Berkeley group in recent years, Mike has been one of the principal architects of future UNIX commercial systems. His decisions will have a lasting effect on the future of computing.

The theme of the AUUG Winter Conference and Exhibition is "Networking", and in this area Mike is a particular expert. He and Van Jacobsen, from the Lawrence Berkeley Laboratory, have recently released into the public domain a much enhanced implementation of the TCP/IP networking protocols for UNIX. Earlier versions of this code now form the basis of many vendor's TCP networking products.

We are sure that Mike will be able to inform us on the current status of networking in UNIX, an area where Berkeley are the clear leaders, and of future plans in this important area. Mike will also indicate the current, and expected future, status of the UNIX research work at Berkeley, from which we should be able to draw some informed conclusions as to the likely appearance of commercial UNIX releases in the next decade.

John R. Mashey
Vice President, Systems Technology
MIPS Computer Systems

Dr. Mashey joined Bell Laboratories in 1973, joining the Programmer's Workbench department the same week it received its first PDP 11/45. He worked on various UNIX-related projects, including PWB/UNIX, the merger of UNIX versions that resulted in UNIX/TS 1.0, and UNIX-based applications for use in the Bell System. In 1983, he moved to Convergent Technologies, ending as Director of Software Engineering for the Data Systems Division. In 1985, he joined MIPS Computer Systems, where he helped design the MIPS R2000 RISC microprocessor, and managed operating systems, networking, and software QA. He was an ACM National Lecturer for 4 years, and has given about 200 public talks on software engineering, UNIX, and RISC architectures.

AUUG 1988 Conference Papers

In order of presentation

UNIX Networks: Why Bottom-Up Design Beats Top-Down
Keynote Address
Paper
Michael Lesk
Bell Communications Research

Future Telecomunications Products
Abstract
Steve Jenkin
Softway Pty. Ltd.

An Overview of the Sydney University Network version IV
Paper
P.R. Dick-Lauder
R.J. Kummerfeld
Sydney University

NeWS Programming
Abstract
Tim Long
Information Concepts

The X window system - an overview
Abstact
Tim Segall
Hewlett-Packed Australia Software Operation

Hows a windowing system
Abstract
Greg Rose
Softway Pty. Ltd.

FTAM and OSI standards are hard to implement from scratch
Paper
Andrew Worsley
CSIRO Division of Information Technology

AUUG 1988 Conference Papers

continued

ISO/OSI under Berkeley UNIX

Paper Unavailable

Mike Karels
Computer Systems Research Group
University of California at Berkeley

The Future of UNIX Software: The Open Computing Platform for the 1990s

Paper Unavailable

Larry Crume
AT&T UNIX Pacific Company Limited, Japan

Future Berkeley UNIX Developments

Paper Unavailable

Mike Karels
Computer Systems Research Group
University of California at Berkeley

Unified UNIX

Paper Unavailable

John Young
Sun Microsystems Australia Pty. Ltd.

Open Software

Paper Unavailable

Tom Daniel
Hewlett-Packard Australia Limited

ABI and OSI: Technology and Future Impact on UNIX

Synopsis and Outline

Ross Bott
Pyramid Technology Corporation

Distributed Trouble: The University of Sydney experience with networked workstations

Winner of AUUG prize for best paper submitted by a full time student

Paper

Rex di Bona
University of Sydney

KEYNOTE ADDRESS

Unix Networks: Why Bottom-Up Design Beats Top-Down

Michael Lesk

Bell Communications Research

ABSTRACT

Distributed operating systems, designed with an idea of central control, have not gained many adherents in the fiercely independent world of Unix systems. By contrast, netnews survives despite attempts to stamp it out. As the bottom up designers extend from mail to remote file systems, and the motivation for top down design starts to shrink with faster workstations, one wonders whether any of the distributed designs will survive. This paper will discuss some Unix networking projects, particularly the early history of UUCP, with analogies to the Newcastle connection, a cached file system project, and various other Unix networks, ending up with a brief mention of DUNE (Bellcore) and Plan9 (BTL). So far, the shared file systems and mail projects are way ahead of the centralized distributed systems.

There are lots of arguments about network design. We have stars vs. buses. We have token rings vs. CSMA/CD. We have left-first addressing vs. right-first addressing. But perhaps one of the most important from the practical point of view is the choice between centrally administered networks and locally accumulated networks. UUCP (Unix* to Unix copy), of course, is the prime example of a locally grown network; there is no central administration at all and so nobody can even tell you how many sites there are on it. By contrast, most people who design networks believe in central control. In this talk I'm going to discuss the past of UUCP and the future of more general networks, to explain why I think bottom up design is the best course.

UUCP began as a cheap kludge. Originally what I really wanted to do was remote software maintenance. I had been rather drowned in keeping things like -ms and tbl running on about 40 computers around Bell Laboratories, and I counted for a while and discovered that about 70% of the bug reports were fixed by simply giving the complainant the latest version of the software. So it seemed that some kind of automatic distribution would be a help. There were the usual questions ("I don't want anybody installing new software on my machine without checking with me") but I thought I had an answer to that, based on a scheme of automatic regression testing. The more immediate problem, however, was how to get software from one machine to another. I wasn't anxious to do unnecessary work, so I inquired around Bell Labs for good networks that might be used to connect all the Unix systems. I found at least three different groups, each of which insisted that they had the solution and in about a year they would begin installing their magic carpet on all the machines. Well, I didn't want to wait a year (I also had some skepticism about these groups, mitigated by the thought that there were so many of them). So I started looking for a cheap way to get things from one place to another. There was already a "cu" program (call-unix) that would dial up one system from another, so I thought I could try running it from a script. It seemed quickly to make more sense to write a C program; the bootstrap was only about 200 lines long. This was of course without any administration or anything else. So when people called with a bug report, I would ask for their userid and password (in those days, this was

* Unix is a trademark of AT&T - Bell Laboratories.

always given immediately) and then put their machine in the table. Pretty soon there were about 50 machines connected, and I could send files to any of them.

The job of keeping UUCP running then prevented further progress on the software maintenance front. During much of this time I was fighting a rear-guard action against the assaults of the managers who thought this program was an awful security hole and should be abolished. I lost, as you all know. (In the 17th century a man named John Hill attempted to establish a privately run penny post for England, but was stopped by the Cromwell regime which objected to both the loss of revenue and the loss of their ability to read letters.)

The good side of all this management interest was that UUCP was rewritten, with elaborate administrative checks. This work was done first by Dave Nowitz, and then again later by Brian Redman, Peter Honeyman, and Dave Nowitz. My thanks to them; they turned a research kludge into a fairly robust program. Some of the administration added was an advantage (you now get told when your mail can't be delivered); a lot of it has meant that UUCP is not a useful interface for closely linked communities, because it can not deal directly with user files, always working through spool directories.[1,2]

Today, better networks have made much of the dial-up nature of UUCP obsolete. The performance of hardwired networks, and the general availability of links such as Ethernet or Datakit, means that few people would want to use dial-up within a building. Without dial-up, the kind of queuing and administration in UUCP is no longer sensible. Numerous proposals have been made for replacement, redesign, and the like. One is even familiar to you since it runs successfully here.[3] But I think some of the lessons from the program are useful and can be applied in future networks. For example, UUCP has had features providing a choice of access methods, the idea being that if there were both a high-speed and a low-speed access method, you'd try the high-speed first and the low-speed route when the first choice fails. This turns out to be a bad idea; if people get accustomed to a high-speed route, they send so much material that you are better off waiting for it to become available than using a low-speed route which will take very much longer. This lesson was learned between 1200 and 300 baud.

The script-driven style of UUCP internals was also useful; the major difficulty has been the inability of such scripts to handle more complex choices and procedures. But the adaptability of UUCP to a variety of system procedures has been fairly good.

A more important message is the importance of being able to maintain the same software interface while switching the hardware connectivity underneath. UUCP originally started with dial-up connections but now the typical mail user has no idea what the actual link between sites is.

Another major lesson is the need to build networks which do not demand to "own" the systems they are connected to. Many designers of networks can think of schemes which require kernel modifications and effective control of the systems on the nets. This is, in practice, intolerable. Many users can't change their kernels; most don't want to. A network has to plug into a variety of systems at the minimum level of disturbance. Bitnet, for example, follows this lesson. It has been very successful connecting a large group of complex systems, many IBM MVS, which are not easily modified.

Electronic libraries are another example of this general problem. It is very tempting to design a system in which you own all the data, store it on your machine, and then offer various services to people who dial in. This is even efficient and effective. However, it doesn't match what the publishers want. In general, they are worried about the consequences of access to electronic data, and they respond to the fear by maintaining control over the data. Often they have particular systems of their own, and insist that the only access will be via this system. It is inconvenient to realize that different publishers

---

[1] D. A. Nowitz, P. Honeyman, and B. Redman, "Experimental Implementation of UUCP - Security Aspects," *Uniforum Conference*, pp. 246-250, Washington DC, January 1984.

[2] R. Kolstad and K. Summers-Horton, "Mapping the UUCP Network," *Uniforum Conference*, pp. 251-257 It is also no longer sensible to bootstrap UUCP onto a new machine by sending a short, 200-line hook over with "cu"; the distribution is too large., Washington DC, January 1984.

[3] P. Dick-Lauder, R. Kummerfeld, and R. Elz, "ACSNET - The Australian Alternative to UUCP," *Summer USENIX Conference*, pp. 11-17, Salt Lake City, Utah, June 1984.

will not put their databases together; it resembles the days when movie actors were under contract to particular studios and certain obvious pairs never acted together. Similarly, one may not be able to combine information from a dictionary and a thesaurus unless they come from the same publishing house.

What of the future? Computer networking today is rather complex.[4,5] Many large installations have to employ gurus to deal with networks and remember odd pieces of information such as "the Netnorth-Bitnet gateway is at British Columbia" or "the Bitnet-Arpanet gateway is at CUNY". (For a while my mail to Toronto and Waterloo went from New Jersey to Ontario via Vancouver, a 6,000 mile path for a 400 mile trip. This proves bandwidth really is cheap today.) But the complexities of networks are a problem in making electronic mail ubiquitous. I did not expect fax growth to overtake email growth; part of this is the standardization on fax (another part is the formatting capability it allows). It would be better if we could make electronic mail compatible among the communities and systems that we all use.

How is universal electronic mail, similar to paper mail, going to come about? There are two general approaches, the centralized and the separated. The centralized approach is exemplified by ISDN. ISDN is an international standard, and is intended to be offered as a service by the telephone companies. It has been an awfully long time coming, and even after implementation starts it is likely to be quite a while before it reaches any kind of ubiquity. The alternative is the continued accumulation of local networks. Most large businesses today have some kind of internal electronic mail. The research community is connected through a linkage of these internal systems either through the Internet/Bitnet systems, or just via dial-up connections similar to UUCP.

The centralized approach creates several difficulties. First, it often requires years of negotiation to overcome administrative difficulties. As computers get cheaper, they become more numerous and it becomes impossible to suggest that the various owners get together and agree on anything. I can not imagine, for example, trying to force the staff of Bellcore to agree on one word processing system. (I can imagine translating a few into a standard form, though). You can get people to buy something new and add it to their system; you can not get them to give up something they have been using and that they like. You can hardly get them to give up something they have been using and that works, even if they don't like it. Think of how long it took to get domain addressing throughout the Arpanet.[6] Of course, the need to remain outside the operating system delays really high performance networks. We have to hope that the OSI model will let us provide these. Its complexity does tend to discourage this.

Another difficulty of centralized systems is the tendency to multiplex at too low a level. In order to achieve high throughput, a network is going to want high-bandwidth channels; in order to minimize costs, these will be shared. But this is inefficient. Ideally, multiplexing for minimal cost should be done only once, by the transmission carrier which actually handles the messages. If it is presented with all the traffic, it can come up with an arrangement of circuits which will minimize cost or maximize reliability, or whatever the goal is. If only part of the traffic is presented, then only a partial optimization can be done. Present bureaucratic and regulatory constraints often encourage the use of high-cost pathways. I hope that the advent of optical fiber will lower the cost of transmission enough that optimization of link cost won't matter as much.

Most important, however, of the difficulties with centralized networks is the extension of human bureaucracy to electronic networks. We are all familiar with mail headers that are longer than the messages they carry, and with the enormous growth of mail programs. I can remember when Unix mail was about four pages of source code; today the source code is 1300 pages. This has, in effect, made the overhead of connecting to networks and maintaining a mail interface much higher. Worse, it is not a routine operation: various networks serve academic sites, sites connected to particular government agencies, sites using particular machine styles, and the rest. To join a net like Arpanet or Janet is not

---

[4] J. S. Quarterman and J. C. Hoskins, "Notable Computer Networks," *Comm. ACM*, vol. 29, pp. 932-971, 1986.

[5] D. M. Jennings, L. H. Landweber, I. H. Fuchs, D. J. Farber, and W. R. Adrion, "Computer Networking for Scientists," *Science*, vol. 231, pp. 943-950, 1986.

[6] Craig Partridge, "Mail routing using domain names: an informal tour," *Summer USENIX conference*, pp. 366-376, Atlanta, Ga., June, 1986.

routine; you have to be a member of the approved community. This is causing electronic mail systems to fragment into systems serving particular groups, and frustrating those just outside the boundaries. For example, last year the British temporarily cut off transatlantic mail access to a particular set of users who were characterized by the fact that they DID pay their bills themselves. It all makes sense, sort of, if you know the whole story, but it is infuriating to those who lose service.

What's happening next? Distributed systems are spreading. We are beginning to see a continuum in the level of connection between systems. The advent of remote file systems, starting with libraries like the Newcastle Connection, has made some systems able to do the kind of remote copy that UUCP was originally intended for. There are transparent systems where the user doesn't even care which CPU is running what. More commonly, today, are systems connected (often by Ethernets) that allow remote copy and execution but do keep in mind who's running where. This makes security easier to enforce. We still don't have, in routine use, geographically distributed transparency; but we will soon, since this is now available in some experimental systems.[7] Soon, we should have distributed file systems connecting machines which do not run the same operating system, following the OSI models. Ideally, we can get a smooth connection between distributed systems and mail.

But more important to me is to have ubiquitous electronic mail. How is this to be achieved? I hope that UUCP will serve as a model for the development of connections between systems of different types, so that electronic mail can become as familiar as word processing and facsimile transmission. This involves some technical questions and some economic questions.

Technically, we need a "gateway" (as described by Judge Greene) which would interface all kinds of electronic transmission. It should permit all of information services, mail, file transmission, and other services. It should not require kernel modifications to computer systems. Just text-based electronic mail does not consume enormous bandwidth, nor extreme efficiency. How could such a system be built? I would suggest an idea of Robin Alston's (British Library): rather than try to standardize all systems, try to just get each to describe its format. That is, agree that any system, upon connection, will be prepared to describe its basic operations (in BNF, or some kind of object description language, or whatever formalism seems appropriate). Then, each system could ask upon connection for a description of the expected format. What I can not see working are isolated systems which hope that everyone will use them for electronic mail and nothing else (eg the initial version of British Telecom's Telecom Gold service).

Another technical problem is to deal with security. Security precautions present difficulties to people trying to do work. An efficient security precaution presents much greater difficulties to an intruder than it does to a legitimate user. The problem is that ordinary users are willing to spend, on every transaction, much less effort than a thief or vandal is willing to spend to do just one transaction. In practice I believe that we should try two general approaches to security: (1) rely more on authentication, so that we know WHO did something, than on sorting legal from illegal acts; and (2) rely more on a variety of small barriers, rather than trying to find a single technical solution to security, since the greatest dangers often turn out to be human frailty rather than technical decipherment. I look forward to the ability of digital phone switches to do calling number identification so that we can identify the people breaking into systems.

The administrative problems are more severe. Electronic mail today is rarely paid for by the message and often is paid for in ways that make it difficult for people to join even if they are willing to pay the bill. We need networks that are on a straight fee basis; fortunately, systems like UUCP and Bitnet can be reasonably cheap. This should make it possible to have a system to which universities, corporations, and individuals can all connect. If connecting to an electronic mail system merely meant the payment of a fee and the installation of some software, rather than elaborate legal negotiations, I believe it would spread much faster. I do not think that a purely separate mail system will prosper, though, since people prefer to use a single computer system for most of their business, not to have separate machines for every purpose. For those that don't have computers at all, there are various systems (eg Dan Nachbar's at Bellcore) which can provide a low-cost paper printing electronic mail systems.

---

[7] J. L. Alberi and M. F. Pucci, *The DUNE distributed operating system*, December 1987. Private memorandum

What I do not believe will work is an imposition of a single system of electronic mail from the beginning. The telephone network, the postal system, the railways, the electric utilities, and the airlines all started as isolated instances. Here in Australia your railway gauges still show the remains of the independent planning, and yet the trains ran. In some countries, some of these utilities have been combined; others have been kept separate. But in all cases, standardization followed technical development. Technology may provide too many ways to do something, but planners will provide nothing.

Fortunately, the trend in general today is against centralized systems. European countries are thinking about breaking up their PTTs; the UK has separated British Telecom from the Royal Mail and is now separating long distance from local service. I hope that international electronic mail can prosper as a relatively uncontrolled service, rather than automatically attaching to any particular organization.

In the longer-range future, what would a "super-UUCP" look like? Well, it won't be a Unix program anymore; it will connect all kinds of systems. It would take from the Arpanet higher speed services; it would take from the standard UUCP the idea of no administration. It will have some pricing for its services. It will have positive user identification for security (which will also mean that it could be used for commercial transactions). And it will be as ubiquitous as the other basic communications services: paper mail, voice mail, facsimile mail, and telephone calls. The historians of the future, instead of despairing because the evanescent telephone conversation has replaced the permanent written letter, will despair at the quantities of electronic mail saved in historical records (although computer searching tools may ameliorate the difficulty). And even average individuals may have the attachment to electronic mail, if not to netnews, that they felt for paper mail a generation ago:

> "None shall hear the postman's knock
> without a quick of heart;
> for who can bear
> to feel himself forgot?"
> -- W. H. Auden, "Night Mail"

Future Telecommunications Products

Or, what's coming our way in the next few years??

Steve Jenkin
Softway Pty. Ltd.

ABSTRACT

It's obvious some things are here to stay, like the Plain Old Telephone Service (POTS), and commercial facilities like fax. Others, like Telegrams and Telex, are evaporating.

Recent new services, like "Viatel" and cellular phones, have really taken off, but only in businesses due to the tarrifs and set-up costs.

As with all new products, the ones that really take-off and survive provide utility at a competitive price. The most obvious new telecommunications products are X.400, ISDN, Metropolitan Area Networks, and Broadcast Services. Costs and availability of these services are yet to be announced, but that doesnt stop a little inspired guesswork, does it?

X.400 is an electronic mail standard. It will do to the plethora of propriatory e-mail services what X.25 did to data services in 1980. Some will prosper, some will disappear or merge, all will have to support the new standard. Users will benefit from "Universal Connectivity", but probably not lower prices.

ISDN, Intergrated Services Digital Network, sounds like a "Really Good Idea". It is a bit of a let down. If you want circuit-switched 64kbit telephone-type links, it is great. You even get to have timed local calls. The Network Provider, Telecom, stands to gain the most from the introduction of the ISDN. Initially users will see no benefits or any of the massive savings.

MAN's, will do for public networks what Ethernet did for Local Area Networking. At 34-150Mbit its high bandwidth and accessibility in and between the Central Business Districts will revolutionise large scale networking, if Telecom gets its pricing right.

Don't hold your breath for any of these new services. 1990 is the first you'll see anything of them, and then only to "selected customers". The new "mean and lean" deregulated Telecom is going after marketshare and profits. They are very strongly targetting the top 300 companies.... and continuing to "milk the cash cows" of telephone services.

In the futher, fuzzier future, say 1995, lies another 'exciting development' :- Broadband ISDN (B-ISDN).

We'll all get 4*150Mbit optical fibres into our homes and be able to dial-up wonderful new services like High Definition TV, Video Libraries and Services, Computer Databases. And we'll all have TV-phones in our personal workstations at home and work, to avoid actually having to Travel to make those important meetings and conferences.... Of course, it will all cost "Next-to-Nothing".

Getting the best value for your money, or just minimising your comms costs is already a complex and everchanging task, and is only set to get worse. Telecom offers little help now, and is unlikely to change.

Costs, advantages, and trade-offs of basic services:- leased lines, dial-up, fax, telex, modems, and Austpac will be discussed. Foward planning, compatibility issues, and the impact of future services and standards will also be covered.

An overview of the Sydney University Network version IV.

P. R. Dick-Lauder
R. J. Kummerfeld

Sydney University

1. Introduction

This paper gives an overview of the design and implementation of version 4 of the Sydney Uni Network software (otherwise known as SUN IV, not to be confused with the workstation of the same name). The current version of this software (SUN III) is used to build the Australian Computer Science Network or ACSnet and has been installed on well over 400 systems. The new system is a complete redesign and reimplementation that retains the same functionality while providing improvements in many areas.

The software implements a message handling system that provides message services such as electronic mail, file transfer, news and directories across a wide area network of computers running the Unix operating system. The system is constructed in three layers:

- message protocol layer
- routing layer
- node to node transport layer

Message service protocols are provided at the message protocol layer. The layers below the message protocol layer provide an end to end message delivery service. A message service such as electronic mail can request that a mail item be delivered to the mail system on a remote machine. The routing layer at each machine is then responsible for passing the message from machine to machine across the network until the destination is reached and the message is passed to the message protocol layer for handling. The node to node layer transports messages between machines.

2. Naming and Addressing

In SUN IV naming and addressing in the message transport layers has changed significantly from that used in SUN III. In SUN IV a name is a type/value pair where the type and value are character strings containing alphanumeric and a small set of special characters. An address is a set of names with a defined ordering on the types. For example, the address of a machine at the Computer Science Department in the University of Sydney might be:

Node = basser
OrgUnit = "Computer Science"
Organisation = "Sydney University"
Country = Australia

With the type ordering from least to most significant being Node, OrgUnit, Organisation, Country.

Very flexible mapping facilities are provided that allow synonyms for the type names and values as well as translation to and from the non-typed addresses used in the current system. For example, the address given above may be translated from "basser.cs.su.oz.au". The new form of addressing is available to message level protocols but current message protocols can be used without change.

3. Regions

The address space of the network is divided into *regions* using type information from node addresses. A *region* is a 'continuous part of a surface or space'. In a network context a region is a fully connected area of the network and a *regional address* is one that includes names that indicate that an object belongs to a particular region. Examples of regions would be a country or an organisation or organisational unit. Regions are used by the routing layer to reduce the amount of information held in routing tables at each node and decrease the processing time when performing routing calculations.

The SUN III system had similar structures known as 'domains' that were also used to reduce the table sizes and processing time. The main difference is that SUN IV region tables contain complete address information about each node. SUN IV also allows regions that are not fully internally connected and regions that may not be traversed by messages travelling between nodes not in the region.

4. Routing

Messages are transported along an optimal route to the destination. The SUN III system provided for shortest path routing using link speed and cost as the metric. SUN IV also provides this but the metric for choosing the best link is now much more flexible and is related to the desired quality of service. Quality of service can be specified in terms of speed of delivery and cost of delivery. System administrators can specify costs for links and the speed is determined dynamically by monitoring queuing delay on each link. Links can be specified to be unidirectional, but in the bi-directional case, links costs and queueing delays apply separately to each direction.

More control over routing is provided to administrators through the use of "link restrictions". The restriction is in terms of the addresses allowed on messages travelling over a link, so that messages whose addresses lie outside the restriction must choose another link. For example, a site may therefore avoid carrying "through" traffic by specifying a restriction that all messages travelling over any link into the site must have an destination within the site.

Routes chosen by the network may be coerced so that all routes to a particular "region" are forced to take a particular link.

Messages addressed to particular regions may also be explicitly *forwarded* to other regions. This can be used, for example, to cause messages with organisation-level addresses to be forwarded to nodes where address resolution data-bases may be kept.

As with SUN III messages may be sent to a single destination, a group of destinations (multicast) or all destinations in a region (broadcast). In each case the optimal number of message hops are used.

5. Messages

Messages in SUN IV have much the same format as in SUN III, except for the addition of "ID" and "Part" fields in the header.

Each message is given a unique identifier at its source, which together with its source address uniquely identifies it across the whole network.

Messages can be broken into parts, and the parts transmitted separately. The "Part" field in the message header, together with the "ID" field, is used to join the parts after they have all reached their common destination.

Individual messages can also be *forwarded* from one site to another. This is done on a message handler basis, so that mail messages for particular users may be forwarded to a different site than their files. This is very much more efficient, for example, than having the mail delivery sub-system perform the forwarding.

6. Node to Node Protocol

The node to node message transport protocol has been redesigned and a new implementation technique used. It is a full duplex protocol that multiplexes transmission of several messages in each direction simultaneously. This allows higher priority messages to overtake lower priority. Nine priority levels are specifiable by message senders.

The new protocol has been designed for maximum throughput with minimum transmission costs, and to avoid interference with link-level protocols such as X.25, TCP/IP, or modems with internal buffering protocols. Another criterion was to allow maximum throughput even when many small messages are being transmitted, a condition that noticeably slows the SUN III protocol. Input and output have been decoupled by using one process for each, and all incoming messages are handled by a single permanently running routing process.

7. Link Management

Link management is performed in three areas:

- node description (commands file)
- connection control (config file)
- call script language

The commands file specifies information about the node such as its full address, type hierarchy and mapping information. It is now mandatory to supply some specific site description details, such as organisation name, and contact phone number. Other site specifications are compatible with network mapping data-bases. Links are

parameterised with cost and message delay times, and any addressing restrictions for messages that may be carried on the link.

The configuration file specifies the links that connect the node to its neighbours and the times that calls should be made on each link.

A link description specifies the link type and how a connection is to be made on the link. The detailed call establishment procedure is described in a program written in a new language designed for the purpose.

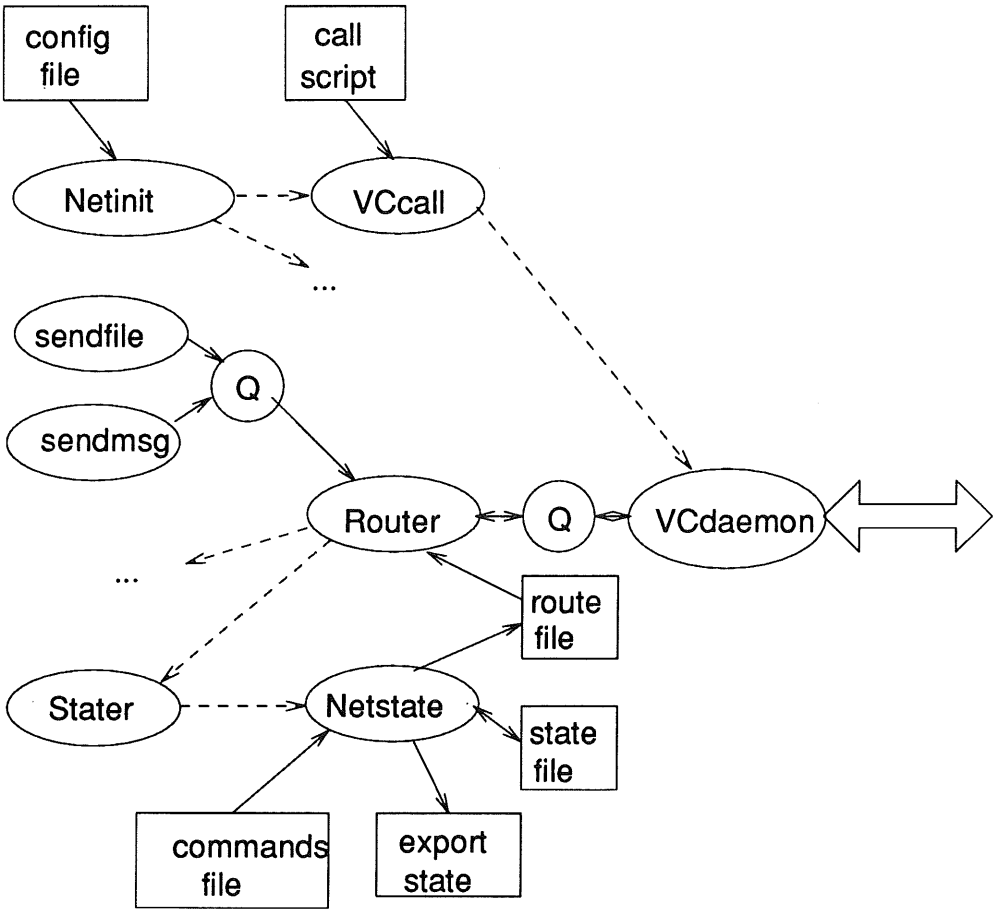
8. Higher Level Protocol Handlers

Higher level protocol handlers that are functionally identical to those used with SUN III are included in SUN IV. In addition a handler will be available that implements the X400 message handling system protocols.

9. System Structure

The system is constructed in three layers: a message handling layer, a routing layer and node to node transport layer. Standing beside this are management programs that control the execution of the programs that implement these layers.

The following diagram shows the major programs and files that make up the system.



At the message protocol layer there are programs that send messages (sendfile, sendmsg). These programs may be invoked by user agent programs for services such as electronic mail. The system doesn't include an electronic mail program but is easily interfaced to all the popular UNIX mail programs.

Also at the message protocol layer are the handler programs that receive messages. These may pass the message to user agent programs such as a mail delivery system or process the message completely such as the simple directory service.

At the routing layer the "router" program receives messages for transmission or delivery and decides on the basis of routing tables how to dispose of the message.

At the node to node transport layer a daemon process transfers the message to another host.

The management programs that control the system include "netinit" for invoking network programs at particular times, VCcall for establishing connections with another host, a message handler for network state messages and "netstate", a program to maintain routing tables.

10. Conclusion

The SUN IV system provides a number of major improvements over the previous version of the software while still maintaining compatibility.

NeWS Programming

Tim Long
Information Concepts

ABSTRACT

NeWS is Sun Microsystems' PostScript based network windowing system. It is intended to provide interactive graphics with device independent, efficient use of network bandwidth and ease of programming. NeWS uses a superset of PostScript which includes event handling, lightweight processes and multiple drawing surfaces. This talk will describe some aspects of the theory and practice of NeWS programming.

NeWS is a trademark of SUN Microsystems
PostScript is a trademark Adobe Inc.

