AUUGN