The X window system - an overview

Tim Segall

Hewlett-Packard Australian Software Operation

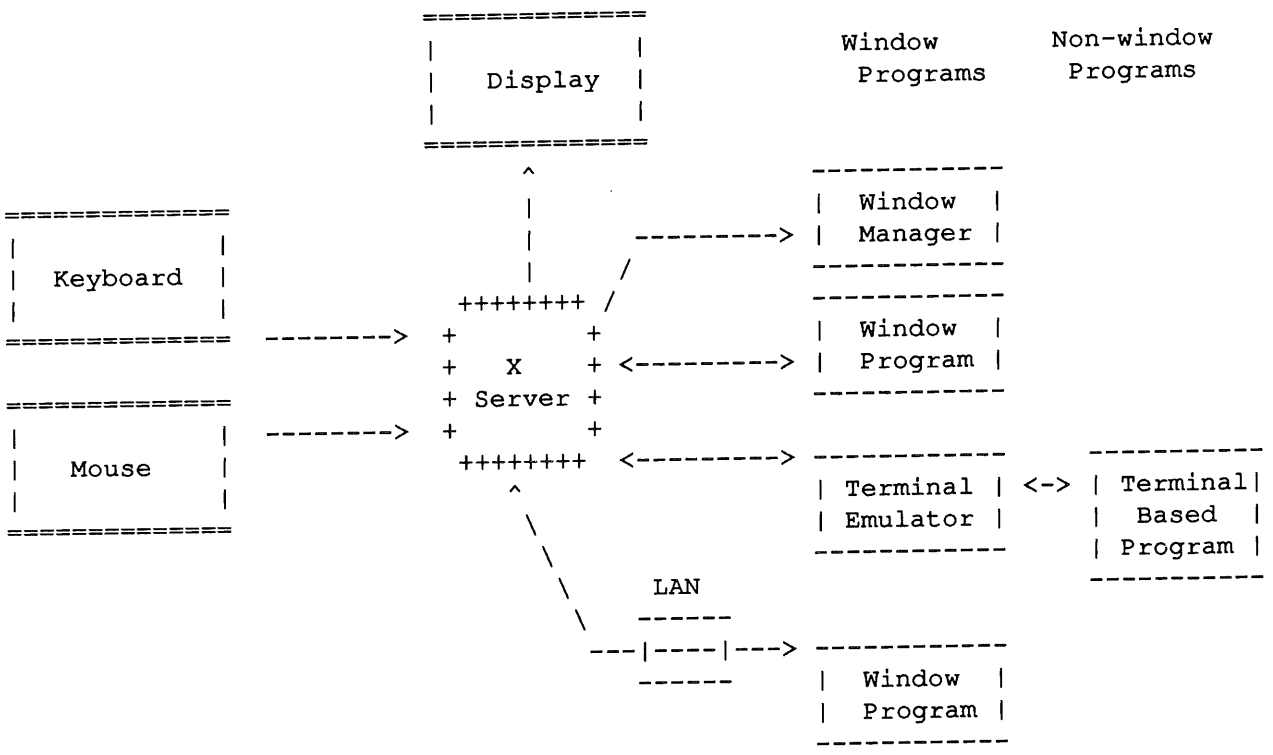
X is a network transparent windowing system developed at MIT. The X display servers are designed to be used with high resolution bitmap terminals. The server distributes user input to, and accepts output requests from various client programs. These client programs can be either local (on the same machine) or remote (somewhere out on the ethernet cable).

X supports multiple overlapping windows, fully recursive subwindows, text and graphics operations within windows. X attempts to provide hooks to 'bolt in' your favourite style of user interface. This is done by allowing you to specify your own 'window manager' client. A including moving, resizing, and iconifying.

The software for X is freely available from MIT, this has encouraged the use and creation of a host of powerful tools. These client programs include terminal emulators, bitmap editors, load monitors, clocks, calculators, games, ...

The X library is the low-level interface to the X server protocol. This interface is documented in the 'Xlib-C Language X interface' manual from the MIT Project Athena and consists of more than 150 subroutines. This library gives complete access to all of the capabilities provided by the X window system, and is intended to be the basis for other high level libraries. In this talk I will explain the fundamental concepts behind this low-level interface and talk briefly about a number of the higher lever interfaces.

The world from the X perspective ....



Hows a windowing system

Greg Rose

Softway Pty Limited

ABSTRACT

Ever wondered how to use a windowing system? These new windowing systems are far too complicated for common-or-garden applications at the moment.

Introducing "Hows" - the Household Old Windowing System(tm). This talk will describe in graphic detail just how windows and computers can interact in the household and every day use of older computers.

The Client/Server model will be examined in some detail along with the security implications of windows.

FTAM and OSI standards are hard to implement from scratch

by

Andrew Worsley

CSIRO Division of Information Technology

Abstract

OSI or "Open Systems Interconnection" will be the international standardised method of communication between computer systems once their definition has been completed. The basic OSI standards are reaching a state where they can be implemented. Yet amongst most people who program computers there is a poor understanding of the standards. The major reason for this lack of knowledge about OSI communication standards is the great effort required to learn enough about the OSI standards to implement them. The standards are hard to understand for those not specialising in OSI and often important parts are incomplete As most computer companies and governments intend to make major use of the OSI protocols for computer communications there will be a great demand for OSI expertise. The article is drawn from the experience of implementing a part of the OSI standards, called FTAM. The work was carried out to gain some expertise in the implementation of OSI standards. This article reports the problems and solutions we found during out implementation of OSI FTAM standard. We also compare the efficiency of the OSI FTAM implementation against a program, ftp, that provides the same service between BSD Unix systems.

1. Introduction

OSI stands for Open Systems Interconnection, which is a standardised method of exchanging data between computer systems, which is still being defined.[1] Because it will be an international standard it will be available to anyone and not restricted a few computer manufacturers or licenced organisations. The hope is that all computers will use these standards as the basis for communication between computers and so will be able to communicate with any other computer, regardless of its manufacturer or operating system.

In 1986 many of the OSI standards were reaching a state of definition suitable for implementation of a prototype version. Consequently CSIRO Division of Information Technology decided to support an implementation of the FTAM standard and some of the other OSI standards necessary to support it. It wanted to develop some expertise in this area in order to help Australia take advantage of the standards as they became complete.

We expected that an efficient and reliable implementation of the OSI standards could be produced from the standards documents alone with out any major problems. Since so many organisations were committed to OSI we expected to find other implementations to test against easily. This would give us confidence that the implementation conformed to the OSI protocols.

2. Methods and Protocol Standards used

2.1. What do FTAM and OSI look like?

The OSI standards are structured as seven layers, one on top of the other. Data passes from the user of OSI at the top of this stack down through each layer to the bottom layer, which is called the physical layer, where it is transmitted from the computer. The seventh, or top layer, has a different standard for each application provided. The layers one to six below this always provide the same services and are always present no matter what application is used. Some of the applications services that standards are being defined for are file transfer called FTAM, mail called MOTIS, remote login called Virtual Terminal and remote job control called JTM. All these application services run over the same bottom six layers.

The application standard FTAM was chosen for implementation as it was the best defined standard at that stage. [2] The name, FTAM, stands for "File Transfer and Manipulation" and it provides the ability to connect to a remote system and transfer files to and from the remote system.

The definitions of OSI standards are published from time to time in Australia by Standards Australia or SA, which until very recently was the SAA or Standards Association of Australia. These standards are available as a collection of about ninety two documents from the SA for roughly $1500. Each standard is the result of a long process of proposals, discussion leading to modifications and eventually agreement by interested people. People interested participate through meetings organised by what ever organisation represents ISO, which means International Standards Organisation in English, (not OSI) in their country. During this process the standard passes through several stages, each closer to final agreement of all the people involved. The final stage is called IS or International Standard and is the finished standard and will not be changed except by the publication of defect reports to fix problems as they are discovered. The stage before a standard is IS is DIS or Draft International Standard, which is supposedly stable enough to base a prototype implementation on. Although the implementation will need to be revised to conform to the IS version of the standard it is not meant to be a major change.

The implementation of FTAM we developed, which we call CSIRO-MU FTAM, is based on DIS FTAM standard which reached that stage two years ago. As this is being written it is still not an IS and the latest proposals contain very substantial changes from DIS to IS. These changes will be incorporated in the program which will probably take a couple of months.

As CSIRO D.I.T. is a contributor to the standardisation process it is supplied by the SA with the proposals, modifications, and results of the various meetings during the standardisation process. Receiving all the modifications enables us to keep track of changes to the standards and so to keep up date with current state of each standard. The implementation of FTAM was based on this material and another document published by the National Bureau of Standards in the United States. This work is the "Implementation Agreements for Open Systems Interconnection Protocols" and represents the agreements of many Implementors of OSI standards about how to deal with the deficiencies of the standards.[3]

2.2. Equipment/Operating system requirements

The implementation is written in C and runs on 4.2 BSD Unix systems.[4] The implementation consists of FTAM and the layers down to and including a class 0 version of transport, which is layer number 4. Below this layer it expects to run over a network layer, which is layer number 3, provided by the operating system.

At the moment the implementation has two interfaces to the network layer. The first is not a proper OSI network layer but mimics one using TCP/IP. To let people test OSI standards over ARPAnet, a TCP/IP network, a standard way of providing transport over TCP/IP was defined in Request for Comments number 1006 (RFC1006). Since TCP/IP is widely available this is a useful substitute until a proper OSI network layer is available on LANs. The second network interface is to X.25 which can be used as a proper OSI network layer. The TCP/IP interface uses standard Berkeley sockets whereas the X.25 interface uses Sunlink X.25, which is only available on the Sun computers.[5] Porting the implementation to use another X.25 interface should not be a major problem as this code is only a small part of the program.

2.3. How was the design work done

First the problem was divided up into as many separate design tasks as practical. Then these tasks were then implemented by a bottom up design strategy. Every attempt was made to make them as independent as possible from each other. Thus the design method was a mixture of top down and bottom up, working from both ends towards the middle. The parts of the standards that were unclear or not defined were left until they were understood, some time later, or left null if their properties are still not defined.

2.4. Design

First it was split into one design task for each layer. For example the implementation of the transport layer, layer number 4 was a single task and similarly for each of the other layers. Then each layer was divided into two parts. The first part handled the assembling and parsing of the layers protocol data units. Protocol data units or PDUs, as they are known in OSI standards, are the pieces of data exchanged by a layer with its corresponding layer

present in the remote machine. The second part handled the interaction with the layers above and below it. This involved checking the correctness of requests and responses it received from other layers, managing the layers state information and ensuring the required actions of that layer are carried out.

The protocol data for the top two layers, presentation and FTAM, all use the same format, known as ASN.1, for specifying their protocol data. So a single module was developed to assemble/parse ASN.1 for both layers. Each ASN.1 protocol data unit is translated into a table which represents the PDU in the program. To encode a particular PDU the encoding routine is passed a pointer to the PDU's table and the parameters for it, and from this it produces the encoded PDU. Likewise there is a decoding routine which takes a pointer to the PDU's table and its encoded form and decodes it into its parameters.

3. Results and Discussion

Writing the program to implement the OSI standards was not the major task. The hardest part turned out to be understanding the standards and determining what to do about the parts that couldn't be understood. This difficulty in understanding all parts of the standards made the task of verifying the implementation was correct even more important. This was not easy because there is currently only one implementation available to test against. The other major activity was making the implementation efficient which required a fair amount of work.

3.1. Standards

The OSI standards are written in a very dry and formal manner. They consist mainly of definitions with very little, if any, description of the meaning or purpose of the constructs they define. There are many references to other standards for definitions of important objects which means that the reader usually has to read several other standards as well to understand things. Sometimes these other standards are not written or just not available. The standards define the parameters, the services and the format of the protocol data but do not put this information in any context by explaining what its purpose is. This makes it very hard material to understand. The reader must put all the pieces together and deduce what the standard is trying to do by himself. Consequently the comprehension of a standard *takes much longer than simple reading it through a few times.* Since there are many standards that need to be read, nineteen for the work described here not counting X.25, this represents is a large investment in learning required by the implementor.

The problem of standards being changed after they were implemented was not a great inconvenience. For example, to implement the changes between DIS presentation and ACSE standards and the IS versions amounted to one months work. The transition from DIS FTAM to IS FTAM will probably be two or three months of fairly straight forward work, as there are a lot of important changes between the two versions.

What helped a great deal with the difficult problem of ambiguous or unspecified parts of the standards, such as addresses, object identifiers and document types, was the efforts of other implementors. In particular the "implementation Agreements for OSI protocols" was a great help. It is published by the NBS, or National Bureau of Standards, in the United States and contains a recommendation of how to handle problems in many OSI standards. It is the result of the efforts of the people who attend the OSI implementors workshop.

The NBS OSI implementors agreements contain a list of known deficiences in the standards and proposed changes to fix them as well as other agreements on problems not addressed by the standards. For example, when you connect to a remote machine to pick up some files, a very useful facility is obtain a list of the files available on the remote machine for transfer. As the FTAM standard stands now it does not define a way of doing this. Even if you provided a method of doing so in your implementation, there is no way to make sure it will work with other implementations. The NBS agreement defines a file type which roughly corresponds to a Unix directory. Thus as long as both the implementations support this special file type, the user can get a list of files available for transfer. Since this is an agreement of other implementors it is likely that this useful facility will be widely supported among FTAM implementations.

There are other implementation agreements published by groups such as SPAG[6] in Europe but the NBS is the more useful. The two groups, SPAG and NBS, have stated they will try to remain compatible and incorporate each others proposals so implementations can be compatible with both. This problem will become less severe as time goes by and the standards are fully defined but even when they are completed there will still be problems, like the file listing one mentioned above, that need to be addressed.

3.2. Design

To achieve an efficient implementation of OSI standards requires a lot of work. The CSIRO-MU FTAM implementation consists of roughly 32,000 lines of C code, occupying about 690 Kilobytes of space. To give a feel for how space efficient the implementation is it is compared against some other file transfer programs in table 1. ISODE [7] is a development environment for OSI protocols which is freely available for a small distribution fee. It contains an implementation of DIS FTAM we will call ISODE FTAM. The first entry in table 1 is ISODE's ftam which uses a straight forward but lengthy encoding and decoding method. This yields a very large implementation with a binary of over three times larger than the CSIRO-MU ftam.

In the CSIRO-MU ftam a lot of effort was put into making it space efficient. In particular the encoding and decoding part was done by providing a table for each PDU. This table could be stored very compactly, yielding a much more compact method of encoding and decoding. In contrast ISODE has a separate function for encoding and decoding each PDU. While this is not the reason for all the size difference between the implementations it is probably the most important one. ISODE has not put much importance on space efficiency in general.

Size of various file transfer program binaries compiled under SunOS 3.5 for Motorola 68000 cpu				
program	user program		server program	
	bytes	percentage of ISODE	bytes	percentage of ISODE
ISODE ftam	727976	100	702292	100
CSIRO-MU ftam	237104	32	231296	32
ftp	97748	13	102032	14

Table 1.

CSIRO-MU ftam is twice the size of the Unix **ftp**[8] command, which provides the same service as the FTAM implementation for Unix machines. Ftp is so much smaller because it is a much simpler and straight forward protocol. In contrast FTAM has a very general protocol and requires several other layers of additional protocol below it. For other applications we expect that supporting the OSI standards means a large overhead in providing all the OSI layers below the existing one and a more general application protocol. In table 2 this OSI protocol implementation is listed as OSI ftam service and in CSIRO-MU FTAM amounted to as much as 140k.

Break down of size of CSIRO-MU ftam program binaries compiled under SunOS 3.5 for Motorola 68000 cpu				
component	user program			
	bytes	percentage of total	bytes	percentage of total
OSI ftam service	143464	60	143464	62
interface to ftam service	19104	8	25504	11
library routines	74536	31	62238	26
total	237104	100	231296	100

Table 2.

The size of the OSI implementations illustrates one of the main weaknesses of the OSI standards. The amount of program code needed to support the protocol is much greater than necessary to do the job. For why would anybody want to use a less efficient communication protocol than they are using at present. The ability to communicate in a standard manner is the only advantage it can offer against this disadvantage. Thus use of OSI protocols leads to considerably larger programs than use of a simple protocol such as ftp uses. This raises the question of the best way to handle this large OSI overhead. Much of it is common to all OSI implementations and consequently could be placed in the kernel or in a memory resident shared library. These areas will need further work to determine which are the best approaches.

To make the implementation efficient in its speed of operation requires a lot of work because of the number of layers that the data must pass through. An FTAM implementation can be have the same order of speed as ftp, if it is

implemented so there is little or no copying of data. This is because during a file transfer most of the data is passed through each OSI layer untouched. If it is passed between layers by pointers, instead of copying. it data can pass quickly through a layer. The layers still have to do some processing of the data, adding or removing headers, so it is not a trivial action. As it passes down a layer that layer adds it header to the data and on the way up a layer it removes its header from the data. This additional work on the headers cannot be avoided but in practice is not a significant delay with large transfers of data. This is because data takes nearly a constant time to pass through a layer, rather than a time proportional to the amount of data, so the larger the piece of data passed the less the delay per byte transmitted.

File transfer throughputs in Kb/s (Kilo bytes per second) for various programs				
Program	direction of transfer (command)			
	from remote machine (get)		to remote machine (put)	
	Kilobytes/second	percentage of ftp	Kilobytes/second	percentage of ftp
CSIRO-MU FTAM	117.±3	73	142.±3	83
ISODE FTAM	122.±10	76	118.±5	69
ftp in binary mode (SunOS 3.5)	160.±10	100	170.± 10	100

Table 3.

From table 3 we can see that both FTAM implementations achieve a transfer rate of as much as two thirds of ftp transfer rate. In the implementation of ISODE a lot of effort went into its data structures to avoid unnecessary copying of the data. Likewise CSIRO-MU FTAM uses special data structures and routines to minimise copying of the data. At the moment CSIRO-MU ftam still spends 10% of its time copying data, future work may eliminate this unnecessary copying and possibly increase the transfer rate to something close to ftp's transfer rate.

The decision to separate each layer into the state part and the encoding part greatly simplified the implementation in practice. It made software much easier to maintain and change, as was necessary when adding in the changes between the DIS and IS versions of some of the standards. Splitting the ASN.1 encoding and decoding part into a special section of its own worked out very well in practice since it meant there was one common set of encoding/decoding routines for the top two layers. The fact that other implementations, e.g. ISODE, also provide special help with ASN.1 encoding/decoding supports our approach. In fact further space would have been saved in the other layers if they too used ASN.1 for their PDUs instead of their own layer specific methods.

Starting from scratch requires you to understand and design everything, which is a lot more work than taking taking an existing OSI system and extending it. The first part of this extra work is designing how data and parameters are passed which is not defined by the standards in any way. It was important to make this very flexible and fast so as not to limit the implementation in any way. The other extra work is providing the many subsidiary services, which are not very well defined by the standards but none the less important. Mostly this is the OSI directory services which look up information in some data base. For example it needs to take a machine name and look it up to produce an OSI address for that machine. Similarly for a particular service on the machine an application entity title needs to be looked up. As directory services are still being defined and have a large standard themselves a very simple look-up mechanism used which scanned a standard file for the data. The format of the file was based on that used by ISODE so only one database is needed for both implementations. Also routines to manage object identifiers of various things such as document types and abstract syntaxes and so on was provided. By extending an existing implementation instead of starting from scratch most of this work can be avoided.

3.3. Testing

Testing such a large program is a substantial task. The first problem is that the data it generates is not readable. So it is essential that there be some way of displaying its data in a readable form. This is especially important when it fails while trying to interwork with another implementation. Then it is necessary to see what it is transmitted at each layer to determine where things are going wrong. Further more it is useful to have some tracing facility to watch various parts of the implementation perform, so as to quickly isolate a faulty part or verify that a new piece of

code is working, amongst the enormous amount of other code present.

CSIRO-MU FTAM was tested against itself which is a good basic test that the code runs. But this is far from a good check that it is following the standards. There are often faults where both sides of a connection will be broken in the same manner and so this will not test not discover the problem. These faults can occur because of the standards were wrongly uninterpreted or simply because you using the same piece of code with the same problem for both sides of the connection. Interworking against an independently written implementation is a much more stringent test. CSIRO-MU FTAM was tested against ISODE's FTAM. This turned out to be very beneficial for both implementations as it did discover several problems in both implementations which have since been fixed.

Even if both implementations do implement the same standard it does not guarantee that they will interwork. The first attempt to interwork the ftam implementations failed because each ftam had implemented a different and incompatible service class. This problem of incompatible subsets of a standards is another trouble area of many OSI standards not just FTAM. Fortunately this is now well addressed by the implementors' agreements. Still these agreements cannot guarantee interworking unless their recommendations are implemented which is not mandatory. A better guarantee of interworking is to implement all possible options of the standard, though this can be prohibitively time consuming to do. The CSIRO-MU FTAM implements all service class options now.

At the moment there is a dirth of OSI implementations even though all the major computer manufacturers have committed themselves to supplying OSI products. This may be because they do not want to release any intermediate versions of the products and are waiting for the standards to be completed. Unfortunately this makes it harder to test implementations and leads to a low level of awareness as only a few academic sites and even fewer commercial sites do work on OSI standards in the public arena. Prehaps this problem will only solved when the final version of the standards becomes available. Unfortunately it is then much harder for their problems to be fixed.

3.4. Conclusion

Although OSI FTAM standard was difficult to implement from scratch, it was achieved and the implementation is reasonably efficient. As another implementation was available it was possible to test the two implementations against each other and so gain confidence that it adheres to the standard.

Gaining an understanding of the standards was both difficult and time consuming. The major difficulty with understanding the standards is the form they are written in. The style of the standards does not give the purpose of the things it defines, making very hard to understand follow. Other problems are the number of standards the implementor is required to understand and the fact that many important parts of the standards are still being defined.

The NBS workshop implementors agreements is a great help to implementors. Then documents such as these do not give an understanding of standards. Another very useful source of help is other implementations, which the implementor can use as a basis for his own work. This allows the implementor to concentrate on his own area of interest with out having to learn or implement any of the other parts which represent a considerable saving of effort.

It is difficult to develop computer communications conforming to the OSI standards but is possible. As the standards stand now they will probably never be as efficient as simple and single purpose protocols but with a lot of effort they can be made to within a factor of two or so of the efficiency simpler protocols. OSI standards will provide a very useful connection between all computer systems but the efficiency issues may constrain the extent to which they replace existing methods.

4. References:

1. A. S. Tanenbaum, *Computer Networks: Towards Distributed Processing Systems*, Prentice-Hall, Englewood Cliffs, NJ (1981). This gives a good introduction to OSI and networking in general, except it is old. A new and very much more up to date version is expected in the next six months.

2. ISO/OSI - ISO/TC 97/SC 21/WG 5, *ISO/DIS 8571 parts 1-4 - File Transfer, Access and Management*, Standards Australia, Canberra (July, 1986). Standards documents that defines DIS FTAM. It gives references to other standards that it uses, which reference others ....

3. Robert Rosenthal, Editor, *Implementation Agreements for Open Systems Interconnection Protocols*, National Bureau of Standards, U.S. Department of Commerce, Gaithersburg, MD 20899 (July, 1987). This document records current agreements on implementation details of OSI protcols among the organisations participating in the NBS/OSI Workshop Series for Implementors of OSI Protocols. These decisions are documented to facilitate organisations in their understanding of the status of the agreements.

4. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).

5. Stuart Sechrest, "An introductory 4.3BSD Interprocess Communication Tutorial," in *Unix User's Manual - Supplementary Documents*, Usenix Association (April, 1986). This gives an introduction to the programming interface in the BSD TCP/IP network. It is found in the Unix Manuals for the BSD version of Unix as published by the Usenix Association in the United States.

6. Standards Promotion and Application Group (S.P.A.G.), *Guide to the use of Standards*, ELSEVIER SCIENCE PUBLISHERS B.V. (November, 1986).

7. Marshall T. Rose, *The ISO Development Enviroment at NRTC*,.ie !"ISODE is an OSI development enviroment for Unix systems which is distributed for a small cost. It contains a DIS FTAM implementation as well as many other useful pieces of software. It is essential material for any implementor of an OSI standard as it provides useable implementations of much of OSI. Available in Australia, for 150 Australian dollars, as part of the ISODE distribution from George Michaelson, CSIRO, D.I.T., 55 Barry St."" ISODE is an OSI development enviroment for Unix systems which is distributed for a small cost. It contains a DIS FTAM implementation as well as many other useful pieces of software. It is essential material for any implementor of an OSI standard as it provides useable implementations of much of OSI. Available in Australia, for 150 Australian dollars, as part of the ISODE distribution from George Michaelson, CSIRO, D.I.T., 55 Barry St.

8. Berkeley Unix User's Manual, "ftp (1c) - ARPANET file transfer program," in *Unix Programmer's Manual - Reference Guide*, Usenix Association (April, 1986). This describes the ftp command which allows the transfer of files across TCP/IP networks. It is found in the Unix Manuals for the BSD version of Unix as published by the Usenix Association in the United States.

ABI and OSF: Technology and Future Impact on UNIX

Ross Bott
Pyramid Technology Corporation

Synopsis

The AT&T/SUN UNIX Applications Binary Interface (ABI) and the new UNIX proposed by the Open Systens Foundation (OSF) can be viewed from at least two perspectives:

(a) A political level involving jockeying among the major computer manufacturers to best take advantage of what has now become a very hot item -- UNIX.

(b) A technical level -- both approaches have significant and possibly surprising, effects on applications developers and UNIX users.

This presentation will concentrate primarily upon technical analysis of the OSF and ABI issues and how they impact the UNIX community. It will conclude with a perspective of how these issues relate to those of XOPEN and POSIX.

An Outline of the Presentation follows:

Applications Binary Interface

(1) A technical definition.

(2) The Politics of ABI: Why do AT&T/SUN feel that it is necessary? Why is it viewed as a threat to other manufacturers? (Relationship to 286/MS-DOS and 370/VM.)

(3) Implications to the computer manufacturer. Are multiple ABI's meaningful? What does ABI mean to a computer manufacturer who isn't one?

(4) Implications to applications developers. Does ABI mean that they will need to support only one binary?

(5) UNIX vs ABI: A consideration of the issues of code portability, porting "distance", long term applications support.

(6) Long term implications of ABI.

Open Systems Foundation

(1) A definition of the OSF UNIX charter.

(2) The Politics of OSF: Definition and rights of supporting vs affiliate members. How is OSF a counter to ABI? Why is IBM invovled? Can OSF and ABI coexist?

(3) Technical evolution of OSF UNIX into the 1990's. Is OSF dependent on AT&T UNIX? The expansion of UNIX capabilities: databases, OSI, office automation, etc. Effect of OSF on computer manufacturers.

(4) Effect of OSF UNIX on applications developers: Software is a commodity. Evolutionary directions of future software companies.

(5) Effect of OSF on UNIX users.

X-Open vs POSIX vs OSF vs ABI

(1) Placing these efforts within a single technical framework.

(2) Is "UNIX as a single operating system standard" a future tautology or contradiction?

Biography

Ross Bott, Vice President and General Manager, Advanced Architectures, joined Pyramid Technology in March, 1982, three months after the founding of Pyramid. Previous roles at Pyramid included operating system designer, manager of the group which designed Pyramids dualPort(TM) OSx, manager of the Networking and Communications Group, Director of Software, and Vice President, International Marketing, before moving to head up Pyramid's advanced technology unit.

Prior to Pyramid, he spent four years at Xerox Palo Alto Research Centre, where he was part of the team which designed Xerox's STAR workstation, the precursor to Apple's Macintosh and Lisa, and other similar personal workstations, and did theorectical work in the representation of knowledge and other areas of artificial intelligence. Ross has a BS in Mathematics from Standford University and a Ph.D. from the University of California, San Diego in Artificial Intelligence.

Distributed Trouble:
The University of Sydney experience with networked workstations.

Winner of AUUG prize for best paper submitted by a full time student

Rex di Bona
rex@cluster.cs.su.oz

Workstation UNIX† systems are in vogue today. To have a collection of nearly independent workstations as your computer system raises problems in many areas. Two of the more important problems raised are software consistency and system security. This paper will discuss the issues of software consistency and system security as found from experience with installing, maintaining, and dealing with a network of thirty machines, each locked in an inaccessible office, which has been set up at the Department of Computer Science at the University of Sydney.

Introduction

The Department of Computer Science at the University of Sydney has installed a system of thirty workstations. This system has been setup to provide computing resources for the staff and postgraduate students, replacing the Vaxen currently used for research.

One of the major goals of the installation was to provide a secure, distributed computing environment. It was hoped that the system would provide computing resources that presented the same user interface that was available on the departments' Vaxen. This decision required that each workstation provide an identical view of the filesystem, so that the machine that was actually being used was irrelevant.

During the setup, maintaining, and upgrading of the system, problems were found with both the software maintenance facilities, and the system security. After installing a new program, or update, on only one machine previously it was found that having to perform the installation or update on many machines was more difficult. This problem was exacerbated by the placement of each workstation. The workstations were distributed to staff members' rooms, and so were inaccessible if the staff member was absent and the room locked. If a workstation was not communicating with the rest of the system, through being powered down, crashed, or for some other reason, then that workstation could not be attended to until the staff member returned. Any installations, or updates that were scheduled to be done to that machine had to be delayed until the machine was again available.

Network Architectures.

There are three different methods of connecting up a collection of machines. The difference between the connection strategies is the user interface. The simplest, and most secure system, is the minimally connected system; each workstation, each server, and any other connected machine is totally independent. The servers provide files for the diskless machines. Each machine has an independent filesystem; no files are shared and no file is visible from multiple machines. In this system a user

† *UNIX is a Trademark of Bell Laboratories.*

would have accounts on just those machines for which login permission has been granted.

In the minimally connected system, users are allowed to have login sessions between machines. They would connect using a remote login program, such as *rlogin*, but no other method of remote access is allowed. In effect the network is treated as a collection of independent machines between which terminal connections are provided. The fact that some machines are diskless and that the login interconnections are done over an ethernet is totally transparent to the user.

A more interconnected system allows for the sharing of files; a file can be seen from multiple machines. This network architecture has possible security problems which are discussed below. The advantages of this network architecture includes allowing a user to access a file from several machines, alleviating the task of keeping the file consistent. Another advantage is that since duplication of files is reduced the amount of free disk space over the network is increased.

Finally, systems can allow more than just file sharing, they allow users to see all of their files from any machine, thus allowing easy execution of jobs on any machine. This ease of access creates a situation where the processors within machines are part of a liquid resource. It does not matter on which machine a job is executed so the load should be spread over the entire network, resulting in improved performance for all machines. A user would execute a job on a machine not currently heavily loaded.

Of these three architectures the third is the most preferred as it provides the greatest flexibility for the user. Any workstation is an entry point into the network and the view of the network is identical from each workstation. This is especially useful for people who do not have a set workstation assigned, but who use a random workstation from a pool of workstations.

Software Consistency.

Software consistency is a problem on a network of heterogeneous machines. The University of Sydney has slight variations in the set up of each of the workstations it is running. Nonetheless it is important that the user environment remain identical, or nearly identical on each machine; hence differences are hidden from the ordinary user. The configuration at the University of Sydney consists of thirty Sun Microsystems workstations, half with, and half without local disks. Three servers provide files for all workstations, and root and swap filesystems for the diskless machines.

Problems arise when software has to be updated on the filesystems that are not globally shared. Because of the desire to have a homogeneous user interface on the heterogeneous network each modification must be tailored for six different system configurations. The current system configurations are: 3/50 with disk, 3/50 without disk, 3/60 with disk and the three different servers, each server differs in both disk capacity and in which filesystems it exports. In its simplest form a software update has to be installed on all six different system configurations. It is possible that machines in each configuration are not available for the update when it is installed.

The filesystems on the system are replicated for several reasons. Firstly it reduces the number of concurrent accesses that are occurring at any server. Secondly it was hoped that in the event of a server becoming unavailable for some reason, foremostly in case of a machine crash, that some clients would be able to provide at least a partial service. Some filesystems, most notably the root, or '/' filesystem, has to be unique for each machine, as machine unique information is stored there.

Local versions of common programs also have to be placed onto new machines and must be updated during operating system upgrades, a problem we encountered upgrading from SunOS 3.2 to SunOS 3.5 recently, where several machines did not have the local shell installed causing spurious behaviour. Attempts to install new versions of existing software can also have this problem of consistency.

In an effort to keep track of which updates had been done on which machines an automated system was developed to do the updating. A group of people, the system administrators, have permission to run the updating program which executed a specified command on a specified group of machines. The group of machines is specified by type: whether each machine has a local disk, each is a server, each machine does not have a local disk etc. If a command could not be successfully executed on a particular

machine the machine name, along with the command, is appended to a system wide file. Upon booting each machine checks to see if any updates are outstanding for that machine, if so the updates are performed where possible.

The Severity of Security Breaches.

A security breach can take one of several forms, each having a different severity. The least severe breach of security is being able to read the video screen of another user. By reading the screen it is possible to examine files that are being examined by the screen user. More severe still is being able to modify the screen of another user. If a users' video screen can be read and modified by a non-privileged user then important information can be obscured or changed, such as console messages.

Still more severe than reading and/or modifying a video screen is reading and modifying a file that belongs to another user. To read and modify files is an active break in whereas reading the video screen is a passive break in. The system breaker is in control during an active break in, whilst during a passive break in the breaker can only benefit from the targets' actions. During a passive break in the system breaker can only observe what the target has executed, or what the target is examining. A common example of a passive break in is incorrectly set paths for shells, where a user thinks that a standard utility is being executed, but instead a program of the same name in the current directory is being executed. The system breaker has placed a command with the same name as a common system utility in a directory in the hope that the superuser will execute the command. The definitive example of this is executing the command *su* instead of giving the full pathname of */bin/su* or */etc/su* depending upon su's location.

The most severe action that can occur is to impersonate another user; to be able to execute commands and have the kernel believe that a different user was executing them. This is done by either pushing characters into another users' terminal input stream, or by corrupting the uid of one of the system breakers' shells.

Network Security.

A computer system is physically secure if no physical action by a user can modify the systems behaviour. In general the best physical security was obtained by having the computer elsewhere; what a person cannot get to a person cannot change. With the advent of workstations the user now has physical access to the machine and, more importantly, to the console of the machine. A user may power down the workstation or the user may use the console escape sequence to restart the operating system. A final possibility is that the user can use the power of the console to modify the operating kernel and gain privileges in that way. Several different methods of attack that have been discovered using the console will be presented, along with possible solutions.

Under UNIX, privileges are arbitrated by the use of a machine wide name space of unique numerical identifiers, or user identifiers (*uids*). The protection schemes in UNIX depend upon these user identifiers. On a network of UNIX machines these name spaces might not be identical. Entries in these name spaces might not be identical either, a person on one machine having user identifier 1355, may not be the same person having user identifier 1355 on another machine, even though the names of the people might also be the same.

Within a kernel running as a single unit, a kernel that has only one name space, user identifiers are secure. A routine within the kernel can trust, implicitly, that a user identifier that it has been passed is correct. On a network of associated kernels this implicit assumption is no longer valid. It is possible that user identifiers passed to a kernel are no longer correct. A kernel must verify somehow that the user identifier it has been passed is one that has privileges within the local system. The source of all verification is the password provided by a user on initial connection to the system. Either this password can be passed with each remote call, or the receiving kernel can do a verification check with the sending kernel. Both of these methods is fraught with danger.

A machine on a network can impersonate another machine if there are no attempts to trap this sort of fraud. The main method used to trap this sort of invasion is through the use of encrypted data streams between corresponding machines. The use of encryption removes the usefulness of tapping into the data stream's media. It does not stop people falsifying data on their workstation *before* it is passed to the encryption routines. This possibility of corruption *within* a trusted workstation is the hardest to discover.

Security Holes.

There are two steps that a system breaker must do to compromise the security of a network of machines. The first is to compromise a single machine. The second step is to compromise machines connected to the first machine. We will now examine methods that can be used to do both of these steps.

There are "traditional" methods used to break UNIX systems. These methods are concerned with badly set permissions on system files and directories, and badly written system programs. Other common methods are using insecure system features, the most common being TIOCSTI on Berkeley systems which allows a program to simulate input on another users' terminal. These methods have been covered in detail elsewhere, and so will not be elaborated here.

More interesting are the problems that arise when communicating between kernels. The problems of system identification; "is this really the machine that it says it is", and of user identification; "is this really that user?" Under current networking systems there is no method readily available to ensure correct identification.

We shall now concentrate on specific examples of security holes. These holes have been found in various versions of the SunOS operating system, versions 3.2, 3.5 and 4.0. They work irrespective of the usual safety features, set-uid over the network, and the promote root to nobody on NFS actions, unless otherwise mentioned. All of these security problems have been tested and verified. Specific examples are available from the author if desired.

Screen Security.

There is no screen security. The video screen on a workstation is very poorly protected. The video screen is accessed by system calls through the special device /dev/fb which is publically readable and writable.

The screen device has to be publically readable and writable as it is owned by root, and the ownership is not changed on login. Under non-windowing systems the terminal that is logged into has its ownership changed to that of the user logging in. The screen cannot be opened exclusively, as no single process does all the screen accesses. This is a consequence of the decision to have each process manage its own window in a windowing environment.

Since the video screen is, by default, going to be used by the person logging in on the console it would be sensible that the video screen should have its ownership changed at the same time. If this was done it does prohibit the use of the video screen by persons logged in remotely, and problems still occur in the case of multiple video screens, who owns which screens, when should the ownership change and how can having a process that keeps the screen open be prevented? A screen is not exactly like a terminal as the contents of the screen can still be read, there is a sense of history as previous output is still available.

Root Security.

The main aim of any general break in is to obtain superuser, or root, privileges. The only way to have superuser privileges is to have a process running that has a uid of 0. Several methods of doing this will now be discussed.

- Single User

The main problem of a workstation is that the user has physical access to the machine console. A user

can reboot the machine at any time. The ability to reboot at any time is given as a selling point for some workstations. It is well known that it is possible to boot a UNIX system single user, where the console is logged in as the superuser. This single user mode is usually used for system reconstruction after a system crash. There is nothing stopping a system breaker from halting the system and booting single user. This would promote the system breaker to superuser privileges without requesting the root password. The boot code for our system has been changed to disallow this for the workstations.

If the root filesystem requires attention so that the machine can be booted this attention can be provided in one of two ways, depending upon the type of workstation. If the faulty workstation is diskless then the root disk partition is mounted on the server itself (The workstation is powered down so there is no consistency problem) and the problem corrected. If the workstation has a local disk then the disk is connected to a diskless workstation and the filesystems mounted on the diskless machine and corrected. In neither of these cases is the single user boot facility required. The servers are kept in a secure room so they are allowed to be booted single user.

• Diagnostic Monitor

The Sun Microsystems workstation has a disgnostic monitor built into each machine. This monitor allows system administrators to boot the workstations from various devices, to check all devices that are present on a machine, and to check that each device is functioning correctly. Also included in the diagnostic monitor are commands that are useful in the debugging of stand alone programs. These commands include the ability to read, write, modify, breakpoint and trace programs running within the workstation.

The UNIX kernel is like any of these programs. It is a standalone program running that happens to run other programs. Using the diagnostic monitor it is possible to change both the data structures and code of the kernel.

• Kernel Data.

Kernel data can be changed by looking in the system include files which give the address of the kernel user data structures. The simplest method is to change the uid of all programs running from the breakers' uid to root. For example if the breaker had uid 1355 then all uids of 1355 would be changed to uid 0 in the kernel user area. All of the system breakers' programs would now be running with superuser privileges.

• Kernel Code.

Kernel code can be manipulated in exactly the same way. It is slightly harder to manipulate the code area as it is write protected, but monitor commands are available to change the memory protection bits. A small amount of experimentation will quickly enable a system breaker to discover the meaning of each protection bit.

The attack described below was done by a person slightly conversant with 68020 code (but armed with a 68020 assembly language manual), no intimate knowledge of the UNIX kernel, and no kernel sources. It resulted in crashing the workstation about a dozen times before the correct procedure was found.

The code to attack is slightly harder to discover. The ideal piece of code to break would allow an unprivileged user to be promoted to superuser easily and without complaining or recording the fact if auditing was enabled. The place to start the attack would be to examine the system calls allowed to users. This would reveal the *setreuid()* system call, which changes the uid of the invoking process. If this system call was to always succeed it would be ideal as it is not executed by system code in the expectation of failure; system code would not execute it to check if you are currently the superuser.

To determine the correct point of attack within the setreuid code the standard utility *adb* can be used to examine the running kernel from an assembly language level. Examining the code reveals several calls to the kernel routine *suser* which appears to be a check whether the current user is the superuser. Modifying the calls so that they succeed no matter what the return value does the trick. The call to *setreuid* on the modified kernel now succeeds no matter what the callers' uid is being changed to. This

is because the kernel believes that the superuser is doing the system call.

To utilise this hole, which is a very hard hole to detect, as the kernel itself is corrupted, the following 'C' program can be used:

```
main()
{
        setreuid(0,0);                  /* set uid and euid to 0 */
        execl("/bin/sh", "sh", 0); /* run a shell as root */
}
```

Figure 1. Setuid Root Program

The program is not elegant, but very functional. It provides a root shell, is not setuid and as such virtually undetectable.

Both of these attacks require that the breaker have an account on the target machine. The diagnostic monitor can also be used to log into a machine without a password, enabling the system breaker to use an account. It is possible to patch the *login* program so that the test for password correctness is ignored. This allows a system breaker to obtain the use of an account on a target machine. This is the hardest method to corrupt a system, as it requires extensive examination of assembly code in the pure data state, no symbolic instructions are provided, the instructions are not decoded, they are just a stream of bytes.

Dominion Expansion.

Once one trusted machine in a networked file system network is corrupted the system breaker may endeavour to expand the number of corrupted machines. This can be accomplished in two ways. Either each machine can be corrupted in a similar fashion to that used to corrupt the first machine, or the fact that the first machine is corrupted can be used to allow the breaker to affect other machines. There are differing methods available to do this.

• Setuid Programs.

If setuid permissions are allowed over the network then creating a setuid root shell is sufficient to corrupt any machine from which that program can be seen. All the breaker has to do is log onto the target machine and execute the setuid root shell which will promote the breaker to root privileges. This attack only works on systems where setuid is allowed over the network.

It is not necessary that the system breaker have an account on the target machine as the breaker can log in as bin or any other standard account. It is possible to log in as bin, even though bin has an impossible password, as the password is not checked on remote logins if the machine being logged in from is a trusted machine.

• Root Access.

If root is believed over the network then to corrupt a new machine becomes a simple task. The breaker has to create a setuid shell but on the target machine, not on the client machine. In other respects this attack is similar to that outlined above for setuid programs.

• Yellow Pages.

The yellow pages system is a distributed database system for delivering, among other things, the password file around a network of machines. The yellow pages system is unprotected in SunOS 3.2 and 3.5. In SunOS 4.0 this attack method has been corrected. The basis of the attack is that any user, not just the superuser can change which server the yellow pages client refers to for the password file. If the corrupted machine is set up as a yellow pages server then the target machine can be told that all password references are to be resolved by reference to the corrupted machine. If the password file served from the corrupted machine has a known password for a root id account then logging in as the root id

account gives the system breaker root privileges on the target machine.

• Root System Calls.

Corrupting a remote machine means creating a method by which root privileges are obtained. The breaker can patch the kernel on the remote machine if the breaker had access to the memory of the remote machine. The diagnostic monitor provides this access, but may not be available. UNIX provides a method for modifying, or at least looking at, system memory. This method is to open the special device /dev/mem which is an image of the physical memory of the workstation. Sensibly the permissions usually associated with this device prohibit ordinary users from writing to memory. The security hole is that root on a workstation is allowed to perform the *mknod* system call to create a special device, such as a physical memory image device *over* the network. This new memory image can be used to patch the kernel in the way outlined above, where *setreuid* is corrupted. The program given in Figure 1 then has to be executed on the remote machine and a second machine is now corrupted.

Mknod is not the only problem here, it typifies the problem of where checking is done over the network. Only the superuser is allowed to execute certain system calls, but the checking for whether the caller is the superuser or not is done on the client machine, not the host machine, so the host machine believes a corrupted client machine, whereas it is actually being given false information.

After a successful attack two machines are now corrupted. This process of corruption can be continued as long as there are filesystems that can be seen from both a corrupted machine, and an uncorrupted machine. In this way an entire network of connected machines can be corrupted from having one machine corrupted.

Conclusion.

The corruption of a network of workstations starts with the corruption of a single workstation. The first line of protection is to secure each individual workstation. The diagnostic monitor must be either removed, or greatly reduced in power, or require a password before use. Other known security holes should also be removed. Once a single workstation is corrupted the task of corrupting others is made easier.

It is shown that not allowing setuid and root access over the network does not increase security within the network currently so these protection devices can be discarded. Discarding these allows for a simpler interface to system programs such as the mail programs, as the need for daemons is removed. The problems with the yellow pages interface, and with the incorrect checking of uids on clients instead of servers, allow for easy penetration of supposedly secure systems. Since it was a trusted client that was corrupted the normal security measures of *rsh* and *rlogin* are circumvented. Encrypting the network traffic will also not restrict the prevention of these methods. The only of protection against these attack methods is to do more checking on the server.

In all cases the protection outlined is not a guarantee that a system is secure. Many bugs and security holes may exist similar to those presented. The main problem is with trying to provide such a general service as NFS without careful forethought to security issues. Patching a known insecure system will not guarantee a secure system, a rethink on the security aspects of the NFS/RPC system should be undertaken. The kernels on each of the machines should not be individual entities, but should be linked closer.

INSTALLING AND OPERATING NFS ON A 4.2 BSD VAX

Jim Reid

Department of Computer Science,
Strathclyde University,
Glasgow,
Scotland,
G1 1XH.

jim@cs.strath.ac.uk
jim@strath-cs.uucp
...!uunet!mcvax!ukc!strath-cs!jim

1. Introduction

The author's experiences in the installation and operation of Sun's Network File System (NFS) on a VAX 11/750 running 4.2 BSD UNIX† are described in this paper. It discusses the changes made to the kernel, the C library and to user-level software to accommodate NFS. Some comparisons on the performance of the VAX 11/750, Sun-2 and Sun-3 computers as both NFS servers and clients are also given.

The problems that were encountered during the installation are also explained. While these may not be typical of the difficulties that can be expected, they should give some indication of the problems that are likely to be faced. Clearly, these will depend on local circumstances such as the hardware configuration or on the nature of any software modifications, particularly those made to the kernel.

The paper discusses various operational difficulties that became apparent when the VAX NFS service became widely used. Details of the adjustments that were made to alleviate or solve these problems are also described.

Some familiarity with the NFS protocol suite, the TCP/IP protocol family (especially its UNIX implementation) and the internals of 4.2 BSD is assumed.

2. Overview of NFS

Sun's Network File System extends the conventional notion of a UNIX filesystem where files are referenced by inodes to a Virtual File System (VFS) using vnodes to access files. A Virtual File System may be mapped to a physical UNIX filesystem on the local host or to a remote file system accessed across the network which may or may not be a physical UNIX filesystem.

Within the vnode is a pointer to a structure which contains a series of pointers to functions defining the operations that may be carried out on that vnode. The operations defined for a given vnode - i.e. the functions that are used to manipulate the vnode or the file associated with it - are determined by the type of filesystem that the vnode refers to. These structures are initialised for each filesystem at mount time.

---

† UNIX is a trademark of AT&T Bell Laboratories.
VAX and UNIBUS are trademarks of Digital Equipment Corporation.
The combination of Sun with a numeric suffix is a trademark of Sun Microsystems Inc.
NFS is a trademark of Sun Microsystems Inc.

For a UNIX filesystem (UFS) that is physically located on the local host, the vnode operations are the usual UNIX system calls - open(), close(), read(), write() and so on. These use inodes for file manipulation in the conventional manner.

For network filesystems, most of the vnode operations are the functions that form the NFS protocol, and have almost a one-to-one mapping with the UNIX filesystem system calls[1]. Instead of acting on an inode, the NFS vnode operations use a structure known as file handle which identifies the file on the NFS client and server hosts. File handles are passed between NFS clients and servers by means of Sun's Remote Procedure Call (RPC) mechanism, along with any data required for a given vnode operation or NFS function. NFS uses UDP to perform remote procedure calls[2].

The responsibility of checking access permissions is shared between the NFS and RPC modules. The remote procedure call modules check the authenticity of the caller and the NFS modules check the user and group id's against the permissions of the file, given by fields in the file attribute structures returned from other NFS calls. The version of the RPC authentication (and hence NFS) in use relies on each user having the same user id and group ids across all hosts on the network.

To ensure portability of data another protocol called External Data Representation (XDR) is used. The XDR routines are responsible for converting basic data types - shorts, ints, doubles and so on - to and from a canonical form, so avoiding machine-specific dependencies like bit and byte ordering and word size. XDR can also be used for combinations of arbitrary data structures.

3. Installation

The distribution provided by Sun assumes that NFS is to be installed on a ''vanilla'' 4.2 BSD system. This distribution was essentially a subset of the source code of Version 2.0 of SunOS, suitably modified for the VAX.

Before the installation could proceed, some fixes had to be applied to the C compiler. These corrected some cases of incorrect constant folding, spurious type mismatch errors and faulty handling of casts in compile-time initialisations.

Armed with a modified C compiler, installation of the NFS sources could begin. The installation entailed significant modification of the kernel and some changes to the C library. A few user-level programs also needed to be altered and new ones added.

The installation of NFS took around three weeks. In view of the substantial changes that were needed, great care was taken to save the original sources, binaries and libraries so that in the event of a disaster everything could be restored to the original state. About a week was taken up recompiling the kernel, the libraries and all the user programs. A further week or so was spent tracking down and fixing two kernel bugs that appeared to be the result of race conditions. In retrospect, the bugs were simple to find and correct. [The VAX used for the installation was providing a computer service so time for testing and debugging the kernel was limited. If a dedicated system had been available, the debugging time would have been considerably shorter.]

4. Changes

This section describes the changes that were made to accommodate NFS. Details of the modifications to the kernel, the C library and to various utilities are given below.

---

[1] For remote filesystems, the system calls are usually performed on the NFS server in response to requests from NFS clients.

[2] The stateless nature of the NFS protocol is simple enough to justify UDP over a more complex transport protocol such as TCP. However, this can introduce an interesting problem which is discussed in more detail in section 7.

4.1. Kernel changes

As anticipated, the kernel had to undergo the greatest amount of modification. The most global change was the introduction of the virtual file system layer and the addition of vnodes. This meant converting most of the kernel modules that used inodes to use vnodes instead. With all VFS file operations done with vnodes, the virtual file system layer was introduced on top of the conventional UNIX filesystem using inodes and the Network File System using file handles.

Sun's NFS distribution provided code for the VFS layer and for vnode manipulation. Also added to the kernel were routines for the remote procedure call mechanism and the external data representation protocol.

Module	Text Size (Kbytes)	Data Size (Kbytes)
VFS	9.5	0.3
Vnode Operations	13.1	1.5
RPC Mechanism	9.9	1.7
NFS Client/Server	8.5	1.0
XDR Modules	6.3	1.2

Table 1 - Size of Code Modules added to NFS Kernel

Overall, the kernel text size was increased by about 45 Kbytes. The static data size remained roughly the same. The initialised data segment was increased by about 10 Kbytes and the bss segment reduced by the same amount. However, extra data space is dynamically allocated and freed through the new heap manager (see below).

The translation of pathnames to inodes (or rather vnodes) is markedly different between the NFS and BSD kernels. Instead of passing the complete pathname into a lookup routine, the lookup is done one pathname component (i.e. a directory) at a time. This new lookup routine may then make an NFS request, depending on whether the pathname component is a local or a remote file.

This style of lookup helps make NFS less UNIX-specific. NFS defines a "generic" directory format, albeit one which is UNIX-like. Accessing a remote directory through this format hides the portability problems that would inevitably arise if an NFS client was to read a server's directory directly. It just wouldn't do for a client to read a remote directory like this if a server's directory format was different from that on the client.

By traversing pathnames one component at a time, NFS avoids the problem of defining and implementing a standard pathname representation[3].

Heap storage is used to build up the pathname. A LRU cache of vnodes and directory names is used to improve the speed of lookups.

One of the spare fields in a physical filesystem inode is used to store a generation number which gets incremented when the inode is freed. The generation number prevents a race condition where an inode could be freed and reallocated to another file while an NFS client still had a file handle that referenced an earlier file that used the same inode. To prevent this happening, the generation number is included in the file handle structure passed in NFS transactions.

The system call argument passing has been "enhanced", passing the call parameters directly from the per-process user area. This enabled system calls to be called from inside the kernel. The NFS system calls gain a performance benefit from this as it can reduce the amount of context switching needed when executing NFS operations.

[3] To the author, it appears that lookups will still go horribly wrong if a UNIX NFS client gets a filename with a '/' character in it from a non-UNIX NFS server.

New system calls were added for NFS client and server operations. A second mount system call was added for mounting remote filesystems. The unmount system call only needed a little modification to work for local and remote filesystem unmounts. The semantics of mounting local and remote filesystems are more involved than the converse operation. [Later versions of SunOS have just one mount system call to handle the mounting of local and remote filesystems.]

A system call was added to get directory entries. For remote directories, the NFS read directory call returns some status information as well as the file name and remote filesystem inode number. Another was added to get a file handle for a given file descriptor[4]. Two variants on the *stat* system call were also added to provide the filesystem attributes for a VFS - essentially the size of the filesystem and the number of used and free blocks. These four system calls are not strictly necessary. As they map directly to functions in the NFS protocol, presumably they have been added as system calls for reasons of performance.

Finally, another two system calls were added to set/get the domain name. The domain name is only used by the Yellow Pages daemons.

Sun claim the UDP packet size limit was increased from 2 Kbytes to 9 Kbytes[5]. This would partly be a side-effect of the statelessness of the NFS protocol. NFS operations are considered atomic, so a larger packet size is needed for better throughput for NFS block I/O transfers that would generally be 4 or 8 Kbytes in size, plus a few bytes for protocol headers and NFS status information.

Several bugfixes and performance improvements were made to the networking code and ethernet device drivers. Some subtle changes were made to the mbuf manipulation code. These had some interesting interactions with a DEUNA device driver that took no account of these alterations (see section 4 below).

A heap manager was added for dynamic storage allocation of NFS data structures such as file handles and filesystem or file attributes. These data structures tend to be transient and somewhat small. Static storage allocation or grabbing clists, mbufs or even blocks from the buffer cache would be wasteful.

For debugging, a stack traceback was added for kernel traps. When called as a result of a kernel panic, the traceback would print the values of some of the stack frames to trace the events that lead to the crash. This was of little help when the panic was the result of a machine check. It was later necessary to make a simple change to the reboot system call to force the kernel to dump memory when rebooting.

Some minor changes were also made to some kernel data structures. This meant that programs which accessed the kernel namelist needed to be modified. [For instance, to interpret the load average statistics as longs rather than as doubles in the 4.2 BSD kernel.]

4.2. Additions to C library

Extra trap entries were made for the new system calls, requiring the C library to be rebuilt.

Since the format of */etc/fstab* changed to allow for mounting network filesystems, extra routines were provided to read the new format. The existing *getfs* lookup functions were re-coded in terms of the additional routines, providing backwards compatibility at the C source code level.

The directory reading routines were altered to use the new *getdirentries* system call, making the read times for remote directories comparable to those for a local directory.

Also added were user-level RPC and XDR routines, making the protocols readily available for general use. Some of the Sun-supplied daemons make use of these facilities (see section 4.4).

---

[4] Since the NFS servers are stateless, open and close system calls are handled by NFS clients using the information returned from remote directory reads to generate file handles (and hence vnodes and file descriptors) to reference the file.

[5] Later releases allocate even more space for UDP datagrams used in NFS requests.

Support for the Yellow Pages was added to some of the lookup routines for several files in */etc: group, hosts, networks, passwd, protocols, services* and *rpc*. [Sun's Yellow Pages provides a distributed lookup service that uses RPC to get information from Yellow Pages servers. This saves repeatedly updating several files as a new user or host is installed on the network.]

The dbm database code was given a close routine, to reduce the number of active file descriptors used. The main beneficiaries of this dbm close are the Yellow Pages daemons that tend to have several dbm files in use at once.

4.3. User Programs

Most of the user programs were unchanged at source level as a result of installing NFS. Once they were recompiled and relinked with the new C library, the programs incorporated the new features. All user programs were recompiled. This was not really necessary. The only ones which had to be changed were those that read information from */dev/kmem* that had changed between the 4.2 and NFS kernels. It was easier to recompile everything rather than select which sources to recompile and install. In any case, it was best that the utilities were relinked with the new C library and re-installed.

As mentioned above, it was not necessary to change programs that used the 4.2 BSD library routines to read */etc/fstab*. The old routines were rewritten to use another set that understood the new format of the file.

One or two programs were changed as a direct consequence of the addition of NFS. These were modified because of possible time skews between clients and servers that could set file access times and so on to "impossible" dates in the future.

The main ones that were affected were *ls* and *ranlib*. For *ls*, a test was added to check that the modification date of a file was greater than the current time and if so print the date rather than the date and time. *Ranlib* was changed to ensure that the date stamp in the table of contents was always less than or equal to the modification date of the library.

The *shutdown* command was altered to make a remote procedure call invoking a broadcast to everyone on clients served by the host that was about to shutdown.

Fsck was changed to understand the generation number field added to an inode.

Not surprisingly, *mount* was changed so it would understand the difference between mounting local and remote filesystems.

4.4. New Programs

Some new daemons and RPC programs were supplied. The most obvious were the NFS server and client daemons. These are trivial programs because the NFS protocol code is in the kernel. All they do is detach from the control terminal, make the appropriate system call and let the kernel do the work.

The *portmap* daemon was supplied for remote procedure calls. This has the task of mapping sockets to programs for remote procedure calls. *Portmap* listens for incoming RPC requests on a known port number and returns the port number the requested RPC server is listening on and some status information. RPC servers pass information to this daemon when they start up.

Some RPC server programs came on the distribution. These could be kicked off from the inet daemon which was also supplied. One RPC server program is a mount daemon that checks NFS mount requests from NFS clients. Another reports kernel activity statistics - the load average, disk transfer rates and so on. Another provides a remote broadcast facility - *rwall* - which is mainly used to warn users on NFS clients when an NFS server is about to shutdown.

Code to support Sun's Yellow Pages (YP) service is also provided. This has daemons that use RPC to provide a distributed database for name service lookups. Updating the Yellow Pages on one YP

master causes the other masters, if any, to be updated automatically and the changes then dispersed to the YP clients. The data is stored in dbm format for fast access.

A further couple of programs report on NFS and RPC statistics.

5. Installation Problems

Two trivial problems prevented the NFS kernel being compiled smoothly. There were a couple of obvious typing errors that had to be corrected in the NFS source code. The symbol sort of the kernel namelist had to be fixed to take account of a change in the names of some kernel data structures.

When the new kernel was booted, two serious faults occurred. The first caused the kernel to crash, panicking "mclput". This happened every time a bulk data transfer was received from the ethernet - typically remote printer requests or file transfers. Ethernet terminal traffic was unaffected. Inspection of the source code and the kernel core dumps suggested that a race condition was corrupting an mbuf cluster - the kernel would panic when asked to free a cluster-type mbuf that was already marked freed.

The second problem was that the NFS daemons hung shortly after NFS requests were made. The daemons would go into an infinite busy wait and could not be killed. In this case, the daemons were waiting for cluster mbufs to become free. This gave further credence to the theory of a race condition.

It was decided that it would be necessary to trace all mbuf activity. The symptoms were characteristic of a race condition. Work was started to add tracing information to every section of code that manipulates mbufs.

This was not to be a trivial task. The mbuf code is used all over the kernel, both as macros and function calls. Kernel printf's would be too slow because of the delay involved. (Mbuf manipulation is often done at interrupt time. If something was printed, the network event would be likely to be missed by the time the console I/O completed.) Analysing kernel core dumps would be little better - the supposed race condition could involve the interaction of two or more processes and the potential offender could have died or exec'ed long before the race was detected.

However, closer inspection of the DEUNA device driver showed that there was no race condition. The solution was embarrassingly simple. The DEUNA driver was using the original 4.2 mbuf manipulation macros. These did not know of (or therefore initialise) a new type identifier for cluster type mbufs. So when it passed those mbufs into the IP and UDP protocol handlers, the cluster type would be unspecified. When the protocol handlers came to free the mbuf, the extra type field would have a random value that would then cause a panic. In the case of the NFS daemons, the cluster type is used to call a function to wakeup the process(es) sleeping for the transfer to be complete and the mbuf freed. This time the DEUNA driver would free the mbuf but would never call the function to wakeup the daemons. The DEUNA driver was promptly changed to use the proper code for manipulating mbufs and the problems were solved.

This DEUNA device driver left some scope for improvement, and its shortcomings soon became painfully clear. These are described in the next section.

6. Operational Problems

As the VAX NFS service became operational, further problems with the DEUNA driver became apparent. Occasionally, the VAX would stop transmitting ethernet packets. This was quickly traced to a bug in the device driver. If the transmit queue of the DEUNA was full, the driver simply wouldn't initiate a packet transmission. Although the bug had been present for some time, it had gone unnoticed prior to the installation of NFS. Clearly, NFS service was placing higher demands on ethernet throughput.

Another problem was that the Sun-3's would often complain about NFS timeouts - the VAX was too slow at responding to requests from these clients. It was considered that there were too few NFS kernel contexts. Sun recommended running four *nfsd* NFS server daemons. By trial and error, we found that it was best for the VAX to run eight server daemons. This increased the number of NFS server processes that could be scheduled, making more kernel contexts available for NFS service and so giving higher NFS capacity. This change reduced the number of NFS timeout warnings dramatically.

For a while after these changes were made, the VAX kernel was plagued by a spell of unreliability. It was common for the kernel to either crash or hang every two or three days. This was particularly frustrating as it proved very difficult to get a crash dump for analysis. Either the VAX generated a machine check as the dump was being taken or the dump would fail to be recovered by *savecore*. The analysis of the few crash dumps that were recovered proved to be inconclusive - elementary data structures such as the kernel stack were corrupt.

A further nuisance was the DEUNA driver regularly complaining about unavailable buffers. This was happening because the DEUNA was trying to transfer packets into buffers in VAX memory faster than the VAX was able to make buffers available. The original device driver only allocated space for three buffers - two for incoming packets and one for outgoing packets.

After experimenting with various numbers of buffers, we settled on using thirty-two buffer descriptors for incoming packets and sixteen for outgoing packets. This figure was a compromise between the needs of the other UNIBUS DMA devices and the requirements of the DEUNA. UBA map resources had to be left for the other DMA devices.

This eliminated the complaints about unavailable buffers from the DEUNA, but caused another problem. Another device - a frame store interface - would only function if it could perform DMA in the bottom 64 Kbytes of the UNIBUS address space. The DEUNA was occupying that space since it was the first device to be allocated UNIBUS map resources and it was taking the first 72 Kbytes of the UNIBUS address space. The DEUNA driver was changed to allocate the first 64 Kbytes of address space and then take the space it needed for its own buffer descriptor entries. After that was completed, the first 64 Kbytes of space would then be freed, making it available to the frame store. The code to do this was ugly.

Curiously, the reliability of the kernel was improved by increasing the buffer space available to the DEUNA. Hangs and crashes became much less frequent, though it is not clear how the change had this effect.

7. User-Generated Problems

After all the upheaval and adjustments mentioned above, the VAX NFS service has settled down and now works well for most of the time. "Normal" I/O from users on NFS clients does not have a dramatic impact on the VAX, although it is noticeable. There are however some user actions that are very damaging.

Processes which don't use the usual block I/O mechanism can prevent severe problems. Any recursive traverse of a remote filesystem puts extreme stress on the file server. In our environment, it was necessary to modify the *find* and *du* commands to prevent them wandering into remote filesystems by default.

Further trouble arises when a fast NFS client does non-block I/O to files on a slow NFS server. Examples include linking executables with the *suntools* or *suncore* libraries, dumping core or demand paging from an executable image on a remote filesystem. In the worst cases, both the client and server grind to almost a complete standstill.

It soon became clear what caused the problem. The *nfsstat* program was reporting an excessive number of retransmitted NFS requests. The Sun-3 client was transmitting the NFS message as eight or nine packets, sent out back-to-back on the ethernet. The VAX could not always accept such a burst of data because the DEUNA was sometimes forced to drop the packet because the interface did not have the internal space to store it. The VAX would try to assemble the packets comprising the NFS

datagram, find there were fragments missing and eventually throw the NFS request away.

Meanwhile, the client would notice that its last request had not been replied to. It would then retransmit the message, causing the cycle to repeat itself. To make matters worse, the bulk of this processing gets done at interrupt time (particularly on the server), leaving little CPU free for other processes.

The DEUNA could cope with burst traffic like this now and again. However, it could not cope when there were several sustained bursts coming one after the other as a result of NFS requests caused by processes that did not perform normal block I/O.

This condition arises out of the statelessness of the NFS protocol. There is no notion of flow control between clients and servers. Since UDP datagrams are used to send NFS messages, it does not matter to the transport protocol if the data even reaches its destination. The NFS protocol could be extended to include client and server status information. That would make the NFS servers stateful, as would using a reliable, flow-controlled transport protocol like TCP. As a work round solution, later versions of NFS have parameters to the mount command to try to achieve a better match between the performance of heterogeneous NFS clients and servers.

8. Timings

A series of benchmark timings are given in the tables below. The first set are for simple block I/O using block sizes of 1, 4 and 8 Kbytes. The VAX and Sun-3 server filesystems used an 8 Kbyte block and 1 Kbyte fragment size. The The Sun-2 filesystem used a 4 Kbyte block and 1 Kbyte fragment size. For comparison, timings are also given for remote copies between each system. The results are presented in tables 2(a), 2(b), 2(c) and table 3 below.

The second set of timings measure the effectiveness of directory lookups on a VAX NFS server. Table 4 below contains the results.

All the systems were lightly loaded when the timings were done. They were up multi-user, but had only one user logged in during the tests. Since the UNIX time command was used, the system times given are approximate though the overall elapsed times were checked with a stopwatch and are accurate. All times in the tables below are in seconds.

In normal operation, the VAX and Sun-3 are clients and servers of each other. To prevent deadlocks when rebooting, the NFS filesystems are soft mounted. Both kernels wait forever for a hard NFS mount to succeed.

The Sun-3 used release 3.0 and the Sun-2 release 2.0 of Sun UNIX. Of course, the VAX used 4.2 BSD. The hardware used was as follows:

CPU	RAM	Disk Drive	Disk Controller	Ethernet Interface
VAX 11/750	4 Mbytes	2 × Fujitsu Eagle	SI 9900	DEUNA
Sun-3	4 Mbytes	1 × Fujitsu Eagle	Xylogics 450/440	Sun-2 (Interlan?)
Sun-2	2 Mbytes	1 × Fujitsu M2243 AS	SCSI	Sun-2 (Interlan?)

Table 1 - Hardware used for benchmarks

8.1. Block I/O

The tests were carried out using all systems as NFS clients and servers. In cases where the client and server systems are the same, the timings are for I/O to the local disk on that system. The times given are an average of 10 runs, reading or writing a 1 Mbyte file using 1, 4 and 8 Kbyte block sizes. Between each run, a few minutes of normal disk activity was done to minimise the impact of any buffer cacheing on the client and server systems.

```
#include <stdio.h>
#define BLOCK           1024
#define ONEMEG          (1024*1024)
char buf[BLOCK];
main()
{
        register int a, b, c;

        if (a = creat("foo",1) < 0) {
                fprintf(stderr, "Can't create foo\n");
                exit();
        }
        for (b = 0; b < ONEMEG/BLOCK; b++)
                if (c = write(a,buf,BLOCK) != BLOCK)
                        fprintf(stderr,"Write error: c = %d\n", c);

        close(a);
}
```

```
#include <stdio.h>
#define BLOCK           1024
#define ONEMEG          (1024*1024)
char buf[BLOCK];
main()
{
        register int a, b, c;

        if (a = open("foo",0) < 0) {
                fprintf(stderr, "Can't open foo\n");
                exit();
        }
        for (b = 0; b < ONEMEG/BLOCK; b++)
                if (c = read(a,buf,BLOCK) != BLOCK)
                        fprintf(stderr,"Read error: c = %d\n", c);

        close(a);
}
```

Figure 1 - 1 Kbyte read and write programs (the 4 and 8 Kbyte versions are similar)

The CPU times are an average of the time each process spent in the kernel during a run. User time was negligible. The programs used for reading and writing are given in figure 1 below.

Table 3 gives comparative figures for remote copies using *rcp*. The file used was also 1 Mbyte in size.

I/O OPERATION	CLIENT					
	VAX		Sun-3		Sun-2	
	CPU	Elapsed	CPU	Elapsed	CPU	Elapsed
8 Kbyte read	1.7	3.2	0.4	7.0	1.1	11.3
4 Kbyte read	1.8	3.3	0.5	7.1	1.5	12.1
1 Kbyte read	4.1	4.0	0.9	7.4	4.1	12.7
8 Kbyte write	2.9	3.1	0.9	23.0	2.3	28.3
4 Kbyte write	4.0	4.8	1.4	22.8	2.5	29.7
1 Kbyte write	10.7	11.0	1.3	22.9	6.3	35.1

Table 2 (a) - VAX NFS Server

I/O OPERATION	CLIENT					
	VAX		Sun-3		Sun-2	
	CPU	Elapsed	CPU	Elapsed	CPU	Elapsed
8 Kbyte read	2.1	8.2	0.3	2.2	2.0	7.4
4 Kbyte read	2.8	10.6	0.6	5.4	2.5	22.7
1 Kbyte read	7.6	13.0	1.3	5.9	5.0	31.6
8 Kbyte write	4.2	23.3	0.6	2.0	2.6	21.9
4 Kbyte write	3.9	26.8	0.6	5.1	2.5	22.5
1 Kbyte write	9.7	43.4	1.3	5.8	5.4	31.6

Table 2 (b) - Sun-3 NFS Server

I/O OPERATION	CLIENT					
	VAX		Sun-3		Sun-2	
	CPU	Elapsed	CPU	Elapsed	CPU	Elapsed
8 Kbyte read	2.5	10.3	2.3	7.0	2.4	5.6
4 Kbyte read	2.5	11.2	2.4	9.6	1.8	5.5
1 Kbyte read	5.4	14.8	4.1	14.2	4.0	6.9
8 Kbyte write	3.7	46.9	2.9	44.4	3.1	6.3
4 Kbyte write	4.6	45.1	3.3	39.6	3.2	6.0
1 Kbyte write	16.9	56.6	11.3	45.9	6.0	7.8

Table 2 (c) - Sun-2 NFS Server

It can be seen from the above tables that the systems have a broadly comparable throughput for serving NFS read requests at between approximately 100 and 150 K bytes per second. This figure is approximately 10-15% of the maximum bandwidth of Ethernet and of the UNIBUS. Bearing in mind the differences in raw CPU performance, this would suggest that the throughput on the ethernet interfaces is the limiting factor here. This transfer rate compares quite well to the normal disk I/O throughput which is roughly twice that of NFS. Obviously, the data has to be fetched from the server's disk before it can be passed over the network.

Sequential reads are a best case for NFS. Both the client and server will notice that the process is reading the file sequentially. The client and server will both initiate read-ahead to give a double cache effect. By the time the server has fetched a read-ahead block, there is a good chance it will shortly see

an NFS request for that block as the client will have started its own read-ahead for the same block.

The times for NFS write requests are also somewhat similar. It takes approximately seven times as long to write a remote file with NFS as it would to write the file locally. The throughput of NFS writes is much lower than that for NFS reads because each write request is performed synchronously on the server. [Again, this is a result of having stateless NFS servers. If NFS write requests were queued for transfer in the normal way, it is reasonable to suppose that NFS write requests would be as fast as the read requests.] The tables show a data transfer rate of between roughly 30 and 50 Kbytes per second.

SOURCE	DESTINATION								
	VAX			Sun-3			Sun-2		
	User	System	Elapsed	User	System	Elapsed	User	System	Elapsed
VAX	0.4	10.7	33.5	0.3	16.3	20.0	0.3	15.7	27.0
Sun-3	0.1	3.3	27.4	0.0	2.9	14.2	0.0	3.2	25.4
Sun-2	0.2	12.8	30.6	0.3	13.9	29.5	0.1	11.6	46.7

Table 3 - Timings for a remote copy of a 1 Mbyte file

The data transfer rate for remote copies in the above table is generally around 30 to 40 Kbytes per second. This is comparable to the performance of NFS write requests, but is two or three times slower than the transfer rate of NFS reads. The difference is probably because of the extra processing needed for a TCP connection (as in an rcp) as opposed to the minimal overheads of an NFS UDP "connection". It would be interesting to repeat this comparison using the new fast TCP/IP code in 4.3 BSD.

8.2. Directory Lookup

One of the filesystems exported by the VAX to the NFS clients is /usr/man. Table four below gives the time taken to search the filesystem to find a given filename on both Sun-2 and Sun-3 NFS clients and on the VAX. As this involves some recursive directory searching and the scanning of some fairly large directories, it should give a reasonable idea of the effectiveness of directory lookups.

Host	User	System	Elapsed
VAX	35.0	35.8	82.3
Sun-3	8.1	39.6	155.4
Sun-2	36.2	51.1	211.8

Table 4 - Time for a recursive ls of /usr/man

From the times above, it should be clear that although the directory cacheing may improve performance, exhaustive lookups are fairly slow for remote filesystems using NFS. It was noticed that while the clients were performing the lookup, the load average on the VAX server increased quite dramatically because the kernel was doing a great deal of protocol handling and the NFS server daemons were kept in a busy wait.

9. Conclusions

The installation of NFS was more straightforward than had been anticipated. Providing the original kernel is largely unchanged from the 4.2 BSD release, NFS can be installed on a VAX in about a week. Most of that time will be taken up by recompiling sources. The interaction of the ethernet device driver with the mbuf code in the kernel should be carefully checked. This area is likely to give problems when a "non-standard" device driver is used.

A system administrator who plans on installing NFS on a VAX should be prepared to experiment to provide a more robust and/or faster service. Areas that are clearly worth exploring include the I/O throughput of the ethernet interface and the numbers of NFS client and server processes. For distributions based on Version 3.0 of SunOS or greater, it would also be worthwhile to try several values for the mount parameters such as the NFS timeout and the block size of NFS reads and writes.

The installation of NFS has had no noticeable impact on the performance of the VAX or on user programs. Most user programs are unchanged by the addition of NFS, though those which use the Yellow Pages file lookup routines experience an increase of about 12 Kbytes in text size. This is because of the extra code needed to parse the YP file formats and then make remote procedure calls when needed.

It should be noted that NFS in the 2.0 release of SunOS is extremely trusting. The routines blindly accept whatever data is presented with no real attempt to verify the identity of the client making the request. The NFS and RPC routines will happily accept an NFS request from anywhere, provided the request looks to be in the proper format. They will also assume that the user-id and group-id information on any request was legally filled in by the client. In short, anyone with knowledge of the protocols could easily compromise the server's security.

This glaring security problem has eventually been fixed. Version 3 of SunOS has added a check so that only NFS requests that are made from privileged ports are fully trusted. [It's not much help, but at least it was a start.] Another optional check would verify the IP address of clients making mount requests. In Version 4 of SunOS, a much more secure authentication system is available. This uses a combination of DES and public key encryption with time stamps to make it extremely difficult to impersonate NFS clients or servers.

Perhaps the most important conclusion to be drawn is that NFS works best when the clients and servers are equally matched in terms of performance, especially in ethernet I/O throughput. It appears that some vendors cannot keep up with Sun's ethernet interfaces - a VAX 11/750 with a DEUNA surely can't - which can readily take up most of the bandwidth of an ethernet cable. If the performance mismatch between an NFS server and a client is wide enough, the consequences are unpleasant for both the NFS client and the server.

10. Future developments

We have no plans to continue working on the VAX implementation of NFS. This is for several reasons. Firstly, our VAX has a hard job trying to act as an NFS server and we no longer feel comfortable using it in using it in that capacity. Secondly, the VAX 11/750 that has been used is expected to be phased out of service shortly. There are also licensing problems that make it awkward for us to obtain a merged source distribution of NFS and 4.3 BSD. Sun only distribute a VAX NFS implementation based on 4.2 BSD, not 4.3 BSD. Even without the licensing difficulties, merging the latest NFS release from Sun with 4.3 BSD is a non-trivial task that is of little benefit given the imminent demise of the VAX. Furthermore, our VAX kernel also includes X.25 socket code from the University of British Columbia. This would also be affected by any move to some combination of 4.3 BSD and a VAX NFS distribution.

11. Acknowledgements

I would like to thank my colleague Jon Malone for his comments and suggestions on the drafts of this document. Allan Black deserves a mention for his help in debugging the kernel dumps. Special thanks are due to Rusty Sandberg at Sun Microsystems for his encouragement and advice when things were going wrong.

Networks : Social And Legal Implications

James Watt
School Of Computing And
Information Technology
Griffith University

ABSTRACT

This paper examines some of the the legal and societal issues posed by the rapid increase in the use of computer networks.

Key Words : Virus,Electronic Mail,Anonymity,Legal And Social Implications

INTRODUCTION

Computer networks have grown at a staggering rate in the past few years. This development and spread of use has brought to bear many benefits. Benefits such as enhanced communication, the sharing of resources,increased information flows and a gradual shrinkage in the time needed to make contact globally.

More and more these services are being enjoyed by a wider cross section of society.

Although technically we surpass our goals daily,the social and legal implications of these advances are often overlooked,or ignored altogether.

It is our belief that we should, as a profession and as a

society, have a greater awareness of the potential pitfalls that such advances in technology can bring. Only by doing this are we able then to take the appropriate steps necessary for their solution.

This paper discusses some of the legal and social issues relating to the use of electronic mail, and to the spread of viruses,both of which have important ramifications for network systems.

ELECTRONIC MAIL

The advent of electronic mail has in many respects revolutionized our means of communication.It is a fast,cheap and dependable form of communication, quite different in many respects from other more conventional written means of interaction.

When we think of electronic mail,there are certain psychological factors which come to mind.Kiesler,Siegel and McGuire[1] tell of the social anonymity of electronic mail.They also describe studies which reveal that users of electronic mail are sometimes more aggressive and less inhibited in their language and manner than they would be otherwise. I am sure we have all seen on many occasions such examples of what is quaintly termed "flaming".

Much of the flaming that does take place is clearly defamatory,and often obscene. It would seem that many people do not consider email a "legitimate" form of communication and that common courtesies and conventions need not apply in this medium.

Electronic mail enjoys widespread use in any network

[1] Kiesler,Sara,Siegal Jane,McGuire,Timothy W,(1984),Social Psychological Aspects Of Computer-Mediated Communication,American Psychologist,vol 39,no 10,p1123-1134

system.With the rapid "internationalization: of networks almost no part of the modern world will be inaccessible to the electronic postman.

It would seem appropriate then for some form of standard to be adopted.This may then be used to regulate out much of the destructive traffic and allow for fuller more constructive communications.

The anonymity spoken of earlier also poses some legal questions.It would seem not at all difficult for a person to assume the identity of another.

When one receives a message through electronic mail,how often do you stop to wonder whether or not the sender is who they purport to be?

This is especially a problem when emailing internationally. This has the potential when placed in devious hands to lead to serious and potentially damaging situations.

Concern has also been voiced about the transferral of information from one country to another.What if the information is illegal in some parts of the world but not others? And what too about the countries in between ? These issues seem to remain unresolved even though quite extensive discussion has taken place.As the tentacles of networks grow,this type of problem can only worsen.

Perhaps part of the answer is to condition people to treat electronic means of communication,in a manner similar to any other form of interaction,and promote some formality into the informality that seems to exist at present.

THE VIRUS

The virus poses a great threat to every user of computing

technology.The possible erasure of valuable data and programs is as much an economic nightmare as a technical one.It is also non- discriminatory as to whom it attacks.The so called Christmas Tree virus demonstrated that no one is immune, no matter how large.

Most at risk are the large institutions, universities, research institutes.These installations often contain hundreds of micro-computers and many mainframes.A single infection could spread its way through and effect every machine in a very rapid time.

The cheapest and easiest way to transport the virus is through billboards and networks.The network provides the ideal vehicle to insure maximum exposure for the infection.

Within the computing community the virus is beginning to cause concern akin to that shown within the general community about aids.What we are beginning to see is a growing mistrust,fear, and a reluctance to accept software on face value.The question "where has this program been?", may become the first we ask.

The public domain and shareware market has no doubt felt some of the ill winds. Pity, as this form of software is popular and was a valuable asset to many in the past.

It may in the long term also alter the way people view networks.This may have the effect of undoing many of the benefits of network technology,as people do not avail themselves of many of the services through fear of contamination.

The virus will no doubt cause some lowering in the public esteem about computing, with the feeling that computers are insecure and that they should not be trusted with our knowledge.

In the near future viral infections may not only be set up and

run by West German teenagers for a prank,but also by disgruntled employees,criminals and general sabateurs.

A legal issue to rise from all of this is liability.Exactly who is liable for the destruction a virus may cause to valuable information.Should it be the manufacturers of the hardware for not allowing for this,or the programmers who designed the operating system,or the software manufacturers,or the operators of the networks it passes through?As the virus increases this issue will become more and more relevant.

Again here we see that the action of injecting a virus into a computer system is probably perceived by most to be less serious than sending harmful objects through the post,or poising food tins in a supermarket.

REMEDIES

Obviously we deal here with very complex issues,with much interleaving of cause and effects.However through the adoption of some basic concepts,we can at least we can go part of the way to their alleviation.

One such factor is that of education.MacKenzie[2] argues that societal aspects should not be simply optional extras for scientists and technologists.We would endorse this concept and add to it by saying that only through better education and understanding can these problems be properly assessed,and without this no satisfactory solutions will ever eventuate.

To this end it is highly desirable that societal concepts be taught in the curricula of computer science and allied fields.It may be that we need become informaticists, instead of purely technologists.

---

[2] MacKenzie Donald,(1986),Why The Social Aspects Of Science And Technology,Is Not Just An Optional Extra,Computers And Society,vol 15 no 4,p2 -6

It is also desirable that legal studies to some extent should also be incorporated in the curricula,for a basic understanding of legal principle is a necessity for delving into legal questions.We would also argue for computer concepts to be incorporated in some manner in law degrees.

Important also that we frame people's opinions about the use of technology.As mentioned earlier, many do not seem to treat electronic mail in the same manner as they would other forms of communication.

In the past much of this has been left to the social scientists Although not diminishing their part in this, we feel that the technologists,the engineers and the scientists,are the in the best position to tackle this problem.This conclusion is based not only on the fact that they are at the rock face of the technology,but also that they are the only ones that truly and fully understand it.

THE ROLE OF LAW

The law is playing an ever increasing part in the computing industry.However the practice of law,with it's many conventions and fondness for legalise, is often at odds with fast pace of change of technology.

We can see Hoffman[3],that privacy laws can affect the way systems are designed.We believe that laws also can affect our attitudes to the design and use of network systems.

New laws or refinements of existing ones would help give a clearer picture of the situation, and to change our perceptions of network use.However laws only become clearly defined if they are applied in cases, and in the technology area there is a dearth of case law.Also we must learn to see network problems,and technology problems in general within a legal

[3] Hoffman Lance J,(1977),Privacy Laws Affecting System Design,Computers And Society,vol 8 no 3,p3 - 6

framework.

Even so problems will invariably arise.

If we examine the television industry we may be able to see some parallels with computing.

Recently the ownership regulations were changed to allow owners a larger share of the viewing audience.This has meant an increase in the size of networks, and to claims that the viewing public may be disadvantaged by this .What do we do when our computer networks get so large and the same claims are levied?

A well known entrepreneur recently unveiled plans to broadcast via satellite to Britain from Europe,thereby putting himself outside the jurisdiction of the British Broadcasting Authority and freeing himself from all regulations.Regulations for instance relating to such things as censorship and obscenity.

So even in case where laws exist difficulties arise .We feel that in some ways some of the societal and legal dilemmas which have faced television networks could in time affect computer ones.

Nonetheless we see the law as still playing a valuable role and will continue to play that role as networks become more complex and far reaching.

CONCLUSION

Whilst it is true that computer networks bring great advances to many facets of life,they do by their design and operation,give rise to many social and legal questions.

Many of these problems get very little or no exposure or discussion.

Better education and understanding can go some way to alleviating the problems.

The law is invariably becoming more involved in technology,and has a role to play in the management of networks.

We believe that through their deep understanding of the technology, scientists and engineers are better equipped to attempt to deal with these problems,than are the social scientists.

Attempts must be made to address these questions at an early stage,lest we get enveloped by a tangled web of socio-legal problems which undo the technical feats that have so far been achieved.

REFERENCES

Armstrong Mark,(1982),Broadcasting Law And Policy In Australia,Butterworths

Criminal Code Act Qld,1899

Encel Prof Sol,(1985),Information Technology And Communication,National Information Technology Council

Englehart L.K,(1986),Computers And Cultural Change,Computers And Society vol 16,no 1

Holoien Martin O,(1977),Computers And Their Societal Impact,Wiley And Sons

Kippax Susan,Murray J,P(1979),Small Screen,Big Busines:The Great Australian TV Robbery,Angus and Robertson

Pask Gordon,Curran Sussan,(1982),Micro Man - Living And Growing With Computers,CenturyPublications

Risks Digest,ACM Committee On Computers And Public Policy,Peter G Neuman

Moderator

Seymour Jim,Matzkin Jonothan,(1988),Viruses - It's Time To Talk,Australian Personal Computer,vol 9,no 7 ,p 67 -78

Tanenbaum Andrew,(1981),Computer Networks,Prentice Hall

Link Management and Link Protocol in SUN IV

P. R. Dick-Lauder
R. J. Kummerfeld

Sydney University

1. Introduction

This paper describes the design and implementation of link management procedures and link level protocol in version 4 of the Sydney Uni Network software.

2. Link Management

Link management is performed in three areas:

- node description (commands file)
- connection control (config file)
- call script language

The commands file specifies information about the node such as its full address, type hierarchy and mapping information. This file has been described by the authors in the companion paper 'An overview of the Sydney University Network version IV'.

The configuration file specifies the links that connect the node to its neighbours and the times that calls should be made on each link. A link description specifies the link type and how a connection is to be made on the link. The detailed call establishment procedure is described in a program written in a call script language designed for the purpose.

3. Connection Control

Systems that are members of a SUN IV based network have one or more connections to other systems. These connections may be activated continuously, at particular times, when messages are available to be sent, or the connection may be only activated when the remote system calls. The configuration file is used to specify the type of link and how and when connection should be made.

The configuration file is read by the 'netinit' process that runs continuously. Netinit invokes call programs and daemons and creates new link directories when necessary. The file contains an entry for each link from the system. The link may receive incoming calls only, in which case netinit will create the necessary link directories when the link first appears in the configuration file. The link may be a permanent connection such as a leased line and netinit will start and restart a daemon on the line. The link may be 'call on demand' and netinit will invoke the call program when messages are available to send. Finally, the link may be one for which calls should be made at particular times, in which case a 'cron' style time specification may be given for activation of the calls.

Each entry in the configuration file consists of lines that begin with the name of the program to invoke followed by flags that are passed to the program and attributes that

are interpreted by netinit. Here is an example:

```
VCcall -T3 host = cluster callscript = hayes.cs
      crontimes = "20 * * * *"
      loginstr = netaccount
      pwstr = netpasswd
      phoneno = 6923524
         >../cluster/call.out 2>&1
```

The entry can be continued on subsequent lines by beginning the lines with whitespace. Flags and attributes are passed to the program specified but some attributes are also interpreted by netinit. In particular, the attribute ''crontimes = "string"'' allows the specification of times to run the program in the same form as the ''cron'' program.

VCcall entries in the configuration file specify a connection program written in the call script language. Attributes from the configuration file entry are made available to the call script program as string variables. This is useful for giving the values for variables such as telephone numbers and passwords.

4. Call Script Language

The call script language, CS, has facilties for opening and conditioning network connections, carrying on a connection dialogue with a remote system, initiating virtual circuit daemons and continuing a dialogue to close the connection.

CS is a very simple language with a small number of primitives. Wherever possible, features were simplified or even removed in order to reduce the size of the language while still retaining the desired functionality.

The only data structure is the simple variable and the only data types are string and integer. The language has no declarations: variables are created when first mentioned. When a variable is used a check is made to see if it has been given a value thus helping to avoid the problems that arise when variables are not declared.

A CS program consists of a series of statements. There is an assignment statement, several statements that interact with network connections (open, device, timeout, write, and others), control flow statements (test, expect, match, next), statements to initiate daemons (execdaemon, forkdaemon), tracing and error statements and a parameterless procedure call statement.

The *open* and *device* statements take at least one argument that specifies the device type. This is used to select a function that handles operations on this device type. Any further arguments are passed to the function which then carries out the desired action. Functions for simple *tty* connections and a number of other device types are provided, new device types can be added by recompiling a table and linking in the device handling function.

The *expect* and *match* statements provide a simple pattern matching facility. The expect statement reads characters from the network device and attempts to match them

against user supplied patterns, branching to a label elsewhere in the call script when a match is found. The match statement is similar except the input string is taken from a variable rather than input.

The *execdaemon* and *forkdaemon* statements either exec or fork a daemon program that carries out the message transfer with another system. Execdaemon does not return. Forkdaemon returns execution to the call script, making the return status available in a variable. Forkdaemon is used when cleanup operations need to be carried out such as hanging up a line.

The CS program is compiled into a simple code and interpreted by the VCcall program. Programs that are called from within another program are compiled on the first call.

To create a connection the VCcall program is invoked with arguments that indicate the call program to use and any variables that are to be given values on startup. Such variables might include a phone number, a login name and a password for a dialup connection. VCcall then compiles the call program and begins to interprete it. The call program would normally initialise variables, open the network device and condition it and then begin a dialogue to create the connection. The dialogue may include interacting with a modem to autodial the destination, logging in and detecting that the remote VCdaemon has been started. The call progam would then start a VCdaemon locally. After the connection has terminated, the call program may regain control and interact with a modem (or other device) to clean up.

Here is a fragment of a call script:-

```
/*
**    script to call a host via a Hayes compatible modem
**
**    The variables modemdev, linespeed, phoneno, loginstr and passwd
**    are imported from the config file.
*/

      set mcnt 3; /* 3 dial attempts */
      set lcnt 8; /* 8 login attempts */
      timeout 60;
      open "dial" modemdev;
      speed linespeed;

start:
      timeout 60;
      trace "resetting modem";
      sleep 2;
      write "+++";
      sleep 2;
      write "ATZ\r";
      sleep 4;
      expect
            "OK" dialing,
            TIMEOUT nomodem;
nomodem:
      fail "modem not responding";

dialing:
      trace "dialing...";
      test mcnt toomany;
      timeout 90;
      write "ATD" phoneno "\r";
      expect
            "BUSY" rstmodem,
            "NO CARRIER" rstmodem,
            "CONNECT" atlogin,
            TIMEOUT rstmodem;

atlogin:

... at this point the login dialogue takes place
      when the daemon is started the following is executed ...

success:
      trace "connection successful";
      forkdaemon Target; /* returns when daemon terminates */
      sleep 4;
      write "+++";
      sleep 4;
      write "ATZ\r";
      sleep 4;
      write "ATS0=0\r";
      return;
```

The CS language is a convenient way to specify complicated connection procedures for SUN IV. Changes can be made and tested in seconds and the language and its compiler/interpreter are powerful enough to handle almost any network connection.

5. Node to Node Protocol

The node to node message transport protocol has been redesigned and a new implementation technique used. It is a full duplex protocol that multiplexes transmission of several messages in each direction simultaneously. This allows higher priority messages to overtake lower priority. Nine priority levels are specifiable by message senders.

The new protocol has been designed for maximum throughput with minimum transmission costs, and to avoid interference with link-level protocols such as X.25, TCP/IP, or modems with internal buffering protocols. Another criterion was to allow maximum throughput even when many small messages are being transmitted, a condition that noticeably slows the SUN III protocol. Input and output have been decoupled by using one process for each, and all incoming messages are handled by a single permanently running routing process.

5.1 Message Protocol

The message protocol allows the reliable exchange of messages via virtual circuits that may break during message transmission. The message transmitter is free at any time to send a new message on an idle channel followed by up to eight kilobytes of data. The message receiver acknowledges a new message and allows further data transmission as soon as possible after receiving the start of a new message.

For messages being restarted after a break, the receiver indicates the start address by determining the amount of data correctly received before the break. The transmitter won't pre-send any data for a message that is being restarted. On receiving the new start address, the transmitter skips any intervening data, and commences data transmission at the new address.

The effect of this approach is that the full bandwidth of a virtual circuit can be utilised immediately on start-up. The delay that occurs between the end on one message being transmitted, and a successful reception indication allowing the start of the next, can be filled by transmitting a message on another channel.

Messages awaiting transmission are described by ''queuefiles'' linked into the transmitter's operating directory. These files are sorted by name before transmission, and assigned to a channel based on the alphabetic range of the first character of the name. Groups of messages may thus be constrained to travel via certain channels.

Incoming messages are each spooled in their own file, and a ''queuefile'' describing each received message is passed to the routing process on successful reception, by linking it into the router's operating directory.

5.2 Packet Protocol

The packet protocol has been designed to exchange ''messages'' of up to four gigabytes in size by allowing individually addressed data packets to be transmitted in sequence. The protocol makes the assumption that the receiver is always ready to receive more data, and that the virtual circuit supporting communications is mostly reliable. Correct data is never acknowledged, but incorrect data is negatively acknowledged asynchronously. ''Incorrect'' means either that the CRC for the packet data was incorrect, or that a gap in the sequence was detected. The transmitter will send all the data in the message as quickly as possible, responding to negative acknowledgments by retransmitting the indicated data interspersed with normal data.

Synchronisation of the protocol occurs on message boundaries, the transmitter refusing to start transmitting a new message until the previous one has been fully transmitted, and the receiver refusing to acknowledge successful transmission until all outstanding negative acknowledgements have been satisfied. Four separate message ''channels'' allow message buffering, so that when one channel is synchronising, another channel can be transmitting data.

Packets headers contain a 32 bit ''address'' and a 16 bit CRC, up to 1024 bytes of data, and an optional 16 bit CRC for the data.

5.3 Implementation

The aim of the implementation is to de-couple the transmitter and receiver so that they can take full advantage of UNIX's synchronous I/O. They are implemented as separate processes, with a single ''pipe'' connecting the receiver to the transmitter. The only data that travel over the pipe are two message synchronisation packets per message, and any negative acknowledgements. This low data-rate allows UNIX ''signals'' to be used by the receiver process to indicate when the pipe should be read by the transmitter process.

6. Conclusion

Link management and the node to node protocol in SUN IV are major improvements over the previous version. The SUN IV system offers extra flexibility and finer control over links and a higher throughput node to node protocol.

The Development of Distributed Processing within the Queensland Government

Peter Waller
Tony Ashburner

Centre for Information Technology and Communication
Queensland State Government

Tom Crawley
Rob Cook

Centre for Information Technology Research
University of Queensland

ABSTRACT

With the development of low cost, high performance mini-computers, there has been a perceived desire by Queensland government departments to shift from centralised mainframe computing to in-house computing facilities. Investigations had led the Centre for Information Technology and Communications (CITEC) to set up a group to facilitate this shift, and as well provide guidelines in terms of the direction to be taken and support to those groups undertaking such steps. The desired approach was to standardise on UNIX† based systems together with SQL based Relational Data Base Management Systems. To assist groups in their preliminary studies and in the installation and day-to-day running, CITEC developed a 'cookbook' outlining the suggested procedures to be followed and also provided a set of administration utilities which would enable system management by non technical specialists.

This paper describes the existing computing environment within the government, the distributed computing strategy adopted, the place of UNIX within this strategy and the role and development of the cookbook and utilities.

1. Introduction

The Centre for Information Technology and Communication (CITEC, formerly the State Government Computer Centre) provides as one of its services, a bureau facility for most government departments. Typical applications are commercially oriented and involve large data base systems. There is not a large volume of technical work done at the centre, and departments requiring this type of work have tended to procure their own hardware.

Approximately two years ago, internal investigations into the future directions of computing within the government indicated that there would be a growing demand for distributed commercial processing in which applications would be developed using relational databases, with less reliance upon a centralised bureau and reduced dependence upon any manufacture's hardware.

The remainder of the paper describes the distributed computing strategy adopted by the Queensland government, the place of UNIX in the strategy and the key role of the cookbook and toolkit.

† UNIX is a trademark of Bell Laboratories.

2. The Existing Queensland Government Computing Environment

The current Queensland government computing environment consists essentially of:

- a centralised bureau located at CITEC housing large mainframes from Unisys, IBM and Prime, which are available for use by government departments. Major applications are high volume transaction processing on large databases developed using Cobol, Mapper and DB2. A CAD facility based on Prime equipment is also supported.
- a state-wide communications network based in CITEC permitting terminal access to the central computer mainframes.
- departmental computing systems which are often located at CITEC for facilities management. These machines may be independently linked to the regional offices as well as linked to CITEC machines. A number of these machines are UNIX based and are often used for technical computing.
- a large number of personal computing systems and workstations which exist in most departments and may be interlinked via local area networks. These are generally not connected to the CITEC network.

The availability of cheap, high performance systems means that more and more departments are acquiring their own computing systems. The expectation is that a greater range of activities will be computerised providing for more up to date information and improved decision making, enhancing the efficiency and effectiveness of government.

The acquisition of computers by departments will cause changes in the relationship between departments and CITEC. Online database applications running on CITEC machines are controlled by the CITEC DBA group. This group imposes controls to ensure the integrity and security of the department's data. Departments acquiring their own machines will be required to carry out these functions.

It is unrealistic to expect departments using small computer systems to employ specialist computing staff. On the other hand, if CITEC undertakes to provide a central support service without any real control over the type of equipment purchased by the departments, it will have a formidable task to provide expertise in a wide variety of different computing environments to far-flung corners of the state.

The current network design, currently centered around computers located at CITEC, will need to be expanded to reflect the decentralisation of computing resources.

3. A Preferred Distributed Operating Environment

CITEC has recommended a common operating environment to departments embarking on a course of distributing computing, and that application development be done using 4GLs utilizing SQL based Relational Database Management Systems (RDBMS). This environment was seen to provide a number of advantages:

- the solution is independent of the hardware used,
- CITEC would be able to provide systems support to all departments,
- departments would not need to provide their own system support,
- applications software developed in one department could easily be ported to another department with a minimum of change,
- staff could move between departments, and between head office and regional offices, without having to be retrained on new computer systems,
- a single operating environment would make the establishment of a communications network between different departments easier,
- CITEC could set up preferred supplier (term) contracts for the supply of the hardware, reducing the need for each department to carry out their own tendering,
- CITEC could set up separate term contracts for the supply of the RDBMS, again reducing the effort and cost to departments.

There are two candidate operating systems available today that fulfill the above requirements; UNIX and Pick. UNIX was seen to have advantages because of its availability on a diverse range of hardware, the moves towards standardisation, the extensive range of applications and application development environments and the availability of suitable RDBMS.

CITEC commissioned CiTR to produce reports on the management[1] and technical[2] issues involved in basing future development on UNIX. A a result of this study, CITEC set up a UNIX group to undertake the following tasks:

- establish term contracts for the supply of RDBMS,
- establish term contracts for the supply of UNIX based computer systems,
- establish term contracts for the supply of UNIX related consultancy services,
- produce a *cookbook* in collaboration with CiTR which would provide guidelines for the acquisition, installation and setting up of a UNIX system together with suggested procedures for administering the system and simplify maintenance and incompatibilities between departmental systems,
- commission CiTR to develop a *toolkit* of UNIX utilities to enable the system administration of a UNIX system to be carried out by non-technical staff.

The purpose was to facilitate the transition of departments to self-reliance with distributed processing and minimise costs to departments of obtaining such systems.

It should be noted that the establishment of hardware and software items on a term contract does not prevent a government department from purchasing other equipment. However that department will have to call tenders for the supply of the required equipment and carry out their own evaluation of the responses.

4. Departmental Self Reliance

An important facet in CITEC's distributed systems strategy is that government departments and their personnel should be as self-reliant in all aspects of the acquisition, commissioning, systems development and day-to-day operation of their own computer systems. This should be achieved without the departments needing to hire more specialist staff and with a minimum of assistance from the support staff within CITEC. The CITEC role should ideally be restricted to consultancy and assistance with any problems that would otherwise prove intractable.

This aim is being achieved through the recommendation of UNIX as a common operating environment in the following ways:

- acquisition of systems, software and services by departments by way of term contracts as described in the previous section,
- development and trialling application software at CITEC's central service, allowing departments to develop their applications in advance of the delivery of their system in an environment where skilled help is close at hand,
- establishment of a systems administration environment allowing simple to use administration tools that are standard across all systems on the term contract,
- provision of a systems administrator's cookbook that allows non-computer experts to install and maintain a UNIX system,
- provision of support within CITEC of the services listed above and other support as requested by departments.

5. Cookbook Overview

5.1. Objectives

The cookbook, titled *The UNIX Administrators Cookbook*, aims to provide instruction on the acquisition, installation and commissioning of UNIX computing environments (both hardware and software), on the staff that will be required to manage and operate the system and on the administration of an operational

UNIX system.

The target audience consists of two groups:

- the management personnel who are contemplating or have been assigned the task of setting up a UNIX system
- the staff responsible for the day-to-day operation of the system.

It is possible that one or both groups may have a minimal computing experience or little exposure to UNIX administration, and the cookbook aims to assist them without the requiring them to become computer experts.

The cookbook is for administrators, not users. A number of suitable texts for UNIX application developers already exist, and end-users require application specific manuals.

5.2. Philosophy

The rationale behind the cookbook was to provide a wealth of explanatory material to help administrators understand what they are doing and what the outcomes will be. In addition it takes a management viewpoint of the operation of computer systems, stressing forward planning at all points and the creation of an operations schedule.

The majority of the cookbook has been written to be as machine independent as possible. Where necessary, reference is made to specific sections of manufactures' manuals. Other specific information is included as appendices which can be easily modified as required. The version of UNIX which is used in the cookbook is System V.2, which is in accordance with the preferred variant of UNIX to be supported within the government.

The cookbook was written in "friendly" (non-technical) writing style. It was hoped that this would reduce the fears non-computer experts may have to UNIX system administration. Particular effort has been made to keep the level of detail even throughout the cookbook. It avoids describing UNIX is depth, but describes the processes that need to be undertaken and is intended to provide sufficient information for administrators to feel confident that they understand what they are doing.

The major reference is

Fiedler,D and Hunter, B. (1986), *UNIX System Administration*, Hayden Books

which represents a good text for details of the procedures and programs that a system administrator needs to use, and the cookbook references it extensively. However Fiedler and Hunter does not provide a management perspective, nor does it explain the system administration goals and methods of achieving them in a way that a person without a computer background can understand.

The cookbook gathers and collates information that can be found in a number of reference works and has been learned through experience by most UNIX system administrators, presenting it in a form that can be used directly. None of the information it contains is original, but we have not been able to locate texts of a similar style of presentation and orientation towards administrators with limited computer background.

5.3. Organisation and Contents

The table of contents for the cookbook is given in Appendix A. The remainder of this section describes each part of the cookbook briefly and illustrates the way the cookbook is presented.

Section 1 deals with the steps needed to be carried out prior to the acquisition of a computer system. In particular, the approval procedures required for Queensland government departments are outlined, together with a checklist of items to be considered in a requirements analysis. The section is completed by a brief review of the hardware, software and other items which are available to departments and statutory authorities through States Store contracts.

This is the only section which is primarily intended for Queensland government departments.

Section 2 itemises and discusses the steps required to be taken to prepare for the installation of the selected computer systems. The areas covered include:

- the possible administrative structure with role definition and duties together with the training required for these roles
- the pre-installation planning required to ensure the installation and power-up can be carried out with a minimum of problems,
- the day-to-day running of the system.

Section 3 gives an overview of the UNIX operating system from an administrator's point of view.

Section 4 outlines the steps to be carried out in the installation and commissioning of a UNIX system.

Section 5 deals with day-to-day management of a UNIX system. The section begins with a discussion of the tasks that the system administrator is expected to perform, and the schedule that should be followed.

This section references a set of administrative tools developed in association with the cookbook. The administrators toolkit is discussed in more detail later in this paper.

Appendix A contains CITEC's recommended standards for maintaining a consistent set of user account names, file structures, operating procedures and security. Consistency in these areas will become increasing important as use of the communications network increases.

Appendix B outlines a proposed schedule for acquiring, installing and commissioning a new computer system.

Appendix C describes the features of each of the computer systems which have been included on the term contracts for UNIX systems. Included in the description is machine specific information.

Appendix D details the administration tools in the administrator's toolkit.

5.4. Cookbook Development

Development of the cookbook was a joint effort between CITEC and consultants from CiTR. CITEC have a considerable depth of experience in the operation of complex computing environments, but very little exposure to UNIX. CiTR was able to provide the benefit of 15 years of experience with UNIX at the University of Queensland, tempered with the commercial experience of CiTR's project management expertise and experience in producing user-oriented documentation. The cookbook has been extensively reviewed by potential users, both computer literate and non-literate, from government departments.

It has always been envisaged that the cookbook would undergo considerable revision and adaptation as CITEC learns more about their distributed computing requirements; as feedback is obtained from departments using the cookbook; and as other organisations make use of the cookbook in their own environments.

Currently the first version of the cookbook is available to Queensland government departments. After feedback has been collected from practical use, the cookbook will be revised and published more widely.

6. System Administration Tools

Proprietary operating systems such as DEC's VMS and IBM's MVS provide a range of standard administration tools which are operating system specific but generally standard across machines running that particular operating system. UNIX was developed in an environment where less emphasis was placed on system administration functions compared to the programming environment.

When UNIX was taken into the commercial marketplace, this problem was recognised and addressed. Unfortunately, each company marketing a UNIX based system offered a different solution to the same problem. Therefore, although the same version of UNIX would be running on two machines from different manufacturers, the administration interface for each UNIX system could appear to be very different.

CITEC elected to develop a set of tools suited to the requirements for UNIX in the diversified commercial environment of the Queensland government. The tools

- provide a standard system administration environment across UNIX systems from different manufacturers
- ease the system administrator's relationship with UNIX by providing more and better feedback than standard UNIX utilities and requiring simple responses to questions, rather than the requiring the administrator to type the standard inscrutable UNIX commands,
- additional error checking.

Developing the tools inhouse:

- provided a technology transfer from CiTR to CiTEC,
- gave CiTEC access to the source code, ensuring that versions of the toolkit can be readily made available on current and future hardware available on the term contracts.

CiTEC system administrator tools were only developed where it was felt that the UNIX operations required to perform a task were too complex or difficult to describe simply. Maximum use has been made of standard UNIX utility programs, with almost all the tools being written as *shell scripts* calling these standard utilities.

7. Networking

The current network is based around a local area network at CiTEC that connects the larger computers together and a wide area network of terminals that access these computers. Network design and strategy has so far been limited to standardising on the method of terminal interconnection. Applications can also use this access method to access data on other machines via virtual terminals.

As the computers are being distributed geographically more widely, the need for coordination of the information used by computers is becoming increasingly important. Each item of information should ideally be captured once, stored in the state-wide computing system, and made available to whoever has the need and authority to use it. This requirement is driving a movement towards *distributed information systems* in which applications running on one computer can access information stored on the other computers as necessary. This in turn is forcing the development of more sophisticated communications protocols (such as the Applications Layer of the ISO OSI protocols) that allow computer-to-computer transactions to be carried by the same networks that today connect terminals to central computers.

Alternative networking strategies are currently being investigated. CiTEC commissioned CiTR to produce a report on the issues of using, and the current state of, higher level protocols[3]. Subsequently, CiTR undertook a joint study in conjunction with Unisys to review possible approaches for Open File System services applicable to the CiTEC environment[4]. Suitable electronic mailing standards are also being investigated.

8. Future Developments of the Cookbook

The first version of the cookbook is currently being distributed to Queensland government departments. It is premature to judge its success at this stage. Early reviews have been positive, and the level of interest and demand from government departments is high.

The accompanying toolkit is in the final stage of testing by CiTEC before distribution to departments.

Developments that are planned or considered include

- extensions to include networking within UNIX systems in a standardised manner
- extension of the toolkit to include additional facilities eg. a tape management system
- the review and adoption of system administration tools provided by third-party software suppliers
- a rewrite by a technical writer to provide a smooth finish to the text.

If the cookbook proves popular it will be worthwhile to consider marketing the text to a wider audience. This will involve reconsideration of the CiTEC specific parts for a general audience, and decoupling the cookbook from its reliance on the *UNIX Systems Administration* book as a companion text.

9. Conclusion

The cookbook/toolkit approach is aimed at encouraging the introduction of UNIX as a standard processing environment for departmental systems. The selection of UNIX hardware on term contracts provides for departmental processing at a much lower cost to the government than would be the case if each department were left to do the task by itself.

The approach adopted by Queensland government to the implementation of distributed processing has been followed with interest by computing departments in other state governments. Response within the Queensland government has been positive.

10. Acknowledgments

CiTEC and CiTR would like to thank the assistance of the Computer Science Department at the University of Queensland for sharing their UNIX expertise and knowledge.

11. References

[1] CiTR (1987), *Management Issues in the Use of the UNIX Operating System*, Project Report
[2] CiTR (1987), *Technical Issues in the Use of the UNIX Operating System*, Project Report
[3] CiTR (1987), *Higher Level Communications Architectures- A Comparative Study*, Project Report
[4] Unisys and CiTR (1987), *Open System Services - A Feasibility Study*, Project Report

Appendix A

Introduction

1. Selection of UNIX Systems
   - 1.1 Introduction
   - 1.2 Identification of Requirements
   - 1.3 State Stores (Term) Contracts

2. Preparation for System Delivery
   - 2.1 Introduction
   - 2.2 Staffing
   - 2.3 Training
   - 2.4 Pre-Installation Planning
   - 2.5 Site Preparation

3. System Management Fundamentals

4. Installing a UNIX System
   - 4.1 Installation Overview
   - 4.2 When the System First Arrives
   - 4.3 Configuring Disks
   - 4.4 Adding New Devices
   - 4.5 CITEC Administration Tools
   - 4.6 Acceptance Testing
   - 4.7 Creating User Accounts
   - 4.8 Third Party Software

5. Managing Your UNIX System
   - 5.1 Fundamentals of UNIX System Management
   - 5.2 Tasks Which Should be Done Regularly
   - 5.3 How to Start and Stop UNIX
   - 5.4 Managing User Accounts
   - 5.5 Managing the File System
   - 5.6 System Breakdowns
   - 5.7 Protecting the System
   - 5.8 Managing System Services
   - 5.9 How to Get the Most Out of UNIX
   - 5.10 Software Development Tools

Bibliography

Appendix A. Sample Systems Standards Documents
   - A.1 Installation Overview
   - A.2 Naming Standards
   - A.3 Security Standards
   - A.4 Operations Standards
   - A.5 User Registration
   - A.6 Printers
   - A.7 Sample User Registration Forms

Appendix B. Installation Timetable

Appendix C. Machine Specific Features

Appendix D. Description of CITEC Utilities

Time Synchronisation on a Local Area Network.

Frank Crawford

Q.H. Tours, PO Box 630, North Sydney 2060
(frank@teti.qhtours.oz)

and

Jagoda Crawford

Australian Nuclear Science and Technology Organisation, Private Mailbag 1, Menai 2234
(jc@atom.oz)

ABSTRACT

Given a number of machines on a Local Area Network (LAN), it is often desirable to keep their times synchronised, for such purposes as consistent timestamps, synchronising distributed processes or even just consistent responses to the system time command.

This paper gives details of an implementation developed at Q.H.Tours on an Ethernet based LAN linking four Unix systems. It discusses the algorithms and their underlying theory, together with results observed for this implementation. Some ancillary programs are also outlined.

One other area that is addressed is that of *fault tolerance*, for example, during system startup, when some hosts are unavailable or when the variation is excessive. The approaches taken in this implementation and their limitations will be given.

1. INTRODUCTION

One of the simplest requests that can be made of a computer system is to display the current time. When the system is part of a local area network (LAN) it is reasonable to expect that it would make no difference on which machine such a request is made. Unfortunately this is not the case. The task of keeping the time synchronised across a LAN is a non-trivial one, and facilities are not generally available to implement these requirement.

Aside from the aesthetic value of getting the same response from all machines it is often necessary for a number of operations across the network, for example, for consistent timestamps or to synchronise remote procedures.

As an example consider a remotely mounted file system under UNIX™. If the user executes the *stat* system call on a file in this filesystem should the time returned be correct for the machine it is physically mounted on or the machine remotely mounted on.

To overcome this problem a number of different methods have been proposed, the most obvious of which is to keep the times on each machine on the LAN synchronised. Other methods include calculating the time differences between machines and adjusting any time related information appropriately. In this paper an implementation of a time synchronisation method across all machines on a LAN is described.

™ UNIX is a registered trademark of AT&T in the USA and other countries.

2. *TIME SYNCHRONISATION*

The simple statement that time should be synchronised across the LAN involves a number of design decisions. These include:

- Is the time absolute or relative, *i.e.* is there an external standard that can be referred to, or is it calculated only from the machines on the network.
- Selecting valid times as opposed to those that are faulty, *e.g.* the original time was set wrong.
- Excluding machines that become unavailable to the rest of the network *e.g.* crash.
- Selecting algorithms for calculating and adjusting times.
- Ensuring that time is monotonically increasing.

A number of methods have been studied on synchronisation of time across both local and wide area networks. These include studies by Marzullo and Owicki [1983] and Mills [1985a]. Aside from these a number of implementations for different systems exist, including Mills [1985b] and Gusella *et al.* [1986a, 1986b] for UNIX systems. These and other implementations are all based on different design decisions, some of which are mentioned above.

3. *THEORETICAL BACKGROUND*

The basis of time synchronisation is the selection of a suitable algorithm to calculate a best estimate from a sample of "mutually suspicious" clocks, *i.e.* no one clock can be guaranteed to be correct. This is a common statistical problem and a number of alternative algorithms have been devised to calculate the best estimate.

A basic assumption for the algorithms discussed in this paper is that the majority of *good* clocks display errors clustered around a zero offset relative to a standard time, while the remaining *bad* clocks display errors distributed randomly over the observing interval. The problem is to select the good clocks and to estimate the correction to apply to each clock to set the best estimate of the time. The algorithms described here use *maximum-likelihood* techniques.

It should be noted that the algorithms discussed can result in errors if the sample distribution departs far from the a-priori assumptions. The theory behind the algorithms discussed here is given in Marzullo and Owicki [1983].

3.1 *Majority Subset Algorithm*

An intuitively obvious algorithm that is suitable for a small number of clocks is that of the majority subset. This procedure is as follows:

Given a set of readings:
$S=\{x_1,x_2,...x_n\}$

Best time estimate = $\overline{clock}$

where

for $i=\left\lfloor \frac{n}{2} \right\rfloor+1$ to n
  for j=1 to C_i^n
    if var($S_{i,j}$)< min(*variance*)
      $\overline{clock}=\overline{S}_{i,j}$, where $S_{i,j}$ is the jth subset of S of size i.

Unfortunately the number of calculations required is $O(n!)$, where n is the number of hosts, and thus this method becomes impractical for greater than about 10 hosts. For example when four machines (n=4) are available on the network, the number of combinations to be considered is $C_3^4+C_4^4$ and are listed in Table 1.

TABLE 1. Majority Subsets for n=4

Subset	Elements
$S_{3,1}$	1 2 3
$S_{3,2}$	1 2 4
$S_{3,3}$	1 3 4
$S_{3,4}$	2 3 4
$S_{4,1}$	1 2 3 4

If it is known that some clocks are more accurate than others then a variation of this algorithm can be obtained by introducing a weighting function to each reading.

3.2 Clustering Algorithms

When a large number of hosts are to be sampled the clustering method, shown below, is more suitable. This relies on the fact that there is a majority of good clocks clustered around the correct time. For a large number of samples this provides a good estimate and requires far less computation than the majority subset algorithm.

Given a set of readings:
$S=\{x_1,x_2,...x_n\}$

Best time estimate = $\overline{clock}$

where

for i=1 to $n-1$
$S=S-x_j$, for max($|\bar{S}-x_j|$)

$\overline{clock}=x_j$, the remaining element in S

4. IMPLEMENTATION

A time synchronisation program, *timed*, selecting the most appropriate of the algorithms given above (based on the number of hosts responding), has been implemented over the network at Q.H.Tours. This network comprises four Pyramid systems running OSx™ connected via an Ethernet™ LAN. OSx is Pyramid's version of UNIX, which combine both AT&T's System V.2 and 4.2 BSD, with some features from 4.3 BSD. The networking package is one of these 4.3 BSD features.

™ OSx is a trademark of Pyramid Technology.

™ Ethernet is a trademark of Xerox Corporation.

4.1 Guidelines

During the design of this package a number of assumptions were made and guidelines were established. Some include:

— Assumptions:

- The minimum time adjustment is 1 sec, *i.e.* even though adjustments to milliseconds is possible, this was ignored.
- The propagation time of information across the network is negligible.

— Restrictions:

- Time must be monotonically non-decreasing (in reality it should be monotonically increasing but this was not possible due to the minimum time adjustment of 1 sec).
- No machine can be guaranteed to be more accurate than any other.
- Maximum time adjust set to 1 sec per 10 second interval, except for setting the time when the machine is first booted.

4.2 Details

The 4.3 BSD networking package includes a number of trivial services which are implemented within the *inet* daemon, a process designed to wait on network calls and pass them to the appropriate program. The trivial services implemented directly by *inet* include:

- time - the current time in seconds since midnight, 1st January, 1900[1] in a machine readable form.
- daytime - the current date and time in a human readable form, *e.g.* "Sun Jul 24 11:58:10 1988".
- chargen - character generator for diagnostics.
- echo - echo back everything sent to it.
- discard - throw away everything sent to it.

Of these, the most useful for time synchronisation is the *time* service. With this it is simple to query each host on the network and then calculate the current best estimate of the time. In order to utilise this service the user only needs to establish a connection with the remote machine. The current time is immediately returned as a 4 byte quantity (a 32 bit integer in network order) and the connection is immediately closed (by the remote host).

As this time is only accurate to within a second it satisfies the criterion that the propagation time is negligible with respect to the actual time. To obtain a better estimate for the time on a remote host, the average of a number of readings is taken.

The time estimates used in this implementation are the differences between the responding host's time and the requesting host's time. It can be shown that the same results are obtained by using these differences in the estimation algorithms. This overcomes the problem of having to make all requests at the same time (which is impossible). It also gives a more convenient value for time adjustment.

This time estimate is used only to give the adjustment direction (advance, retard or no change). If any change is necessary then the current system time is adjusted by one second (except on boot, as described below). The synchronisation daemon then pauses for a period dependent on whether any adjustment was necessary and then repeats the entire process.

---

1. This is different from UNIX which is seconds from 1st January *1970*.

The reason for the difference in this period is that if an adjustment is made then it is likely that further adjustments are required so a very short period is set, whereas if no adjustment is made then a change is unlikely in the short term[2].

4.3 Fault Tolerance

As with any system that is affected by a number of separate parts, any of which can exhibit errors (and further where the system is expected to correct these errors) it has been necessary to include a degree of fault tolerance. The ultimate responsibility of any fault tolerant system falls on a human to correct major inconsistencies. In this implementation an effort has been made to report such situations so that they can be corrected, without being overwhelming.

Faults reported include all time adjustments that are greater than a preset limit (currently the delay to the next check, if there was no adjustment), and refusing to do adjustments if the difference is more than a day (most likely caused by an incorrect setting by the operator). It also includes tracking which machines are not responding and when they resume responding. An effort is made to handle time adjustments when the machine is first booted, however without a source independent of the LAN it is impossible to be sure that it is correct.

The problem with time correction at startup is that there is no prior knowledge of the other machines. It is possible that only one machine is being rebooted, so the time on the others is still correct, or that all machines are being rebooted (*e.g.* after a power failure) so none are correct. In an effort to minimise errors the current implementation adjusts the time by the total adjustment value at boot if it is positive but makes no adjustment if it is negative (*i.e.* time could only progress while the machine was down).

4.4 Other Programs

Although the time synchronisation daemon is self contained, there are two related programs that would be useful. These are:

- A method to modify internal parameters of *timed*, *e.g.* the delay between time checks. Also it would be useful to be able to request some internal statistics.
- A program to set the time simultaneously on all the machines on the network (*i.e.* a network version of *date yymmddhhmm.ss*).

Both of these have not yet been implemented but it would be trivial to do so.

5. EXPERIENCES

Time synchronisation daemons have been running on all the machines at Q.H. Tours since mid-May and have, in general, lived up to expectations. When first started they brought the times on all machines into line within about 20 mins, and they have remained synchronised ever since. Even so there are some areas that could be improved.

The first of these is that there is some "jiggling" of the time adjustment. The design constraints were that time would only be adjusted to within one second (due to the accuracy of the readings). In practice there are a number of conflicting adjustments over a very short time. For example the synchronisation daemon will first add 1 second, at the next time check subtract 1 second, and so on. This generally only occurs for a short period, and is most likely caused by the various times differing by about half a second, and measurement errors causing slight differences. In fact the current implementation takes the mean of three readings and the differences can often be seen to vary from one third to two thirds of a second.

2. It is assumed that the clocks are reasonably accurate.

A second problem occurs with *cron*, the daemon designed to start other programs at given times. The version supplied within OSx's AT&T universe (and thus presumably with System V.2) appears to sleep between jobs, then start the next job immediately on waking and before checking the current system time. This is assumed because there have been occurrences of jobs being started twice, once early and then at the correct time. This problem was seen prior to the implementation of *timed*, especially during periods of low activity, *e.g.* the middle of the night.

A final point about this program is that it is probably only suited to an environment consisting of a small number of machines. The effect of having each machine on the network requesting the time of every other machine on the network would cause considerable network congestion on a large network. However for a small network it is both simple and adequate. It should also be noted in this implementation, that after synchronisation is achieved, time checks are only performed once every 20 mins.

6. CONCLUSION

The implementation of time synchronisation at Q.H. Tours turned out to be simple and resulted in a reliable system. It has been running since May with no major problems and has solved the original problems of synchronising all the host's clocks to within 1 second.

Although there are some minor additions that can be made, especially in terms of ancillary programs, it is now effectively complete and has been virtually relegated to the background and forgotten about like most daemons.

7. REFERENCES

Marzullo, K. and Owicki, S. [1983] - "Maintaining the Time in a Distributed System", *Operating Systems Review*, Vol. 19, no. 3, pp. 44-54.

Mills, D.L. [1985a] - "Algorithms for Synchronizing Network Clocks", *DARPA Network Working Group Report RFC-956.*

Mills, D.L. [1985a] - "Experiments in Network Clock Synchronization", *DARPA Network Working Group Report RFC-957.*

Gusella, R., Zatti, S. and Bloom, J.M. [1986a] - "The Berkeley Unix Time Synchronization Protocol", *Unix Programmer's Manual 4.3 Berkeley Software Distribution*, Vol 2c.

Gusella, R., Zatti, S., Bloom, J.M. and Smith K. [1986b] - "Time Installation and Operation Guide", *Unix Programmer's Manual 4.3 Berkeley Software Distribution*, Vol 2c.

On MICROLAN 2 - An Office Network

Richard Lai
ICL Australia Ltd (Melbourne)
ACSnet : lai@iclmlb.oz

Abstract

MICROLAN 2 provides a neat way of connecting several ICL's Distributed Resource System (DRS) 300s and workstations together. DRS300 is a micro system running under DRS/NX, ICL's port of UNIX* System V.2. There is one trunk cable from a DRS300 and the workstations and other DRS300s are multi-dropped to this main cable.

MICROLAN 2 is commonly used to set up a small office network. The cable can take some convenient routes through the office, for example around the walls. At variuos points, connection boxes for processors and workstations are fitted to the cable. Should machine be moved to another location later, it can be easilly plugged into another connection box.

This paper describes the implementation and design of MICROLAN 2.

1. INTRODUCTION

Where more than two processors are to be connected into a small network, a MICROLAN 2 cable can be used. There is a MICROLAN 2 port on the front of the DRS300 processor. The cable that is fitted to this port is normally used to connect the processor to a number of its workstations. At points on the cable there are connection boxes to receive a lead from a workstation. However, these connection boxes will also take a lead from the MICROLAN 2 port of another DRS 300 processor, allowing up to four processors to communicate across the MICROLAN 2 cable which may be up to a kilometre long.

A MICROLAN 2 cable is used to set up a small office network in which the cable itself takes some convenient route through the office, for examlple round the walls. At various points connection boxes for processors and workstations are fitted to the cable. Since the potential length is 1000 metres, the processors and workstations that are plugged into the connection boxes can be set quite far apart and yet have instantaneous communications. There may be more boxes than processors and workstations

*UNIX is a Trademark of AT&T.

which, when in need, can be simply plugged in the boxes in the desired locations. By placing connection boxes throughout a building, computing resources can be redistributed according to the changing requirements.

The protocol used is High-level Data Link Control (HDLC). The protocol includes comprehensive error detection and recovery for transmission. The UNIX commands "cu" and "uucp" can be used to communicate with remote systems connected via MICROLAN 2.

The system is resilient against failures of DRS300 machines. It is not dependent on a particular DRS300 system present in order for the network to function. It supports up to 4 DRS300s running under DRS/NX with up to 4 workstations on each DRS300 participating in the network.

2. USER PERSPECTIVE

Interfaces to other DRS300s connected via a MICROLAN 2 network are provided through special files. It takes the form :

/dev/netxdy

where x is the Small Computer System Interface (SCSI) address of the DRS300 processor being talked to and y is a number between 0 and 7 that agrees for the two processes that are communicating.

The procedures to set up the network are :

a) All the DRS300s on the network are to run at the same speed. This is configured by the 'hdlcspeed' parameter in the definition file.

b) Each of the DRS300s has a unique node name and SCSI address.

c) Up to four workstations attached to each DRS300 can take part in the network. They use special files in the range /dev/tty50 to 53. The MICROLAN 2 address of each workstation taking part in the network is to be set accordingly.

If the workstation is accessed using /dev/ttyN, (where N is 50 to 53), the MICROLAN 2 address is as follows :

MICROLAN 2 Address (decimal) = Processor SCSI address * 4 + (N - 50)

d) The system is rebooted.

e) The utility 'netbuild' is run at each of the DRS300s to create the required network special files and to edit the appropriate files to complete the network setup.

3. INTERFACE

The RS485 interconnect medium is to connect videos and processors to a DRS300. This is the mulitdrop version of RS422, which defines particular signal characteristics for an electrically balanced network. The specification for the DRS300 MICROLAN 2 network includes opto-isolation, which is not addressed in the RS485 standard.

MICROLAN 2 uses a multi-drop, half duplex, 2 wire cable. The link is via the HDLC interface socket on the front of the processor. It uses the MICROLAN 2 multi-drop line. There is one trunk cable from the processor and the devices are connected by spurs to the main cable.

MICROLAN 2 workstation driver is responsible for driving workstations and DRS300s all multi-dropped from a single HDLC line. It uses a simple round-robin algorithm. The devices are polled at approximately 20 times a second. On each poll cycle a buffer of data can be sent and received from each device. The line speeds are fixed when the line is being driven. The line speed is configurable from 153 Kbits per second to 307 Kbits per second. There tends to be a higher error rate at 307 Kbits per second.

Details of RS422 and RS485 are described in [2] and [3].

4. A NETWORK EXAMPLE

To illustrate the networking concept, an example giving the physical connection is shown in figure 1. Processors 1, 2, 3 and 4 having SCSI addresses 1, 2, 3 and 4 respectively and eight videos are networked together via MICROLAN 2. Processor 1 controls videos 1 and 4; processors 2, videos 6 and 7; processor 3, videos 2, 3 and 5; and processor 4, video 8.

Applying the rule of 2(c) and assigning /dev/ttyN in increasing order from 50, the addresses of the eight videos are :

Video	MICROLAN 2 Address (decimal)
1	4
2	12
3	13
4	5
5	14
6	8
7	9
8	16

ACTUAL MICROLAN 2 WIRING

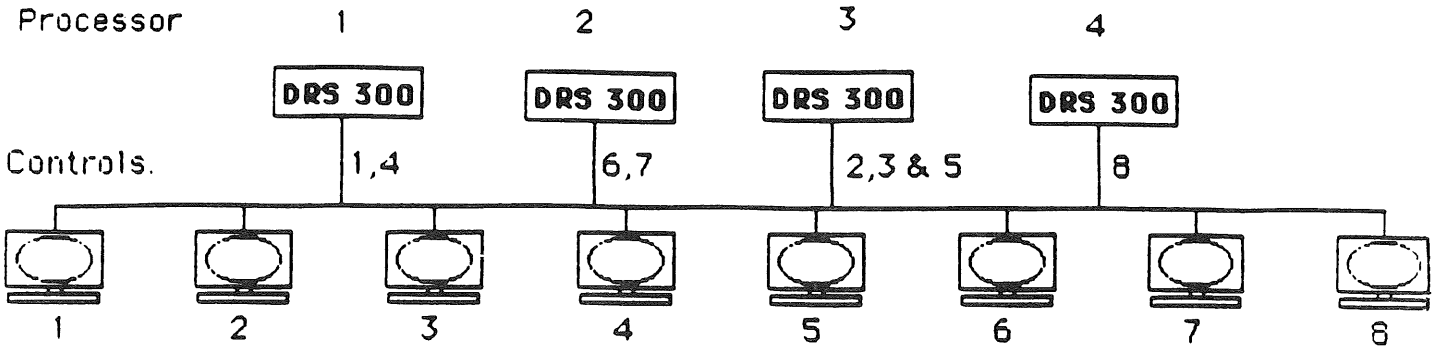


figure 1

CONCEPTUAL SYSTEM APPEARANCE

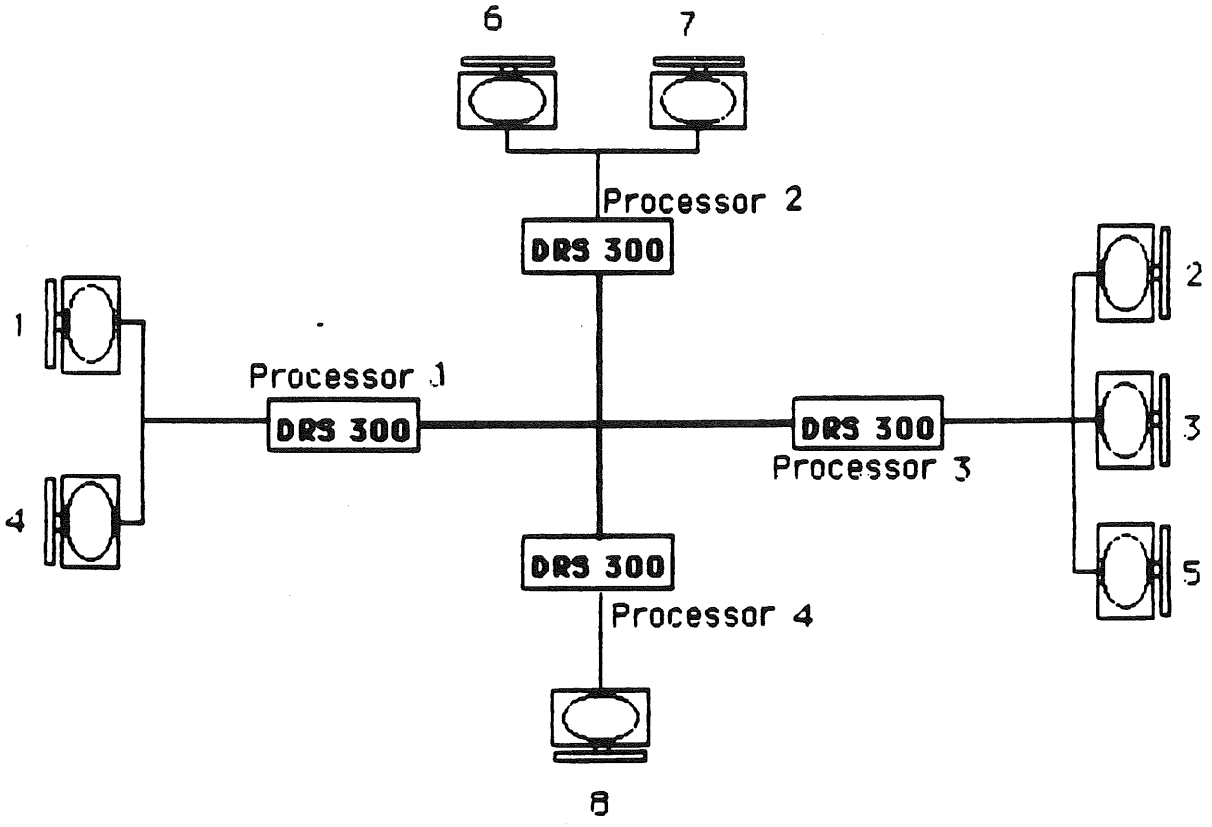


figure 2

A conceptual system appearance, illustrating the relationships of processors and videos and the interworking, is shown in figure 2.

5. LINK LEVEL PROTOCOL

The protocol used is High Level Data Link Control (HDLC). It provides serial bit by bit transmission over the MICROLAN 2 cable. It is a two-way alternate (half-duplex) multipoint (multidropped) protocol. It provides error detection and recovery procedures.

On an HDLC link, there is a primary device and a number of secondary devices. The primary is the DRS300 and has responsibility for controlling the link. The primary devices issue commands. The secondaries receive commands and return responses. The primary controls the use of the link by secondaries using a sending address to select a secondary and sending a poll flag to allow the secondary to transmit. Data is transmitted between the primary and secondaries. Data is not transmitted between secondaries.

A detailed description of HDLC is given in [1].

6. ROTATING MASTER SYSTEM

To cater for machines that crash, newly loaded machines that arrive on the network, starting up on an inactive network, etc, rotating master and resilience techniques are employed. No special role is given to any particular machine.

Master is the machine responsible for polling at a particular instant. Rotating master system is a scheme to allow sharing of the polling load among different machines on a network and to provide resilience in the event that some machines go offline. In the rotating master system, the machines on the network are regarded as forming a ring. Broadly speaking, each machine becomes master in turn. During its turn as master, a machine may deal with traffic for its own videos and send network traffic. When it has finished its turn, it passes the master to the next machine in the notional ring. The message used to pass on the master is called a 'relinquish'. Unlike other messages, no reply is given to a relinquish.

Video traffic is to take precedence over network traffic. The following rules are applied :

a) No network traffic may be sent if the timer used to control the interval between video polls has expired;

b) A limit is imposed on the length of a single transmission of network traffic, so that if the polling timer expires while network transfer is in progress, polling is not held up significantly.

To ensure each machine with network traffic to send gets an opportunity to send it, a 'network reservation byte' is included in the relinquish message used to pass the master on.

7. DEAD MACHINE

When a machine passes on the master, it gets confirmation that the network is healthy when it becomes master again, but has no way of detecting that some intermediate machine on the ring has crashed other than setting a timer. This timer is called the 'watchdog timer', and every machine keeps one It is used to detect that nothing has been heard for a particular length of time. There are two values used to start this timer : startup value, used to wait when a machine first comes online, and initial value, used to wait after a machine has passed on the master.

It is started every time the master is passed on. The timer is switched off when the machine becomes master, and is reset to its initial value if any other sort of message is received. For instance, network traffic may be sent to the machine from another machine when that machine is master.

If the timer expires, the machine should assume the role of master. Other machines may do the same at approximately the same time. There is a strong probability that all remaining machines will each assume the role of master within a short time of each other, since circulation of master is comparatively quick. The mechanisms for dealing with network establishment and duplicate masters are dealt with later.

The initial value of the watchdog timer is chosen to be long enough so that, with reasonable loads on each machine in the ring, the master completes its circulation within the timeout period. Conversely, it is short enough so that after any contention caused by duplicate masters is resolved, keyboard response does not suffer unduly.

A machine is thought to be dead if it fails to respond to network traffic sent to it. It is not offered the master again until it gives an indication that it is alive again. This is done by sending a message called 'network pair initialise' to the dead machine at a low rate compared with that at which master is passed on, but faster than the watchdog timer initial value. Only when a response is received is the master passed on to the now healthy machine.

8. NETWORK ESTABLISHMENT

When first switeched on, each machine supposes that it is joining a 'live' network, and would therefore expect to receive the master or a network pair initialise. It therefore sets its watchdog timer to its startup value and waits. If another machine establishes contact with it, eventually passing it the master, the new machine should find the first live machine that it can pass the master to.

If the new machine's timer expires, it must assume the role of master. It may be that other machines switched on at the same time are doing exactly the same thing. It is necessary to ensure one and only one master will emerge and will do so in a resonable time. A scheme has been devised to resolve this contention.

Directed broadcast is a message with a secondary address which includes within it a primary address and another secondary address. Only the machine with that secondary address should send a response to the primary address. When a machine assumes the role of master, it sends out directed broadcast versions of network initialises at fixed intervals. The intervals are different for each machine. The directed broadcast invites a particular machine to respond. If it does, the machine assumes that other machines heard the broadcast, implying that it is now the one and only master. It therefore carries on as normal.

If a message other than the response to the directed broadcast arrives during this interval, it knows that another machine has assumed the role of master, and so it should give up the role itself.

9. DUPLICATE MASTERS

Duplicate masters can arise exceptionally because contention has taken place and been resloved, when a machine that thinks it is master did not hear about the contention, or at least its resolution. Collisions of messages occur as a result of duplicate masters.

Any error that occurs unexpectedly is treated as a potential collision. The action to be taken is to inform any duplicate masters that they should give up being master. This is done by sending a relinquish in directed broadcast form to the next machine thought to be responding. If a duplicate master received the relinquish, because it was waiting before passing on the master, or waiting for a response to a video or network device message, it would cease to be master, thus solving the problem. If it did not receive the relinquish , it would either be because of a genuine error, or because it was in the process of transmitting itself. The most likely outcome in this case would

be that the relinquish collided with whatever the duplicate master was transmitting, thus killing off the oringinal master, again solving the problem.

If two relinquishes clash with each other, then both masters they represent get killed off, and contention would be used to get things started again.

10. CONCLUSIONS

MICROLAN 2 allows interworking among several DRS300 systems, and allows all the workstations assoicated with these machines to coexist on a single MICROLAN 2 network. The system is resilient against failures of DRS300 machines. It is not dependent on a particular DRS300 to be present in order for the network to function.

The system has much enhancement potential in exploiting the sharing of machines and videos on the same network, allowing for example, dynamic reconfiguration of videos among the machines. More sophisticated file transfer facility and remote session access can be implemented besides 'cu' and 'uucp'.

References

1. ISO 3309 Data Communication - High Level Data Link Control Procedures - Frame Structure, International Organisation for Standardisation.

2. RS - 422 Electrical Characteristics of Balanced Voltage Digital Interface Circuits, Electronics Industries Assoication.

3. RS - 485 Electrical Characteristics of Generators and Receivers for Use in Balanced Digital Multipoint Systems, Electronics Industries Assoication.

4. Ritchie, D.M. and Thompson, K., "The UNIX Time Sharing System", Communication of the ACM, pp365-375, 1974.

5. System V Interface Definition, AT&T, 1985.

6. X/Open Portability Guide, X/Open, 1985.

7. Halsall, F., Data Communication, Computer Networks and OSI, Addison-Wesley, 1988.

Programming on UNIX System X release Y

Stephen Frede

Softway Pty Ltd

1. intro

This paper is meant to provide a useful reference guide to anyone writing or porting programs for versions BSD version 4 release 2 (BSD4.2), BSD version 4 release 3 (BSD4.3), AT&T System V (SysV), AT&T System V release 2 (SysV.2) and AT&T System V release 3 (SysV.3). When porting programs, the usual problem is to port BSD programs to SysV, rather than vice-versa.

Note that this is a guide only, and readers should refer to the appropriate manual entries for relevant details. Also note that is is very incomplete and correspondence will certainly be entered into. Sections marked TODO will be implemented in forthcoming releases of this paper.

2. Common changes

To many people, the most common porting problem they have to deal with is to get programs from Usenet to run on their systems. In many cases, only a few changes need to be made. The most common of these are listed here for convenience. Refer to the individual descriptions given later in the paper for more details.

SysV	**BSD**
strchr( )	index( )
strrchr( )	rindex( )
#include <string.h>	#include <strings.h>
memcpy( )	bcopy( )
srand48( )	srandom( )
lrand48( )	random( )
uname( )	gethostname( )

3. System calls and library routines

All of the system calls and some of the non-compatible library routines are listed here in alphabetical order, so that they can be found easily. Descriptive parameter names are provided for some but not all of the routines as an aid to comparison more than anything else. Anything given as "pathname" is a string which refers to a relative or absolute pathname in the filesystem. The argument fd is an integer file descriptor.

accept( ) (BSD) See *Sockets*

access(pathname, mode) (any)

BSD provides definitions in <sys/file.h> not provided in SysV.

```
#if    BSD
#    include <sys/file.h>
#endif /* BSD */

#ifndef R_OK
#   define   R_OK   4
#   define   W_OK   2
#   define   X_OK   1
#   define   F_OK   0
#endif /* R_OK */
```

acct(pathname) (any)
Use is compatible. However, there are minor differences in the structure stored in the accounting file. SysV has two extra fields in the `acct` structure: `comp_t ac_rw` (no. block read/writes) and `char ac_stat` (exit status). Also the `comp_t ac_io` means no. disk I/O blocks under BSD and chars transferred by read/write under SysV. Finally, the `ac_comm` field is 10 bytes long in BSD, and 8 bytes in SysV.

adjtime(delta, olddelta) (BSD) See *Time*
This system call is used to adjust the system clock by temporarily speeding it up or slowing it down, so that the clock always gives increasing values. No comparable functionality in SysV.

alarm(secs) (any)
This SysV system call is implemented as a library routine under BSD.

bind( ) (BSD) See *Sockets*
brk(address) (any)
chdir(pathname) (any) *Compatible all versions*
chmod(pathname, mode) (any) *Compatible all versions*
chown(pathname, owner, group)
Use is compatible. Under BSD, only the super-user may change the owner of a file, while under SysV, anyone may "give away" files they already own.

chroot(pathname) (any) *Compatible all versions*
close(fd) (any) *Compatible all versions*
closedir(dirp) (BSD, SysV.3, PD) See *Directory*
connect( ) (BSD) See *Sockets*
creat(pathname, mode) (any) *Compatible all versions*
dup(fd) (any) *Compatible all versions*
dup2(oldfd, newfd) (BSD)
The BSD system call:

```
newfd = dup2(oldfd, wantedfd);
```

is equivalent to the following under either BSD or SysV

```
#include      <fcntl.h>
...
close(wantedfd);
newfd = fcntl(oldfd, F_DUPFD, wantedfd);
```

environ (any) *Compatible all versions*
execl( ) (any) See *exec*
execle( ) (any) See *exec*
execlp( ) (any) See *exec*
exect(name, argv, envp) (BSD)
Used when tracing (with *ptrace(2)*) the executed process.

execv( ) (any) See *exec*
execve( ) (any) See *exec*
execvp( ) (any) See *exec*
exit(status) (any) *Compatible all versions*
_exit(status) (any) *Compatible all versions*
f...(fd, ...)
A number of system calls exist starting with the letter "f", which correspond in function to a system call with the same name without the leading "f". The "f" system calls take a file descriptor as their first parameter. The corresponding system calls take a pathname as first parameter. Of these, fstat( ) is available on all versions, but fchmod( ), fchown( ) and ftruncate( ) are BSD specific. To emulate these under SysV, you will need to keep the pathname of the file available and use the corresponding "non-f" system call.

fchmod(fd, mode) (BSD) See *chmod( )*
fchown(fd, owner, group) (BSD) See *chown( )*
fcntl(fd, cmd, arg) (any)
Use is compatible if the `cmd` argument is `F_DUPFD`, `F_GETFD`, or `F_SETFD`. BSD has the extra `cmd` arguments `F_GETOWN` and `F_SETOWN` for dealing with job control, which is not applicable under SysV. SysV has the extra `cmd` arguments `F_GETLK`, `F_SETLK` and `F_SETLKW` which deal with file locking (see the section on File Locking).

The `cmd` arguments `F_GETFL` and `F_SETFL` are used in the same way under both systems, but the file status flags being accessed are different. Under SysV, the status values `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_NDELAY` and `O_APPEND` may be accessed. Under BSD, `FAPPEND` corresponds directly with `O_APPEND` under SysV, and in fact uses `O_APPEND` as a flag for *open( )* (so you can't just #define it). The BSD flag `FASYNC` is used with job control – there is no equivalent functionality under SysV. The SysV.2 flag `O_SYNC` is used to ensure that data is physically updated on disk after every *write( )*. Under BSD, calling *fsync( )* after every *write( )* can be used to achieve the same effect.

The SysV `O_NDELAY` flag affects reads and writes on pipes and reads from tty devices: a read from an empty pipe or a tty device with no information available, which would otherwise block, returns with 0. The BSD `FNDELAY` flag affects reads and writes on sockets and tty devices: an operation which would otherwise block returns an error and sets `errno` to `EWOULDBLOCK`.

flock(fd, operation) (BSD) See *Locking*
fork( ) (any) *Compatible all versions*
fstat(fd, statbufp) (any) See *stat( )*
fstatfs(fd, buf, len, fstyp) (SysV.3) See *statfs( )*
fsync(fd) (BSD)
The effect of a *write( )* followed by an *fsync( )* can be duplicated under SysV.2 by setting the `O_SYNC` flag on the file descriptor prior to the *write( )*.

ftruncate(fd, length) (BSD) See *truncate( )*
getdents(fd, buf, nbytes) (SysV.3) See *Directory*
This system call is designed to read directory entries in a filesystem independant format. The *directory(3)* routines should generally be used in preference to using this routine directly.

getdtablesize( ) (BSD)
This gives the maximum no. of open file descriptors available to a process. Under SysV, use `NOFILE` defined in `<sys/param.h>`. The information is only available at compile time under SysV.

getgid( ), getegid( ), getuid( ), geteuid( ) (any) See *Access permissions*
The only difference in these functions between versions is the return value. Under SysV, they are int. Under SysV.3, they are unsigned short. Under BSD, they are `gid_t` and `uid_t`, as defined in `<sys/types.h>`.

getgroups(gidsetlen, gidset) (BSD) See *Access permissions*
Used to get group access list under BSD. Under SysV just use *getegid( )* to get the single value group id that is used for all access permission checks.

gethostid( ) (BSD)
Designed to get a unique number (as set by sethostid( )) for networking purposes. Under SysV, read this from a file.

gethostname(name, namelen) (BSD)
Designed to get the standard host name for this machine. Under SysV, this information is provided by the *uname( )* system call.

```
#if    BSD
#   include   <sys/param.h>
    ...
    static char                 hostbuf[MAXHOSTNAMELEN + 1];
```

```
    ...
    hostname = gethostname(hostbuf, sizeof hostbuf) == -1 ? 0 : hostbuf;
#else  /* BSD */
#   include   <sys/utsname.h>
    ...
    static struct utsname  names;
    ...
    hostname = uname(&names) == -1 ? names.nodename : 0;
#endif /* BSD */
```

Under SysV the name of the system (as obtained by *uname( )*) is configured into the kernel and cannot be changed at runtime. An alternate approach to emulating `gethostname( )` and `sethostname( )` is to read and write the name to and from a known file. This approach has the problem that other programs using `uname( )` will obtain possibly different information. It is rare that a user program would need to change the system name except at boot time.

getitimer(which, value) (BSD) See *Time*
getmsg( ) (SysV.3) See *Streams*
getpagesize( ) (BSD)

The value returned by this routine is the system page size. Under SysV this is often defined as `NBPP` (number of bytes per page) in either `<sys/param.h>` or `<sys/immu.h>`, but is rarely required. On BSD it is the granularity of memory allocated by *sbrk( )*.

getpeername( ) (BSD) See *Sockets*
getpgrp( ) (any)

Under BSD, this system calls takes a pid parameter and returns the process group id of the nominated process group. This capability is not available under SysV – the function call does not take any parameters and returns the process group of the current process.

getpid( ) (any) *Compatible all versions*
getppid( ) (any) *Compatible all versions*
getpriority(which, who) (BSD)

Used to get the scheduling priority for a process, process group, or user. SysV can only provide information about the current process.

```
#if    BSD
#   include   <sys/resource.h>
       ...
       errno = 0;
       prio = getpriority(PRIO_PROCESS, 0);
#else  /* BSD */
       errno = 0;
       prio = nice(0);
#endif /* BSD */
```

getrlimit( ) (BSD)

Used to determine what current limits exist on resource usage for the resources process cpu time, maximum file size able to be created, maximum process data area size, maximum process stack size, maximum size core file to be created, maximum resident set size (ie amount of physical memory being used). Similarly *setrlimit( )* is used to set these values. Under BSD both a hard and soft limit may be specified for consumption of these resources. The process may be notified (eg via a signal) when a soft limit is reached, and will be denied the resource when a hard limit is reached.

The only equivalent functionality in SysV is provided by the *ulimit( )* system call, which allows a process to examine and set the file size limit, and to get the maximum possible data area size. There is no soft limit capability.

```
        long   currfilemax, newfilemax, currdatamax;
#if     BSD
#    include   <sys/time.h>
#    include   <sys/resource.h>
        struct rlimit rl;
        ...
        getrlimit(RLIMIT_FSIZE, &rl);
        currfilemax = (long) rl.rlim_max;
        ...
        rl.rlim_max = (int) newfilemax;
        setrlimit(RLIMIT_FSIZE, &rl);
        ...
        getrlimit(RLIMIT_DATA, &rl);
        currdatamax = (long) rl.rlim_max;
#else  /* BSD */
        long   ulimit( );
        ...
        currfilemax = ulimit(1, 0L) * 512;
        ...
        ulimit(2, (newfilemax + 511) / 512);
        ...
        currdatamax = ulimit(3);
#endif /* BSD */
```

getrusage( ) (BSD)
The system call allows a process to enquire about the utilisation of various resources by itself or its children. There is no equivalent functionality available directly through the system call interface under SysV. If such information is required, system accounting may be able to provide it. A process may directly examine the process accounting file, or use the output of the commands *sar(1)*, *sar(1M)*, *timex(1)*, *acctcom(1)*. Probably the most useful approach is to use *timex(1)* to run the command in question and use the resulting output information.

getsockname( ) (BSD) See *Sockets*
getsockopt( ) (BSD) See *Sockets*
gettimeofday(tp, tzp) (BSD) See *Time*
index(str, ch) (BSD)
Equivalent under SysV to *strchr(str, ch)*.

ioctl(fd, request, argp) (any) See *Tty*
The actual use of this command is compatible amongst all versions, though the possible values of the parameters vary widely. This system call is used to perform device specific actions on various devices. The only device where a modicum of standardisation exists is the tty interface.

kill(pid, sig) (any) *Compatible all versions*
killpg(pgrp, sig) (BSD)
This system call can be replaced with the *kill( )* system call under any version.

```
#if     BSD_ONLY
        status = killpg(pgrp, sig);
#else
        status = kill(-pgrp, sig);
#endif
```

link( ) (any) *Compatible all versions*
listen( ) (BSD) See *Sockets*

lseek(fd, offset, whence) (any) *Compatible all versions*

BSD defines tokens for the *whence* argument of this call.

```
#ifdef BSD
#   include  <sys/file.h>
#endif /* BSD */

#ifndef       L_SET
#   define    L_SET  0
#   define    L_INCR 1
#   define    L_XTND 2
#endif L_SET
```

lstat(pathname, statbufp) (BSD) See *Symbolic links, stat( )*

Like stat( ), but provides information about symbolic links, instead of the file referred to by the link. No equivalent in SysV.

mkdir(pathname, mode) (BSD, SysV.3)

This system call was introduced into SysV with release 3. For a non-privileged process, the only alternative is to execute the setuid root *mkdir(1)* command. A privileged process could duplicate the code in *mkdir(1)* to create the directory with *mknod(2)*, but this is not recommended.

```
#if    SysV && SysVver < 3
#include      <fcntl.h>
mkdir(path, mode)
char   *path;
int    mode;
{
    int             pid,
                wstat;
    switch(pid = fork( ))
    {
       case -1:
           return -1;
       case 0:              /* child */
           close(2);
           fd = open("/dev/null", O_WRONLY);
           if (fd != 2)
              fcntl(fd, F_DUPFD, 2);
           execl("/bin/mkdir", "mkdir", path, 0);
           _exit(1);
    }
    /* parent */
    while ((s = wait(&wstat)) != pid && s != -1)
       ;
    if (s != pid)
       return -1;
    if (wstat != 0)
       return -1;
    umask(mask = umask(0));
    mode &= ~mask;
    return chmod(path, mode);
}
#endif
```

mknod( ) (any) *Compatible all versions*

mount( ) (any)

BSD is compatible with SysV up to release 2. SysV.3 is incompatible. The following code handles the simplest case, where the file system being mounted is the same type as the root filesystem.

```
        char    *special, *pathname;
        int     rwflag;
#ifndef         SYSV3
        mount(special, pathname, rwflag);
#else   /* SYSV3 */
        mount(special, pathname, rwflag == 0 ? 0 : 1, 0);
#endif /* SYSV3 */
```

msgctl( ) (SysV) See *Message Queues*

msgget( ) (SysV) See *Message Queues*

msgrcv( ) (SysV) See *Message Queues*

msgsnd( ) (SysV) See *Message Queues*

nice( )

The SysV system call is compatible with the BSD library routine, except that no value is returned by the BSD version.

open(pathname, flags, mode) (any)

For SysV, flag values are defined in `<fcntl.h>`; for BSD they are defined in `<sys/file.h>`. See *fcntl( )* for details on the flag values.

opendir(filename) (BSD, SysV.3, PD) See *Directory*

pause( ) (any) *Compatible all versions*

pipe( ) (any) *Compatible all versions*

plock(op) (SysV)

This system call allows a process to lock (or unlock) its text and data segments in memory. This means that they are immune to all normal swapping. There is no functional equivalent under BSD.

poll( ) (SysV.3) See *Streams*

profil(buff, bufsiz, offset, scale) (any)

These are compatible except for the interpretation of the scale argument. A 1:1 mapping of pc to words in buff is specified by a scale of 0x10000 under BSD and 0xFFFF under SysV.

ptrace( ) (any)

Similar but differences.

```
#if     BSD
#   include   <sys/signal.h>
#   include   <sys/ptrace.h>
#endif /* BSD */

#ifndef         /* PT_TRACE_ME */
#   define    PT_TRACE_ME   0
#   define    PT_READ_I     1
#   define    PT_READ_D     2
#   define    PT_READ_U     3
#   define    PT_WRITE_I    4
#   define    PT_WRITE_D    5
#   define    PT_WRITE_U    6
#   define    PT_CONTINUE   7
#   define    PT_KILL             8
#   define    PT_STEP             9
#endif /* PT_TRACE_ME */
```

putmsg( ) (SysV.3) See *Streams*

quota( ) (BSD)

The *quota( )* and *setquota( )* system calls are used to implement per-user disk quotas on BSD systems. There is no equivalent functionality available under SysV.

read( ) (any) *Compatible all versions*

readlink( ) (BSD) See *Symbolic links*

readdir(dirp) (BSD, SysV.3, PD) See *Directory*

readv(fd, iov, iovcnt) (BSD)

```
struct iovec /* defined on BSD in <sys/uio.h> */
{
    char      *iov_base;
    int             iov_len;
};

readv(fd, iov, iovcnt)
int           fd;
struct iovec  *iov;
int           iovcnt;
{
    struct iovec    *iovt;
    int                   n,
                    nread;
    char            *buf,
                    *bp,
                    *malloc( );

    n = 0;
    /* count up no. bytes to read */
    for (i = 0, iovt = iov; i < iovcnt; i++, iovt++)
       n += iovt->iov_len;
    if (!(buf = malloc((unsigned) n)))
       return -1;
    if ((nread = read(fd, buf, n)) == -1)
    {
       free(buf);
       return -1;
    }
    n = nread;
    bp = buf;
    for (i = 0, iovt = iov; i < iovcnt && n; i++, iovt++)
    {
       int    nx;
       nx = iovt->iov_len;
       if (nx > n)
           nx = n;
       memcpy(iovt->iov_base, bp, nx);
       bp += nx;
       n -= nx;
    }
    free(bp);
    return nread;
}
```

reboot( ) (BSD)
This system call is used to shutdown and possibly reboot the system. It may be sensibly invoked in two possible ways, and these can be emulated under SysV.3 versions.

```
#if    BSD
#   include   <sys/reboot.h>
        ...
        reboot(RB_HALT);     /* halt; no reboot */
        reboot(RB_AUTOBOOT);       /* halt; then reboot multiuser */
#else  /* BSD */
#   include   <sys/uadmin.h>
        ...
        uadmin(A_SHUTDOWN, AD_HALT, 0);   /* immediate halt; no reboot */
        uadmin(A_SHUTDOWN, AD_BOOT, 0);   /* immediate halt; reboot multiuser */
#endif /* BSD */
```

The 2nd parameter of *uadmin( )* could also be `A_REBOOT`, indicating that the system shutdown is immediate, rather than after killing all user processes, flushing the buffer cache and unmounting the root filesystem. Also note that the 3rd parameter of *uadmin( )* has a machine dependant function.

recv( ) (BSD) See *Sockets*
recvfrom( ) (BSD) See *Sockets*
recvmsg( ) (BSD) See *Sockets*
rename( ) (BSD)
rewinddir(dirp) (BSD, SysV.3, PD) See *Directory*
rindex(str, ch) (BSD)
Equivalent under SysV to *strrchr(str, ch)*.

rmdir(pathname) (BSD, SysV.3) See *mkdir( )*
sbrk(incr) (any)
seekdir(dirp, loc) (BSD, SysV.3, PD) See *Directory*
select( ) (BSD) See *Sockets*
semctl( ) (SysV) See *Semaphores*
semget( ) (SysV) See *Semaphores*
semop( ) (SysV) See *Semaphores*
send( ) (BSD) See *Sockets*
sendto( ) (BSD) See *Sockets*
sendmsg( ) (BSD) See *Sockets*
seteuid(euid), setegid(egid) (BSD) See *setreuid( ), setregid( )*
These are library routines on BSD which call the *setreuid( )* and *setregid( )* system calls.

setgroups(ngroups, gidset) (BSD) See *getgroups( ), Access permissions*
Under SysV, use *setgid( )*.

sethostid( ) (BSD) See *gethostid( )*
sethostname( ) (BSD) See *gethostname( )*
setitimer( ) (BSD) See *getitimer( ), Time*
setpgrp( ) (BSD, SysV) See *getpgrp( )*
The SysV call

```
    status = setpgrp( );
```

is equivalent to the BSD code

```
    status = getpid( );
    if (setpgrp(0, status) == -1)
       status = -1;
```

Under BSD, this call can be used to alter the process group of another process. There is no SysV mechanism to do this.

setpriority( ) (BSD) See *getpriority( )*
Used to set the scheduling priority for a process, process group, or user. SysV can only set the priority of the current process.

```
#if     BSD
#    include   <sys/resource.h>
        int     pri;
        ...
        setpriority(PRIO_PROCESS, 0, pri);
#else   /* BSD */
        int     pri, incr;
        incr = pri - nice(0);       /* calculate increment required */
        errno = 0;
        nice(incr);
#endif /* BSD */
```

setquota( ) (BSD) See *quota( )*
setregid(rgid, egid), setreuid(ruid, euid) (BSD) See *Access permissions*
These system calls are used to change the real and effective uid and gid of the current process. If a parameter is given as –1, the current value is used (ie no change is requested). Under SysV, there is no capability for a non super user process to change its real uid or gid, and no capability for a super user process to change its real uid or gid independant of the effective uid or gid.

setrgid(rgid), setruid(ruid) (BSD) See *setreuid( ), setregid( )*
These BSD library routines use the *setreuid( )* and *setregid( )* system calls.

setrlimit(resource, rlp) (BSD) See *getrlimit( )*
setsockopt( ) (BSD) See *getsockopt( ), Sockets*
settimeofday(tp, tzp) (BSD) See *gettimeofday( ), Time*
setuid(uid), setgid(gid) (any) See *Access permissions*
shmat(shmid, shmaddr, shmflg) (SysV) See *Shared Memory*
Attach an already existing shared memory segment to the current process.

shmctl(shmid, cmd, buf) (SysV) See *Shared Memory*
Perform various operations on a shared memory segment.

shmdt(shmaddr) (SysV) See *Shared Memory*
Detach a shared memory segment from the current process.

shmget(key, size, shmflag) (SysV) See *Shared Memory*
Create a shared memory segment.

sigblock(mask) (BSD) See *Signals*
sigmask(signum) (BSD) See *Signals*
This is a macro defined in <signal.h> on BSD systems which is used to construct a signal mask from a signal number.

sigpause(sigmask) (BSD) See *Signals*
TODO

sigreturn(scp) (BSD4.3) See *Signals*
TODO

sigset(mask) (SysV.3) See *Signals*
sigsetmask(mask) (BSD) See *Signals*
TODO

sigstack(ss, oss) (BSD) See *Signals*
TODO

sigvec(sig, vec, ovec) (BSD) See *Signals*

socket( ) (BSD) See *Sockets*

socketpair( ) (BSD) See *Sockets*

stat(pathname, statbufp) (any)

Used in the same manner on any version, but the stat structure varies between SysV and BSD. Fields in common are listed below. The types are defined in `<sys/types.h>` and the actual names may vary from those given here (eg `ushort` in SysV is `u_short` in BSD).

```
dev_t  st_dev;          /* dev on which inode resides */
ino_t  st_ino;          /* inode number */
ushort st_mode;     /* file mode */
dev_t  st_rdev;     /* dev of this inode (if applicable) */
short  st_nlink;    /* no. links */
ushort st_uid;          /* uid of file owner (short on BSD) */
ushort st_gid;          /* gid of file group (short on BSD) */
off_t  st_size;     /* file size in bytes */
time_t st_atime;    /* time of last access */
time_t st_mtime;    /* time of last data modification */
time_t st_ctime;    /* time of last inode modification */
```

Additionally, BSD provides the fields:

```
long   st_blksize;  /* optimal blocksize for i/o */
long   st_blocks;   /* no. blocks actually allocated */
```

statfs(pathname, buf, len, fstyp) (SysV.3)

This system call is used to retrieve generic information about a filesystem. It replaces the *ustat( )* system call used before SysV.3 (though SysV.3 also supports *ustat( )*). Its introduction is because of the filesystem switch introduced with SysV.3 – reading the filesystem superblock directly is no longer a practicable alternative for the many different possible filesystem types. To obtain information about a filesystem under BSD, or more information than is provided by *ustat( )* under SysV, a process has to open the filesystem directly and read the superblock.

stime(timep) (SysV) See *Time*

This system call is used to set the system's idea of time of day. Under BSD, *settimeofday( )* can be used instead.

strchr(str, ch) (SysV)

This system call is equivalent on BSD to *index(str, ch)*.

strrchr(str, ch) (SysV)

This system call is equivalent on BSD to *rindex(str, ch)*.

swapon( ) (BSD)

Used under BSD to add a swap device for interleaved paging/swapping. No equivalent functionality under generic SysV. However, some implementations of SysV provide an implementation specific means of doing this. For example on the 3b series running SysV.2.1 and up, the *sys3b( )* system call with *cmd* parameter `S3BSWPI` can be used to add or delete swapping areas.

symlink( ) (BSD) See *Symbolic links*

Used to create a symbolic link to a file. No equivalent in SysV.

sync( ) (any) *Compatible all versions*

sys3b(cmd, arg1, arg2, arg3) (SysV)

This system call is machine dependant. Other implementations of SysV (and indeed BSD) may provide similar machine specific system calls to handle tasks not otherwise provided by the release. Typical uses of this sort of system call are to operate lights on the machine, provide access to internal system tables, access or change kernel configuration parameters, access or change boot information, access or change swap areas, access or change information stored in non-volatile RAM, etc.

syscall( )
Indirect system call. This is very system dependant, and is not intended to be used by C programs.

sysfs( ) (SysV.3)
This system call is used to determine the type of a filesystem.

telldir(dirp) (BSD, SysV.3, PD) See *Directory*
time( ) (any) See *Time*
times( ) (SysV) See *Time*
truncate(path, length) (BSD)
The system calls *truncate( )* and *ftruncate( )* are used to reduce a file to a particular length. Under SysV, trucation of a file is only possible to 0 length.

uadmin( ) (SysV) See *reboot( )*
ulimit( ) (SysV) See *getrlimit( )*
umask(mask) (any) *Compatible all versions*
umount(path) (any) *Compatible all versions*
unlink( ) (any) *Compatible all versions*
ustat( ) (SysV) See *statfs( )*
utime(pathname, times) (SysV) See *utimes( )*
utimes(pathname, tvp) (BSD)
Under SysV, one only needs write permission on a file to change the times. On BSD, only the owner (or super-user) may do this. In both cases, the inode change time is set to the current time, and in neither case can the inode change time be set to any arbitrary value.

```
long    atime, mtime; /* new access and modification times in seconds */
#if     BSD
#   include   <sys/time.h>
        struct timeval tvp[2];
        ...
        tvp[0].tv_sec = atime;
        tvp[1].tv_sec = mtime
        utimes(file, tvp);
#else   /* BSD */
        struct utimbuf      tbuf;
        ...
        tbuf.actime = atime;
        tbuf.modtime = mtime;
        utime(file, &tbuf);
#endif /* BSD */
```

vfork( ) (BSD)
This system call is provided for efficiency only where the child process intends to immediately exit or call *exec( )*, and can be replaced in SysV with *fork( )*. If you are writing portable code and intend to use *vfork( )* if the code is running on a BSD system, you should read the manual entry carefully before doing so.

vhangup( ) (BSD)
This system call causes a number of actions to take place, basically disassociating the current process's control terminal from any other references. TODO

wait(statusp) (any)
Although the definition of the parameter under BSD is a pointer to a union wait, rather than to an int under SysV, the usage and meaning is generally compatible.

wait3(status, options, rusage) (BSD)
This system call allows a parent to collect more detailed information about its children, and to optionally avoid hanging.

write(fd, buf, nbytes)	(any)	*Compatible all versions*
writev(fd, iov, iovcnt)	(BSD)	See *readv( )*

4. curses

System V introduced the terminfo terminal description database scheme to replace the previous termcap database, which is still in use by BSD. At the same time, a new version of the curses library was introduced. Fortunately it is largely compatible with the old version, but there are a number of differences. The most common of these incompatibilities is given below. The best way to write code which distinguishes the two versions is to #ifdef on a token like `A_REVERSE` which is defined in `<curses.h>` in the terminfo version, but not in the termcap version.

Note that BSD programs that use termcap routines only are compiled with -ltermcap, while those that use curses routines must be compiled with -lcurses -ltermcap. All SysV programs that use any of these routines are compiled with -lcurses only.

```
#ifdef A_REVERSE
cbreak( )
nocbreak( )
attron(x)
attroff(x)
wattron(win, x)
wattroff(win, x)
beep( )
flash( )
```

```
#ifndef A_REVERSE
crmode( )
nocrmode( )
standout( )
standend( )
wstandout(win)
wstandend(win)
putchar(' 07')
{
    if(BP)
       _puts(VB);
    else
       putchar(' 07');
}
```

```
saveterm( )
resetterm( )
erasechar( )
```

```
savetty( )
resetty( )
ioctl(o, TIOCGETP, &sg), sg.sg_erase
```

5. Directory

The traditional directory structure, and that used in SysV looks like:

```
struct direct
{
        ino_t  d_ino;
        char   d_name[DIRSIZ];
};
```

where `ino_t` is normally a 2 byte quantity and `DIRSIZ` is always 14. This is defined in `<sys/dir.h>` on SysV systems (although it will not be in SysV.4). BSD4.2 introduced long filenames, up to `MAXNAMLEN` (`<sys/dir.h>`) bytes in length. In order to access this new directory structure in a relatively simple and portable fashion, the *directory(3)* library routines were introduced. These should be used wherever directories are to be read. They are in the standard C library in BSD systems, and many vendors provide them for SysV systems, either in the standard C library or in an alternate library (eg try compiling with -lndir). Additionally, there are public domain versions freely available for SysV and other operating systems (such as MS-DOS). Also, SysV.3 has these routines provided as standard.

If the world were a perfect place, this would be all there was to it – just use these routines, and tack a PD implementation of them onto software for systems that don't already have them. Unfortunately, the world not being a perfect place, there is more to the story.

When BSD implemented long directory names, they chose the same structure name in the same include file (`struct direct` in `<sys/dir.h>` to refer to the new directory structure supported by the system,

and the `readdir( )` routine returns a pointer to one of these structures. This gives compatibility routines on other systems two alternatives. They could either retain maximum compatibility with BSD, in which case programs using them could not `#include <sys/dir.h>`; or they could opt to change the name of the directory structure returned by *readdir( )*. While routines are available to do the former, the IEEE 1003.1 and the SVID (and various other PD implementations) have opted for the latter approach.

The implication is that you need to handle both BSD-compatible and POSIX-compatible (inc. SysV.3) cases. The following code can be used as a guide:

```
#if    BSD_DIR
#   if        BSD    /* BSD 4.2 */
#      include       <sys/dir.h>
#   endif     /* BSD */
#   if        V9     /* Bell Edition 9, and some others */
#      include       <ndir.h>
#   endif     /* V9 */
#   if        BRL    /* BRL System V emulation */
#      include       <dir.h>
#   endif     /* BRL */
#   define    dirent direct
#else  /* BSD_DIR */
                     /* SysV.3 or library compatible with same */
#   include  <sys/dirent.h>
#endif /* BSD_DIR */
```

Just in case you thought things weren't too bad, the BSD struct direct is different from the SysV.3 struct dirent. Fields in common are long d_ino (the inode number), short d_reclen (the length of this record rounded up), and d_name (the actual filename, which can be treated as type char * though the actual declaration varies). Most importantly, d_namlen does not appear in the SysV.3 structure, so portable programs should use strlen(...->d_name) instead. Both provide `MAXNAMLEN` as the maximum possible name length.

That's it as far as most programs are concerned. But I should mention the SysV.3 *getdents( )* system call. It has nothing to do with parking your car in the street, but is designed to read directory entries in a file system independent format. The impetus for this system call is the famous filesystem switch implemented in SysV.3; *getdents( )* can be used to read directory entries regardless of the actual type of the filesystem. The *readdir( )* routine uses this system call and should be used in preference to using this call directly (in fact the manual entry explicitly states that this call should not be used for other purposes).

6. Exec

Note that BSD allows files which begin with "#! interpreter", to be executed directly, while SysV allows only a.out style files to be executed directly. This means that an instance of one of the *exec...( )* routines in a BSD program may have to be changed under SysV to execute the interpreter (typically /bin/sh) for that program instead. For example if `/usr/bin/thing` is a shell script:

```
#if    BSD
       execl("/usr/bin/thing", "thing", "it", 0);
#else  /* BSD */
       execl("/bin/sh", "/usr/bin/thing", "thing", "it", 0);
#endif /* BSD */
```

7. Time

TODO

8. Sockets

Under BSD, sockets are used as the primary IPC mechanism. A socket is described as "an endpoint for communication between processes", with each socket having queues for sending and receiving data. It has been said that sockets are a good idea done badly. BSD has 18 system calls, 25 error numbers and 14 related library routines to implement sockets. Porting a BSD program which uses sockets to SysV will usually require major changes to the code, and probably sigificant design changes. The following list is purely to enable you to identify a BSD program which uses sockets.

System calls: *accept( ), bind( ), connect( ), getpeername( ), getsockname( ), getsockopt( ), listen( ), recv( ), recvfrom( ), recvmsg( ), select( ), send( ), sendmsg( ), sendto( ), setsockopt( ), shutdown( ), socket( ), socketpair( ).* Related library routines: *endhostent( ), endprotoent( ), gethostbyaddr( ), gethostbyname( ), gethostent( ), getprotobyname( ), getprotobynumber( ), getprotoent( ), res_comp( ), res_expand( ), res_init( ), res_mkquery( ), res_send( ), sethostent( ), setprotoent( ).*

9. Tty

TODO

10. Semaphores

TODO

11. Message Queues

TODO

12. Streams

A very nice idea implemented in SysV.3, which would be even more relevant if tty devices had been implemented using streams, which in SysV.3 they are not.
TODO

13. Signals

TODO

14. Symbolic links

BSD supports the concept of symbolic links. A symbolic link is a file in the filesystem which refers to another file. An attempt to open the symbolic link results in a file descriptor referring to the file to which the link is pointing. Similarly a *stat( )* system call and most other system calls which act on files will affect the referenced file rather than the actual symbolic link. An *unlink( )* or *rename( )* system call will affect the symbolic link itself. A new *lstat( )* system call has been introduced to access information about the symbolic link itself.

15. MAUS

This acronym stands for Multiple Access User Space. Some implementations of SysV up to SysV.2 had a set of system calls known under this acronym to implement shared memory. These should normally be avoided and the more standard (though still only SysV) shared memory operations used instead.

16. Shared Memory

SysV supports shared memory through the use of the *shmget( ), shmat( ), shmctl( )* and *shmdt( )* system calls.

17. Access permissions

Every BSD process has a set of group ids used for file access permission checking, as well as the real and effective gid used by SysV. The *getgroups( )* and *setgroups( )* system calls are used to access and change this set. When any file access is attempted, the gid on the file is compared with all the gids in the list. Under SysV, group file access is determined solely by the effective group id of the process.

Under SysV, the gid of a file when it is created is set to the effective gid of the process that created it. Under BSD, the gid of the file is set to the gid of the directory in which the file is beaing created.

Under both BSD and SysV, a process may change its effective uid to match its real uid. Under BSD only, a process may also change its real uid to match its effective uid. The same is true of group ids.

Note that when a process has an effective uid set by executing a setuid program, this effective uid is saved seprately from the current effective uid, so that if the effective uid is changed by the process back to the real uid, it may be changed back to the saved effective uid. This is true for both SysV and BSD, and the same applies for group ids.

An undocumented but deliberate feature of SysV only is that a process cannot change its effective uid to its real uid, if that real uid is the super-user.

The sticky bit has a special meaning when applied to a directory in BSD4.3. Where this is the case, in order to unlink or rename a file in that directory, as well as having write permission on the directory a user must own the file or directory (or be the super-user).

18. Include files

TODO

A study about the implementation of real-time disk I/O scheduling on UNIX operating system

J.B. Lee*, H.J. Kim*, K.W. Rim*, K.O. Park*
* ETRI Dae Dog Dan-Ji P.O. Box 8, Chungnam, Korea
Y.J. Park**
** Hang-Yang univ., Seoul, Korea

ABSTRACT

To use UNIX system in the real-time environment, many real-time supporting features such as preemptive process scheduling, contiguous file system, modular kernel, memory locking mechanism and the other features must be added to the operating system. Although these features are implemented in the UNIX operating system, real-time disk I/O scheduling mechanism is required to use applications efficiently in the real-time system where the disk I/Os are used frequently.

In this paper, the real-time scheduling mechanism is proposed. The proposed mechanism was implemented and tested to provide faster disk I/O for the real-time process than the general time sharing process on the UNIX operating system. A buffer mechanism and a disk I/O scheduling algorithm were modified for the real-time disk I/O scheduling.

1. Introduction

Requirements in computer application areas becomes more complicated as computers are used more widly. In particular, the real-time system should process external events generated in application areas and should reply for the external events within a limited time depending on the specific usage of the system to satisfy those various requirements. Typical areas for the real-time systems are data acquisition, transaction processing, process control and factory automation area. In general, a structure of the real-time system depends on the environment of applications because the real-time criteria depends on the environment of the application area.

The UNIX operating system has been used originally in the general computer system. Therefore, UNIX operating system was not suitable for real-time applications naturally. But, many real-time UNIX products such as Masscomp RTU(Real Time Unix), HP-UX of HP company, MERT and DMERT of 3B20D system and etc. were developed to satisfy these real-time requirements. A basic idea to support the real-time process is that general time sharing process should not be scheduled to run while the real-time process are executing. So the real-time process can be performed right after the general time sharing process is executed while the other general time sharing process are waiting for the CPU. This is a preemptive process scheduling mechanism. But, if the system included this feature can not execute real-time process in a limited time, another features such as a contiguous file system, a modular kernel, a memory locking mechanism and the other features should be optionally added to finish the real-time process in a limited time.

In this paper, we propose a real-time disk I/O scheduling mechanism to execute disk I/Os generated by real-time process faster than by the general time sharing process. It can be used in the real-time applications efficiently where disk I/Os are frequently used. And it was implemented and tested on the KONIX(Korean UNIX) operating system of th SSM-16 developed in ETRI.

2. Real-time disk I/O scheduling

2.1 Development Background

Real-time disk I/O scheduling was implemented on the real-time UNIX with a preemptive process scheduling alone. Process scheduler of the system can preempt disk I/Os generated by the general time

sharing process that are waiting for CPU for the service, and the scheduler passes the control to real-time process prior to general time sharing process. If the real-time process generates disk I/Os, it blocks itself on the sleep state while disk I/Os is processed by the disk driver. Disk I/O requests which are generated by the general time sharing process are put into the request queue and are sorted by its cylinder number. If a disk I/O generated by the real-time process is treated in the same manner, the real-time process has to sleep until all preceding requests and real-time disk I/Os are completed. Therefore this mechanism drops the efficiency of the real-time processing.

But the real-time disk I/O scheduling mechanism can provide to serve real-time disk I/O requests directly after the request being executed is served while the other early arrived disk I/O requests is waiting. So this mechanism increases the efficiency of the real-time process.

2.2 The Method of Development

As disk I/O requests are generated by a process, the system examines whether wanted disk blocks are already existing in a block I/O buffer cache of the memory. In case that the blocks exist in the buffer cache, the system will return them from the buffer cache without physical disk I/O. If they do not exist in the buffer cache, the system allocates the buffer and fills a buffer header with appropriate contents The system calls a disk driver to request blocks from a disk. And "disksort" function in a disk driver is called for disk seek scheduling. The function puts I/O requests in the request queue sorted by its cylinder number to minimize the overall disk seek time.

To implement real-time disk I/O scheduling, real-time relevant informations are included in the buffer header. The buffer header is used to identify whether the disk I/O is generated by real-time process or not. There are two methods to give the real-time concerned informations to the buffer header. First method is one using special real-time system calls such as r_read() and r_write(). The real-time concerned informations are recorded in the buffer header through these system calls. Major advantage of this method is that the system can execute only required disk I/Os as real-time disk I/Os among disk I/O requests generated by the real-time process.

Major disadvantages of this method are that the system can hardly determine the sequence among real-time disk I/O requests. It means that to implement priority based real-time disk I/O scheduling is difficult. And new system calls should be made for this method.

The second method is to record those informations in the process table when the real-time process being created. Then, the system can identify easily whether disk I/O requests generated are from real-time process or not by reading process table. The major advantage of this method is the flexibility to be able to determine the processing sequence even among the real-time disk I/O requests. Because the priority based real-time process scheduling can be implemented, the disk I/Os generated by real-time process can be easily implemented in the same manner. And major disadvantage will be that all disk I/Os generated by the real-time process are handled as real-time disk I/Os. As a result, that method can possibly hamper other more urgent disk request.

3. Implementation Issues

The second method mentioned above was adopted to schedule real-time disk I/O requests according to priorities of their real-time process in this implementation. And this method is designed to provide user level compatibility, so it does not make special system calls for a real-time read and a real-time write system call. As it was mentioned above, process table is used to provide the real-time concerned informations.

3.1 the implementation of system call for generating real-time process

First of all, header file "proc.h" was modified to represent the real-time process.

```
#define SREAL 0200      /* real-time process */
                        /* to define p_flag in proc table */
```

And a system call to generate the real-time process is implemented as followings.

```
/*
 * make real-time process
 */

realtime()
{
      register struct proc *pp;
      register struct a {
            int   flag;
      } *uap;
      if (!suser()) { /* permitted to super user only */
            u.u_error = EPERM;
            return;
      }
      ...
      uap = (struct a *)u.u_ap;
      pp = u.u_procp;
      if(uap->flag)
            pp->p_flag |= SREAL;
      else
            pp->p_flag |= ~SREAL;

        ...

}
```

Also, a realtime() system call is registered in the system call entry, and library interface (libc/sys/realtime.a68) for the system call is made.

3.2 Buffer header modifications

The flags and variables are added to represent real-time disk I/O as followings. And various flags can be defined more to schedule real-time disk I/Os based on its priority.

```
struct buf {
      short b_flag;
      short b_rtios;  /* real-time i/o flag */
      ...
}

/* b_rtios flags */
#define B_TSIO    00; /* general time sharing */
#define B_RTIO    01; /* real-time */
```

3.3 System I/O process routine modifications

The system I/O process routine(bio.c) for block I/O is modified to identify whether the requested process is real-time process or not by reading the real-time concerned informations in the process table. If real-time process orders disk I/O, the system sets the flag(b_rtios) in the assigned buffer header to real-time(B_RTIO). The disk I/O scheduling routine(dsort.c) of a disk driver schedules by referencing this information. When the system I/O process routine releases the buffer used, the flag is reset to time

sharing(B_TSIO) again.

3.4 Implementation of real-time disk I/O scheduling routine

A logical queue for real-time disk I/O requests before the other logical queue for the other disk I/O requests is placed in a request queue. Disk I/O requests are sorted according to their cylinder number in each logical request queue. By this mechanism, the real-time disk I/O requests are executed early compared to the other disk I/O requests.

4. Performance evaluation

4.1 Evaluation method

We evaluated real-time disk I/O scheduling with self-developed test program without testing tool proper to this test. This test was performed by executing test programs for real-time process and general time-sharing process respectively at the same system environments. That is, it is the evaluation method to analyze advanced response time of real-time process by comparing response times. We analyzed the performance and response time for each process by running following test program.

```
main()     /* test program for real-time process */
{
   get start time; /* s_real_time */
   while(count) {
      do real-time disk i/o; /* rtio */
   }
   get terminating time; /* t_real_time */
   calculate elapsed time;
      /* (rtio)t_real_time - s_real_time */
}

main()     /* test program for general time sharing process */
{
   get start time; /* s_ts_time */
   while(count) {
      do time-sharing disk i/o; /* tsio */
   }
   get terminating time; /* t_ts_time */
   calculate elapsed time;
      /* tsio(t_ts_time - s_ts_time) */

}
```

A test was performed to measure response times at five different phases, which were made at our convenience, at the same system environments. Five phases were divided according to disk load level.

Phase 1 : No load
Phase 2 : Light load (only vi executed)
Phase 3 : Middle load (vi and INGRES.demo executed)
Phase 4 : Light heavy load (vi and two INGRES.demo executed)
Phase 5 : Heavy load (vi, two INGRES.demo and cat *.c)

As a result of executing test programs in each phase, the following table was obtained.

load level	tsio response time	rtio response time	evaluation (tsio-rtio)
Phase 1	1min 36sec 918	1min 32sec 015	4sec 913
Phase 2	1min 40sec 080	1min 33sec 445	6sec 635
Phase 3	2min 00sec 825	1min 49sec 745	11sec 080
Phase 4	2min 09sec 405	1min 57sec 425	11sec 980
Phase 5	2min 11sec 800	1min 58sec 735	13sec 065

Table 1. Test result for each Phase

It is easily noticeable from table 1 that the response time for disk I/O has been significantly decreased by the real-time process rather than the general time sharing process.

5. *Conclusions*

Most of important application of the real-time system is a disk I/O related one. Therefore, fast disk I/O for those real-time process is very important to attain the required performance. In this paper, we presented the implementation of real-time disk I/O scheduling. With that, we can improve response time for the real-time process even when the system suffers from the heavy load. Further study area is a development of algorithm for the real-time disk scheduling in order to differenciate from real-time processes, and implementation of more improved disk seek algorithm.

6. *References*

1. H.M. Deitel," An Introduction to Operating System", Chap 12-15, Addison Wesley, 1982

2. H. Lycklama & D.L. Bayer, "The MERT Operating System", The Technical Journal, Vol.57, No.6, p.2049-2085, 7. 1978,

3. B. Look, "Real Time Extensions to the UNIX Operating System", UniForm Conference Proceedings, p.293-299, 1. 1984

4. J. M. Scanlon, "The 3B20D Processor & DMERT Operating System", The Bell System Technical Journal, 1. 1983, p.167-179

5. W.N. Toy & L.E. Gallaher, "The 3B20D Processor & DMERT Operting System", The Bell System Technical Journal, 1. 1983, p.181-190

6. Maurice. J. Bach,"The Design of the UNIX Operating System", Prentice-Hall Inc.

7. Stephen Evanczuk,"Real-Time O.S.",Electronics, 3. 1983

8. P. Sherrod & S. Brenner,"Minicomputer System offers Time Sharing and Real Time Tasks",Computer Design, 8. 1984

9. J.B. Lee, S.J. Jang, S.M. Kim, "A study of real-time UNIX", ETRI, TD88-1640-72, 1. 1988

The Process Life Cycle in STIX

Sunil K Das & Aarron Gull
City University London
Computer Science Department
EC1V 0HB UK

Abstract

STIX is a port of the MINIX operating system onto the Atari ST micro-computer. This paper discusses first the experiences gained by the authors in creating a cross-development system using a Gould super mini-computer as host and an Atari as target. Following this is a description of STIX processes and their implementation, and the associated system calls for process creation and termination. The primary interest in this discussion concentrates upon how the authors overcame the problems presented by the lack of hardware memory management within the Atari.

1 STIX - Project Overview

The MINIX operating system [Tanenbaum, 1987a] is a rewrite of the seventh edition UNIX system, frequently referred to as Version 7 or V7, which runs on the IBM PC, XT or AT. The system calls available to the MINIX user are compatible with those of V7. For users of IBM micro-computers, MINIX is a low cost, UNIX-like system which is an alternative to those provided by IBM. Moreover, MINIX is available to teachers and students for use in operating system courses since it is fully documented [Tanenbaum, 1987b] and the source code is available on floppy discs without licencing restrictions.

The City University Computer Science Department in London houses in the order of 100 Atari 520 and 1040 ST micro-computers for use as terminals, and as development devices associated with computing projects and teaching. Thus, the idea of porting MINIX from the IBM PC to the Atari ST range of micro-computers [Gerits; 1988] was conceived and called the STIX project.

There are inherent difficulties in porting MINIX to the ST. A significant problem was selecting and adapting a suitable C compiler which could be used for the port. Tanenbaum supplies a C compiler with MINIX which can be used on the PC to recompile the operating system. The compiler, based on the Amsterdam Compiler Kit [Tanenbaum, 1983], is supplied in binary form only. A mc68000 C compiler was required for the STIX project because the

Intel 8086/8088 is the internal processor for IBM micro-computers whereas the ST is based on the Motorola 68000 processor. Furthermore, it was imperative to have access to the source code of the compiler in order that modifications could be made to it.

Since it is very difficult to obtain source code for a public domain mc68000 C compiler, it was necessary to use a compiler for which the Computer Science Department had a source licence. This compiler could not be used on the ST because of site licencing problems. Thus, it was decided to use the Department's Gould computer as host for development, with the Atari as the micro-computer target, since the Gould's UNIX based operating system UTX/32 was source licenced. In order to use a host-target pair, a cross-development environment was needed comprising a C compiler and associated libraries, which would produce code of a suitable form to down-line load and run on the target ST.

After a brief description of the cross-development environment, this paper discusses the novel software decisions taken to overcome the Atari's lack of hardware memory management. The life cycle of a process is investigated to clarify the effect of STIX on the memory and process managers. The behaviour of the *swapper()* algorithm is examined together with its relationship to the system calls *fork()*, *exec()* and *exit()*. The paper concentrates upon the implementation of these system calls and the effect that relocation has on *exec()*.

2 The Cross-development Environment

A cross-development environment offers several advantages over the traditional method of developing software on the target machine. The host is much faster, more stable, and has greater disk space. Also, it provides access to a range of UNIX tools such as **lint**, **make**, **SCCS** and **RCS**, which are specifically intended for C program development. Moreover, on completion of the system, it is available for the use of others who want a more hospitable ST development environment. For this purpose, it was decided to provide many function calls which would give an interface to the resident ST operating system, **TOS**.

The source code to the compiler and the C library is not supplied with MINIX. The Stanford UNIX Mac C Development Kit (**SUMacC**) [Croft; 1984] is distributed in the public domain under the condition that it may be "used" but not "sold" because it is subject to a Stanford copyright. **SUMacC** version 2.0 became available in November 1984 and was packaged for a VAX computer running either BSD UNIX or Eunice. It was considered possible to adapt the kit to any UNIX box having a mc68000 C compiler. The **SUMacC** based mc68000 cross-development system available to the authors consisted of two compilers, one from Berkeley and one from Stanford, which are merged by the use of conditional compilation statements. The Stanford code was removed and the remaining system adapted for cross-development.

Cc68, the C compiler steering program, had to be modified to reflect the

Berkeley file structure of STIX; the compiler had to know where to find the header files and libraries. The assembler **as68**, had to be modified in order to produce the correct header format required by the linkage loader. The compiler was not distributed with its own C preprocessor **cpp68**, so the public domain Florence EUUG C preprocessor was ported for this purpose. The majority of the work undertaken on the compiler centred around **ld68**, the linkage loader. Apart from having numerous bugs, **ld68** had to be modified to produce the new executable header formats required by STIX.

The source code to the MINIX C library was, at that time, in the public domain but not readily accessible to the authors. Consequently, it was decided to reimplement the C library. Each library routine was decompiled by hand back into its C or mc68000 equivalent. It is important to realise that writing the C library was a significant amount of work; the library is composed of 13,000 lines of C code and 2,000 lines of assembler. Writing the library however, although a long and rewarding academic exercise, is not of particular interest to report.

Once ported to the Department's Gould computer, the cross-development system had to be modified to produce code of a suitable form to run on the target micro-computer. For MINIX on the IBM PC, there exists four segmentation registers so processes can be relocated to run from any place in memory. The Atari ST does not have segmentation registers. Furthermore, it has no memory management hardware at all. Under these circumstances the ideal cross-compiler would produce *executable* code which is *relocatable*. Thus, when STIX either forked or execed a process, the program text would be loadable into any available memory, and executable no matter where the segment had been located in user memory. It is unlikely, due to some of the operations possible in C, that such a compiler exists. One was not known to the authors.

The strategy adopted for process creation in the STIX environment was to retain the *fork()* and *exec()* system calls, and to incorporate a **swapper()** to interchange the address spaces in user memory of two processes. Substantial revisions would have to be made to the memory management code to control this movement of processes. This approach has the advantage of retaining the Version 7 interface, but the disadvantage of adding complexity to the STIX kernel and memory manager.

A detailed description of the cross-development system [Gull; 1988] is available from the authors.

3 Process Management

The composition of the STIX KERNEL is very similar to that of MINIX; the KERNEL contains four modules: the kernel, the memory manager, the file system and the init process. The kernel image is composed of the process manager, the system task and the device drivers for the terminal, printer, memory, disk and clock.

STIX processes are very similar to V7 processes. A process is the image of a program in execution and has a number of associated resources: text, data, bss and stack segments allocated from memory; a set of arguments and an environment list; the processor registers; a number of open files; the concept of a current working directory; and a group and user identifier.

A STIX process is the entity that is created by the *fork()* system call; every process, except those of the KERNEL, is created when another process executes a *fork()*.

3.1 The Memory Map

The MINIX memory manager neither supports hardware paging nor process swapping onto a swap device. MINIX processes are never swapped out and never relocated to another place in user memory. Once user memory has been allocated to a MINIX process, it remains there until its termination. The memory region allocated to a MINIX process remains constant in size during its life time. The memory manager and the kernel's process manager in MINIX use the following structure in their process tables, giving both access to the **virtual** and **physical** addresses:

```
struct mem_map {
    unsigned mem_vir;       /* virtual  address */
    unsigned mem_phys;      /* physical address */
    unsigned mem_len;       /* length */
};
```

In STIX, these two fields have different names and are used for different purposes. The process manager uses an *execution* and *current* address, while the memory manager uses an *execution* and *creation* address, in their corresponding process table structures. The **execution**, **current** and **creation** addresses of a process are three pointers to the regions in user memory from where the process executes, is currently residing, and to where the process is located on creation by *fork()* or *exec()*. When a process is in execution, its execution and current addresses will be the same; when the process is relocated into some other region of user memory they will differ.

A difficulty occurs when a process tries to give up its memory, for example during the *exec()* or *exit()* system calls. The memory manager needs to know the whereabouts of the process; it does not know, at a give time, where in memory any particular process is located. The current address of a process is known only to the process manager. The problem has been solved by adding the internal system call *sys_swap()* to the kernel. This call returns a given process into its creation region. If another process occupies the creation region, the two are interchanged. The creation address of a process is known to the memory manager and so after the interchange, it is aware of a process' location in memory. Note that the memory manager has no knowledge of this swapping.

An example follows to clarify the new usage of these address fields in the structure *mem_map{}*. Assume process A begins life at address 0x50000 and has a length of 0x10000. Its execution, current and creation addresses are all 0x50000. Process A then forks a child process CA. Process CA also has to run at address 0x50000, therefore its execution address is 0x50000. When process CA is created, the memory manager reserves enough memory to hold the image of its parent process. This area of memory is the creation address of the process. Let us assume that 0x67000 is the start address of the first hole available capable of containing a process of length 0x10000. Since the process CA is located initially at 0x67000, this becomes its current address. When process CA begins execution its current address becomes 0x50000 and the current address of process A becomes 0x67000.

3.2 The Context Switch

The context of a process is its state, as defined by its text, data, user and supervisor stacks, the values of the processor registers, and the values stored in its kernel, memory manager and file system process tables. When executing a process, the system is said to be executing in the context of the process. A context switch is made whenever a process cannot continue to run. When doing a context switch, the kernel saves enough information so that it can later switch back to the process and continue its execution. A context switch will only occur when the processor detects an exception. Most exceptions are caused by the clock interrupting the processor. However, other exceptions are generated by the process itself when it executes code. This is the manner in which system calls are made to the kernel. STIX supports the traditional activities of context saving and restoring.

3.3 The Swapper

The *swapper()* is called whenever a process needs to be restarted. In STIX, we use the term *swapper()* to mean the program which interchanges the address space of two processes which concurrently reside in user memory. MINIX processes are *never* relocated in user memory and *never* swapped out of user memory to a swap device.

Due to the nature of the *fork()* system call and the ST hardware, several processes may exist which have to execute in the same region of memory. The *swapper()* ensures that a process which is about to be restarted is located in the required region of memory.

STIX processes are of fixed size which is determined at the time of *exec()* and cannot be increased or decreased dynamically as in virtual memory systems. This means that processes which require to run at the same address will always be of the same length and therefore, can be physically swapped with each other.

The algorithm of the *swapper()* is detailed below.

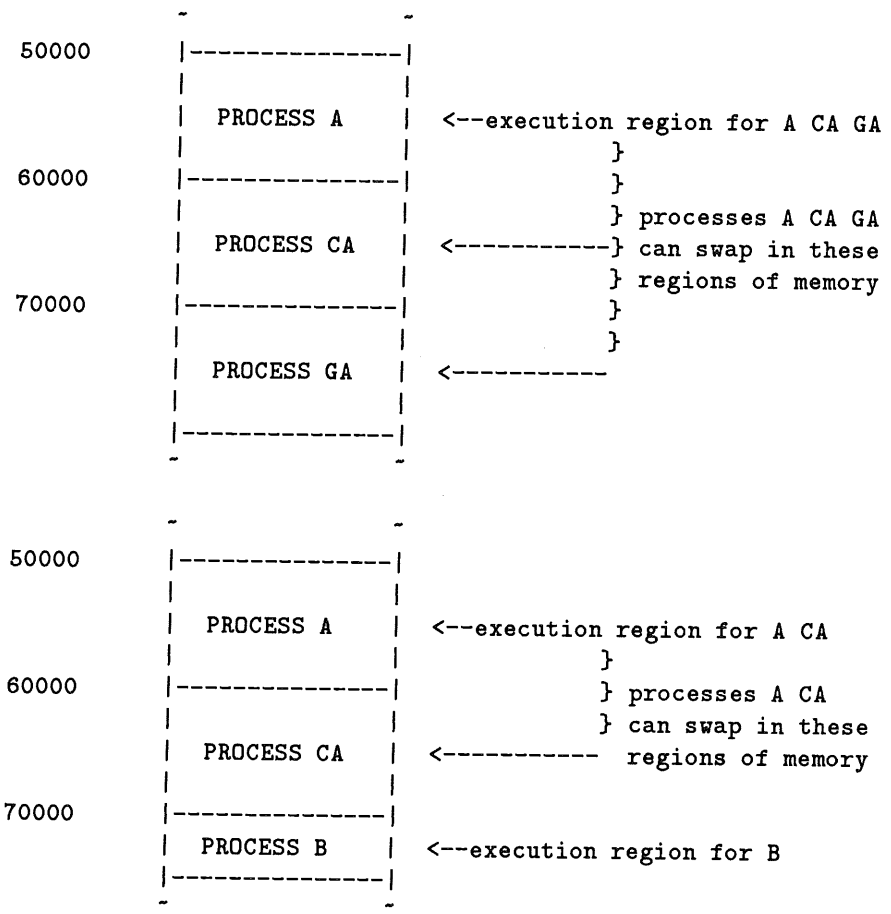
```
algorithm swapper
input:  proc table pointer
output: none
{
    if (!currently panicing && process not in execution region) {
        if (KERNEL process)
            panic(attempt to swap in kernel);

        for (all user processes)
            if (another process is in execution region) {
                found = true;
                break;
            }

        if (found == true) {
            if (processes are of different lengths)
                panic(cannot swap different length processes);
            swap process with that in execution region;
        } else
            copy process into its execution region;

        update the current address fields of the processes;
    }
}
```

Processes are always multiples of 4 bytes in length and start on even byte boundaries. This allows the *swapper()* to copy processes 4 bytes at a time which greatly increases the speed of moving processes around in memory.

Although swapping processes around in memory is slow, the actual cost in system performance is small; most processes will *fork()* and immediately follow with an *exec()*.

The *exec()* system call will relocate the new process so that it no longer needs to swap in order to execute. The child process is scheduled to run first and therefore the system often only needs to swap twice; the child process is swapped in prior to its *exec()* and the parent process is swapped in after the child has execed.

Consider the the following example: process A forks a child process CA, which in turn forks a grandchild process GA. This creates three processes whose execution addresses are equal (0x50000 say) and therefore, they can be swapped around in memory as each is given a time slice. Process GA now execs process B which gains a new execution address (0x70000 say). This has the effect of reducing the overhead of swapping processes, thus increasing the speed of the system. This scenario is displayed below.



Some care must be taken when copying data to and from processes. This will occur as part of message passing or due to kernel system call activities. Processes pass system call addresses which are valid when the process is swapped into its execution region. Such addresses are invalid when the process has been swapped out. Several routines have been modified in order that this is taken into account.

Swapping does have an unfortunate side effect however: interrupts are disabled during swapping thus preventing thrashing when large processes are involved. This means that clock interrupts can be lost when the processor is performing large amounts of swapping. Hence, the time-of-day clock tends slowly to lose time. Moreover, there is a significant flaw in the way processes are implemented on the Atari; the hardware provides no mechanism for protecting processes from each other. There is no way of stopping a malicious program which reads or writes outside its allotted memory. This is attributable to the hardware and not to the implementation of the STIX kernel.

4 System Calls

To make the task of invoking system calls easier, a C library has been written, which provides the programmer with convenient functions which interface with the KERNEL.

The STIX system call interface is superficially identical to that of MINIX; all of the MINIX calls have been provided and have identical functionality. Internally however, STIX system calls have been changed extensively. The modifications fall into two categories: those changes which were required for the software to run on the Atari and those which increase the power and efficiency of the KERNEL.

The more interesting modifications and design philosophies of the STIX KERNEL will now be described in detail.

4.1 Fork()

In order to retain the *fork()* system call, this implementation allows STIX to support process swapping. The concept is theoretically simple: STIX processes are of fixed sizes, therefore it is possible to move the process with the time-slice into the memory it requires, copying the process which resides there into the memory vacated by the first. If a process forks a child, these processes continue to execute by swapping in and out of the memory originally occupied by the parent.

The *fork()* system call itself knows little about the swapping process itself: this is handled by the context switching code. *Fork()* simply creates another entry in the process table, which contains new *execution* and *current* addresses.

This procedure actually reduces the complexity of the *fork()* system call; a process is treated as a contiguous region of memory rather then separate text, data and stack segments. Swapping, however, does make process termination and subsequent memory reallocation considerably more complex and is accomplished by the *exit()* system call.

4.2 Exit()

The *exit()* system call has the task of clearing up after processes which have terminated. This is on the whole a trivial task. However, there are three points of interest.

First, the memory manager has no understanding of swapping; it is only the kernel which knows the current address of a process. The memory manager knows the execution and creation address of a process, not the current address. This makes releasing the memory of a terminated process difficult: the memory manager does not know where a process is located.

The solution to this problem in STIX is achieved by adding the internal system call *sys_swap()*. This call causes the kernel to swap a named process

with anything currently residing in a given address. By calling *sys_swap()* using the terminating process number and creation address as parameters, the kernel can determine where any individual process is in memory.

Second, a parent process (typically it has an execution and creation region in common) whose creation region is being shared by its children (it is the child's execution region) must not release that memory when it terminates. The memory must be retained until the last child exits. The memory of a process with no children may be released when the process exits.

Assume process A forks a child process CA. Process A now terminates. Its memory cannot be released because it is being used by process CA as an execution region. This memory must be retained until process CA exits. If process CA had terminated first A's memory would have been released as soon as it had terminated.

Third, MINIX does not release memory associated with a process until the parent process has executed a *wait()* system call. This could result in a *fork()* or *exec()* system call erroneously failing due to lack of free memory.

STIX releases the memory of processes when they *exit()*. This leaves entries in the process table which have no memory associated with them. This is similar to the zombie process state of UNIX. The algorithm of *exit()* is illustrated below.

```
algorithm system_call(exit)
input:  return code for parent process
output: none
{
    store exit status for next wait() system call;

    if (parent is waiting)
        release parent and tell everybody;
    else
        suspend process;

    kill pending timers;
    tell_kernel(to swap process into creation region);
    tell_kernel(to stop scheduling process);

    if (no other process runs in execution region)
        free(execution region);

    if (execution region != creation region)
        if (no other process runs in creation region)
            free(creation region);

    /* Process is now in zombie state */
}
```

4.3 Exec()

The *exec()* system call invokes another program, "overlaying" the memory space of the current process with a copy of an executable file.

The *exec()* system call has been made considerably more complex by the port to the Atari, the added complexity being due to the lack of hardware memory management. The main problems which need to be overcome are program relocation and memory reallocation.

STIX can only support the small memory model of a process; the split instruction and data model implemented on a machine without hardware memory management. The maximum segment size of 64K has been removed from the KERNEL, but a practical limit of 128K is imposed by the superstructure.

A STIX executable file consists of several parts:
(i) A set of headers that describe the attributes of the file;
(ii) The program text and data; and
(iii) Relocation information.

The header describes the sizes of the text, data, bss and relocation sections of the executable, the total memory the executable requires and the magic number, which gives the type of the executable file.

The program text and data form the image which is initially loaded in the process address space.

The relocation information is a list of offsets within the text and data. Each element of the list represents a program byte, word or long which is an address to be relocated. Text and data relocation allows a process to be loaded and executed in any area of free memory.

A process invokes the *exec()* system call by calling one of its front-end functions. The *exec()* front-end builds an image of the new process stack from the supplied arguments and environment. The stack image is passed as an argument to the *exec()* system call proper.

The algorithm of *exec()* is described below:

```
algorithm user_exec
input:  pathname and arguments
output: -1 returned and errno set if exec failed
{
    Build a relocatable stack image with parameters;
    system_call(exec);

    /* Exec failed */
    set errno and return(-1);
}
```

The KERNEL validates the arguments passed by the *exec()* front-end. If they are invalid the kernel returns an error number to the user which relates to the reason for failure.

If the supplied parameters are valid, the KERNEL copies the stack image from the user space into a safe region of memory; when the *exec()* takes place the process image will be destroyed.

The kernel then opens the file which holds the executable image. The header information is examined; if there is not enough free memory to hold the new image the *exec()* system call fails and returns an error number to the user. This is the last point at which the kernel can recover from an error and return control to the calling process.

The kernel then attempts to free the memory occupied by the current process image. The creation region of the process is released and the execution region is also released if it is not required by any other executing process.

The image of the old process is now lost and free memory is allocated for the new process. This is filled by the image of the new process which is read in from the executable. The KERNEL uses the relocation information to modify the process image to run at its new address.

The KERNEL makes extensive checks in order to detect files which have been accidentally (or deliberately) corrupted. Corruption could take the form of invalid headers or modified/omitted relocation information. The kernel validates that the sizes of the text, data and relocation information of executable files matches those specified in their headers. The kernel also validates that the addresses to be relocated are within the text or data space of the process in question.

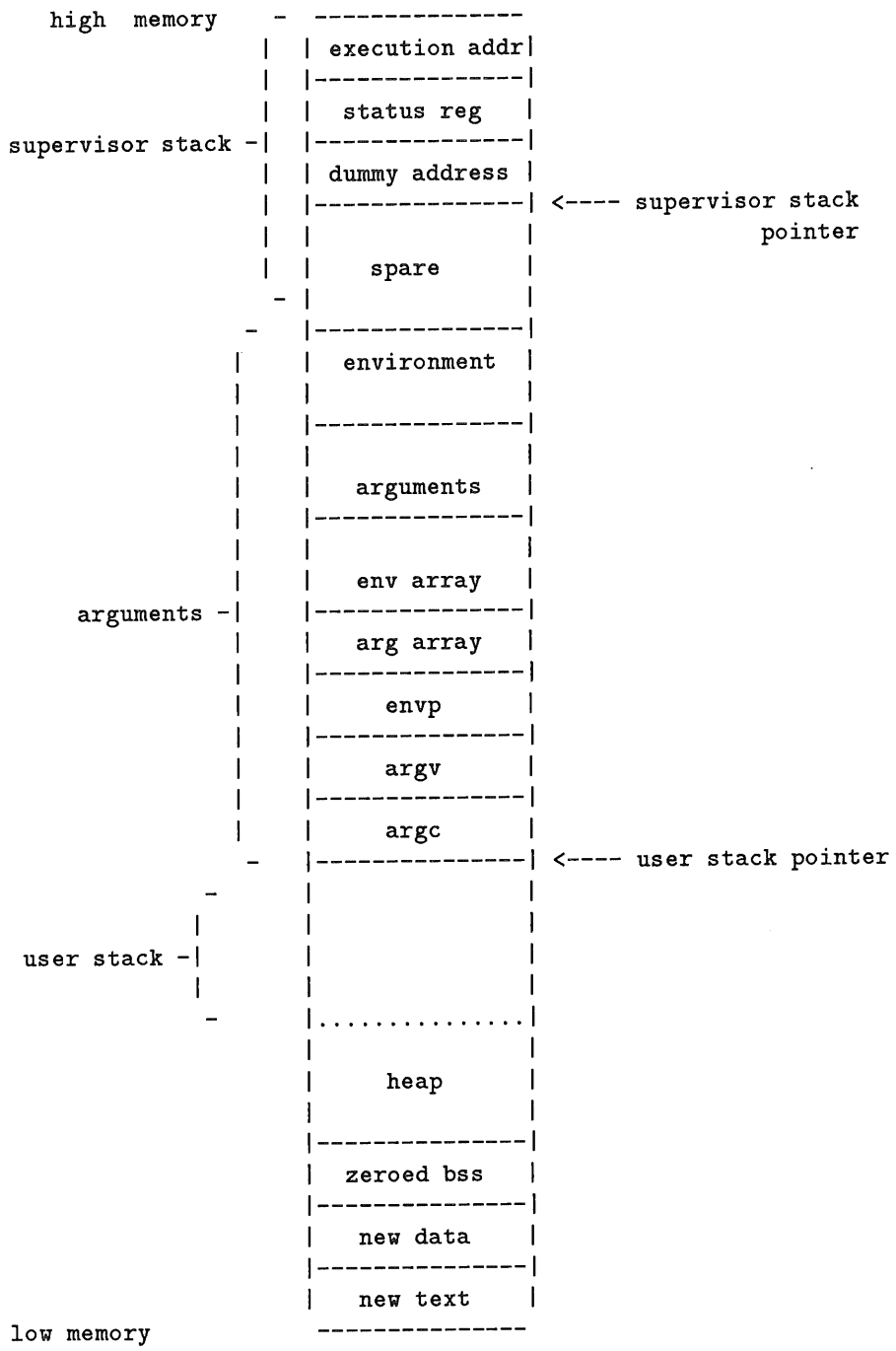
If either of these checks fails the process exits without returning to the user; continuing execution after such errors could result in a system crash.

The *exec()* call will now succeed. The KERNEL performs several tasks to initialise the new process. These include filling the process stack and bss with zeroes, copying and relocating the saved stack image at the top of the user stack, setting the permissions and scheduling the process to be restarted.

The algorithm described above is detailed below.

```
algorithm system_call(exec)
input:  pathname and stack image
output: only is exec fails
{
    if (invalid stack image || filename is !executable)
        return(error);
    Fetch the new stack from the user;

    Read the file header and extract the header sizes;
    if (!enough free memory for new image)
        return(error);

    /* Last point a failed exec() will pass back to user */

    tell_kernel(to swap process into creation region);

    if (no other process runs in execution region)
        free(execution region);

    if (execution region != creation region)
        if (no other process runs in creation region)
            free(creation region);

    Reclaim memory to hold new image;

    Read in the new image from the file;
    if (the header information is invalid)
        exit(process);

    Relocate the image to its new address;
    if (the relocation information is invalid)
        exit(process);

    zero the stacks and bss of the process;
    Copy the stack image into the stack;
    Relocate the stack image;
    Take care of setuid and setgid bits;
    Reset all caught signals;

    tell_kernel(tidy the stack and reschedule the process);
}
```

After a process has executed a successful *exec()* system call and prior to being restarted its memory map will resemble that in the diagram below:


high memory
execution addr
status reg
supervisor stack
dummy address
<---- supervisor stack pointer
spare
environment
arguments
env array
arguments
arg array
envp
argv
argc
<---- user stack pointer
user stack
heap
zeroed bss
new data
new text
low memory

The most important features of the diagram are:
(i) The new text and data have been loaded and relocated;
(ii) The bss, heap and stacks have been filled with zeroes;
(iii) The arguments and environment have been copied onto the top of the user stack. Arrays of pointers to each entry have been set up. Argc, argv and envp have been pushed onto the user stack; and
(iv) The supervisor stack has been constructed so that the process will restart execution in the C startup routine.

The C startup routine *crtso* is an assembly language routine which simply calls *main()* as a subroutine. No arguments are passed by *crtso*; these have already have been placed on the stack by the kernel. *Crtso* simply builds the standard parameter passing stack frame around the arguments. The routine main can then be treated as a standard C function.

5 Results and Conclusions

In material terms the result of the project is 3 disks: the STIX boot disk, a 400K root disk and a 680K /usr disk. These are all that are required to use the STIX operating system.

The KERNEL, comprising kernel, mm, fs, and init boots correctly. The KERNEL passes the entire Tanenbaum validation suite, which is a set of extensive tests.

The system retains version 7 compatibility. The authors wrote several programs which were developed under UNIX, using version 7 compatible function calls. These programs ran as expected when they were down-line loaded and run under STIX.

The authors opinion is that the STIX KERNEL has all the required functionality. Many MINIX bugs and "features" have been corrected. A large number of utilities have been provided to enable users to find their way around the system and so actually do serious work. The system has proved remarkably stable; users can make full use of all the available utilities without the system crashing. During extensive testing sessions, no system crash has ever occurred.

As a side effect of the project, an ST development system has been created on the Gould super mini-computer. This system could be used by anyone who wishes a more hospitable Atari ST development environment than that provided by the target machine itself.

The original aim of this project was to port the MINIX operating system to the Atari 520/1024 ST range of computers. The goals of the project state that the port is complete when it is possible to run the Tanenbaum validation suite successfully. The final product fulfills this criterion; it is functionally correct and highly stable.

6 References

[Croft; 1984] Croft W; *SUMacC: Stanford UNIX Mac C Development Kit*; Stanford University Medical Centre; Nov 1984.

[Gerits; 1988] Gerits K, Englisch L & Bruckman R; *Atari ST Internals - The authoritative insiders guide*; Abacus Software; 1988.

[Gull; 1988] Gull A; *STIX: A Port of the MINIX Operating System to the Atari ST*; BSc Final Year Project Report - City University London Computer Science Department; May 1988.

[Tanenbaum; 1983] Tanenbaum A, van Staveren H, Keizer E & Stevenson, J; *A Practical Toolkit for Making Portable Compilers*; Communications of the ACM; Vol 26, pp 654-660; September 1983.

[Tanenbaum; 1987a] Tanenbaum, A; *MINIX: A UNIX Clone with Source Code*; EUUG Newsletter; Vol 7, No 3, pp 3-11; 1987.

[Tanenbaum; 1987b] Tanenbaum, A; *Operating Systems: Design and Implementation*; Englewood Cliffs, NJ; Prentice-Hall; 1987.

AUUG 1988 Election Results

President:
Greg Rose

Secretary:
Tim Roper

Treasurer:
Michael Tuke

Committee
Rich Burridge
Frank Crawford
Chris Maltby
Tim Segall

Returning Officer
John O'Brien

Assistant Returning Officer:
- no candidates remained.

THIS PAGE INTENTIONALLY LEFT BLANK

AUUG

Membership Categories

Once again a reminder for all "members" of AUUG to check that you are, in fact, a member, and that you still will be for the next two months.

There are 4 membership types, plus a newsletter subscription, any of which might be just right for you.

The membership categories are:

Institutional Member
Ordinary Member
Student Member
Honorary Life Member

Institutional memberships are primarily intended for university departments, companies, etc. This is a voting membership (one vote), which receives two copies of the newsletter. Institutional members can also delegate 2 representatives to attend AUUG meetings at members rates. AUUG is also keeping track of the licence status of institutional members. If, at some future date, we are able to offer a software tape distribution service, this would be available only to institutional members, whose relevant licences can be verified.

If your institution is not an institutional member, isn't it about time it became one?

Ordinary memberships are for individuals. This is also a voting membership (one vote), which receives a single copy of the newsletter. A primary difference from Institutional Membership is that the benefits of Ordinary Membership apply to the named member only. That is, only the member can obtain discounts on attendance at AUUG meetings, etc, sending a representative isn't permitted.

Are you an AUUG member?

Student Memberships are for full time students at recognised academic institutions. This is a non voting membership which receives a single copy of the newsletter. Otherwise the benefits are as for Ordinary Members.

Honorary Life Memberships are a category that isn't relevant yet. This membership you can't apply for, you must be elected to it. What's more, you must have been a member for at least 5 years before being elected. Since AUUG is only just approaching 3 years old, there is no-one eligible for this membership category yet.

Its also possible to subscribe to the newsletter without being an AUUG member. This saves you nothing financially, that is, the subscription price is the same as the membership dues. However, it might be appropriate for libraries, etc, which simply want copies of AUUGN to help fill their shelves, and have no actual interest in the

contents, or the association.

Subscriptions are also available to members who have a need for more copies of AUUGN than their membership provides.

To find out if you are currently really an AUUG member, examine the mailing label of this AUUGN. In the lower right corner you will find information about your current membership status. The first letter is your membership type code, N for regular members, S for students, and I for institutions. Then follows your membership expiration date, in the format exp=MM/YY. The remaining information is for internal use.

Check that your membership isn't about to expire (or worse, hasn't expired already). Ask your colleagues if they received this issue of AUUGN, tell them that if not, it probably means that their membership has lapsed, or perhaps, they were never a member at all! Feel free to copy the membership forms, give one to everyone that you know.

If you want to join AUUG, or renew your membership, you will find forms in this issue of AUUGN. Send the appropriate form (with remittance) to the address indicated on it, and your membership will (re-)commence.

As a service to members, AUUG has arranged to accept payments via credit card. You can use your Bankcard (within Australia only), or your Mastercard by simply completing the authorisation on the application form.

AUUG

Application for Ordinary, or Student, Membership Australian UNIX* systems Users' Group.

*UNIX is a registered trademark of AT&T in the USA and other countries

To apply for membership of the AUUG, complete this form, and return it with payment in Australian Dollars, or credit card authorisation, to:

AUUG Membership Secretary
P O Box 366
Kensington NSW 2033
Australia

- Please don't send purchase orders — perhaps your purchasing department will consider this form to be an invoice.
- Foreign applicants please send a bank draft drawn on an Australian bank, or credit card authorisation, and remember to select either surface or air mail.

I, ... do hereby apply for

☐ Renewal/New* Membership of the AUUG	$65.00	
☐ Renewal/New* Student Membership	$40.00	(note certification on other side)
☐ International Surface Mail	$10.00	
☐ International Air Mail	$50.00	

Total remitted AUD$_____
(cheque, money order, credit card)

* Delete one.

I agree that this membership will be subject to the rules and by-laws of the AUUG as in force from time to time, and that this membership will run for 12 consecutive months commencing on the first day of the month following that during which this application is processed.

Date: __/__/__ Signed: _____

☐ Tick this box if you wish your name & address withheld from mailing lists made available to vendors.

For our mailing database - please type or print clearly:

Name: ..... Phone: ..... (bh)

Address: .... (ah)

..... Net Address: .....

.....

.....

.....

Write "Unchanged" if details have not altered and this is a renewal.

Please charge $_____ to my ☐ Bankcard ☐ Visa ☐ Mastercard.

Account number: _____ _____ _____ _____. Expiry date: __/__.

Name on card: _____ Signed: _____

Office use only:

Chq: bank _____ *bsb* ____-____ *a/c* _____ *#* _____

Date: __/__/__ *$* *CC type* ___ *V#* _____

Who: _____ *Member#* _____

Student Member Certification *(to be completed by a member of the academic staff)*

I, ... certify that

..... *(name)*

is a full time student at ..... *(institution)*

and is expected to graduate approximately ___/___/___.

Title: _____ Signature: _____

AUUG

Application for Institutional Membership Australian UNIX* systems Users' Group.

*UNIX is a registered trademark of AT&T in the USA and other countries.

To apply for institutional membership of the AUUG, complete this form, and return it with payment in Australian Dollars, or credit card authorisation, to:

AUUG Membership Secretary
P O Box 366
Kensington NSW 2033
Australia

• Foreign applicants please send a bank draft drawn on an Australian bank, or credit card authorisation, and remember to select either surface or air mail.

..... does hereby apply for

☐ New/Renewal* Institutional Membership of AUUG $300.00

☐ International Surface Mail $ 20.00

☐ International Air Mail $100.00

Total remitted AUD$_____

(cheque, money order, credit card)

* Delete one.

I/We agree that this membership will be subject to the rules and by-laws of the AUUG as in force from time to time, and that this membership will run for 12 consecutive months commencing on the first day of the month following that during which this application is processed.

I/We understand that I/we will receive two copies of the AUUG newsletter, and may send two representatives to AUUG sponsored events at member rates, though I/we will have only one vote in AUUG elections, and other ballots as required.

Date: __/__/__ Signed: _____

Title: _____

☐ Tick this box if you wish your name & address withheld from mailing lists made available to vendors.

For our mailing database - please type or print clearly:

Administrative contact, and formal representative:

Name: ..... Phone: ..... (bh)

Address: .... (ah)

..... Net Address: .....

.....

.....

.....

Write "Unchanged" if details have not altered and this is a renewal.

Please charge $_____ to my/our ☐ Bankcard ☐ Visa ☐ Mastercard.

Account number: _____ _____ _____ _____. Expiry date: __/__.

Name on card: _____ Signed: _____

Office use only: **Please complete the other side.**

Chq: bank _____ *bsb* ____-____ *a/c* _____ *#* _____

Date: __/__/__ *$* *CC type* ___ *V#* _____

Who: _____ *Member#* _____

Please send newsletters to the following addresses:

Name: ..... Phone: ..... (bh)
Address: .... (ah)
..... Net Address: .....
.....
.....
.....

Name: ..... Phone: ..... (bh)
Address: .... (ah)
..... Net Address: .....
.....
.....
.....

Write "unchanged" if this is a renewal, and details are not to be altered.

Please indicate which Unix licences you hold, and include copies of the title and signature pages of each, if these have not been sent previously.

Note: Recent licences usally revoke earlier ones, please indicate only licences which are current, and indicate any which have been revoked since your last membership form was submitted.

Note: Most binary licensees will have a System III or System V (of one variant or another) binary licence, even if the system supplied by your vendor is based upon V7 or 4BSD. There is no such thing as a BSD binary licence, and V7 binary licences were very rare, and expensive.

☐ System V.3 source
☐ System V.2 source
☐ System V source
☐ System III source
☐ 4.2 or 4.3 BSD source
☐ 4.1 BSD source
☐ V7 source
☐ Other *(Indicate which)* .....

☐ System V.3 binary
☐ System V.2 binary
☐ System V binary
☐ System III binary

AUUG

Application for Newsletter Subscription
Australian UNIX* systems Users' Group.

*UNIX is a registered trademark of AT&T in the USA and other countries

Non members who wish to apply for a subscription to the Australian UNIX systems User Group Newsletter, or members who desire additional subscriptions, should complete this form and return it to:

AUUG Membership Secretary
P O Box 366
Kensington NSW 2033
Australia

- Please don't send purchase orders — perhaps your purchasing department will consider this form to be an invoice.
- Foreign applicants please send a bank draft drawn on an Australian bank, or credit card authorisation, and remember to select either surface or air mail.
- Use multiple copies of this form if copies of AUUGN are to be dispatched to differing addresses.

Please *enter / renew* my subscription for the Australian UNIX systems User Group Newsletter, as follows:

Name: ..... Phone: ..... (bh)

Address: .... (ah)

..... Net Address: .....

.....

..... *Write "Unchanged" if address has*

..... *not altered and this is a renewal.*

For each copy requested, I enclose:

☐ Subscription to AUUGN	$ 65.00
☐ International Surface Mail	$ 10.00
☐ International Air Mail	$ 50.00

Copies requested (to above address) _____

Total remitted AUD$_____

(cheque, money order, credit card)

☐ Tick this box if you wish your name & address withheld from mailing lists made available to vendors.

Please charge $_____ to my ☐ Bankcard ☐ Visa ☐ Mastercard.

Account number: _____ _____ _____ _____. Expiry date: __/__.

Name on card: _____ Signed: _____

Office use only:

Chq: bank _____ *bsb* _____ - _____ *a/c* _____ *#* _____

Date: __/__/__ *$* *CC type* ___ *V#* _____

Who: _____ *Subscr#* _____

AUUG

Notification of Change of Address
Australian UNIX* systems Users' Group.

*UNIX is a registered trademark of AT&T in the USA and other countries.

If you have changed your mailing address, please complete this form, and return it to:

AUUG Membership Secretary
P O Box 366
Kensington NSW 2033
Australia

Please allow at least 4 weeks for the change of address to take effect.

Old address (or attach a mailing label)

Name: ..... Phone: ..... (bh)

Address: .... (ah)

..... Net Address: .....

.....

.....

.....

New address (leave unaltered details blank)

Name: ..... Phone: ..... (bh)

Address: .... (ah)

..... Net Address: .....

.....

.....

.....

Office use only:

Date: ___/___/___

Who: _____ *Memb#* _____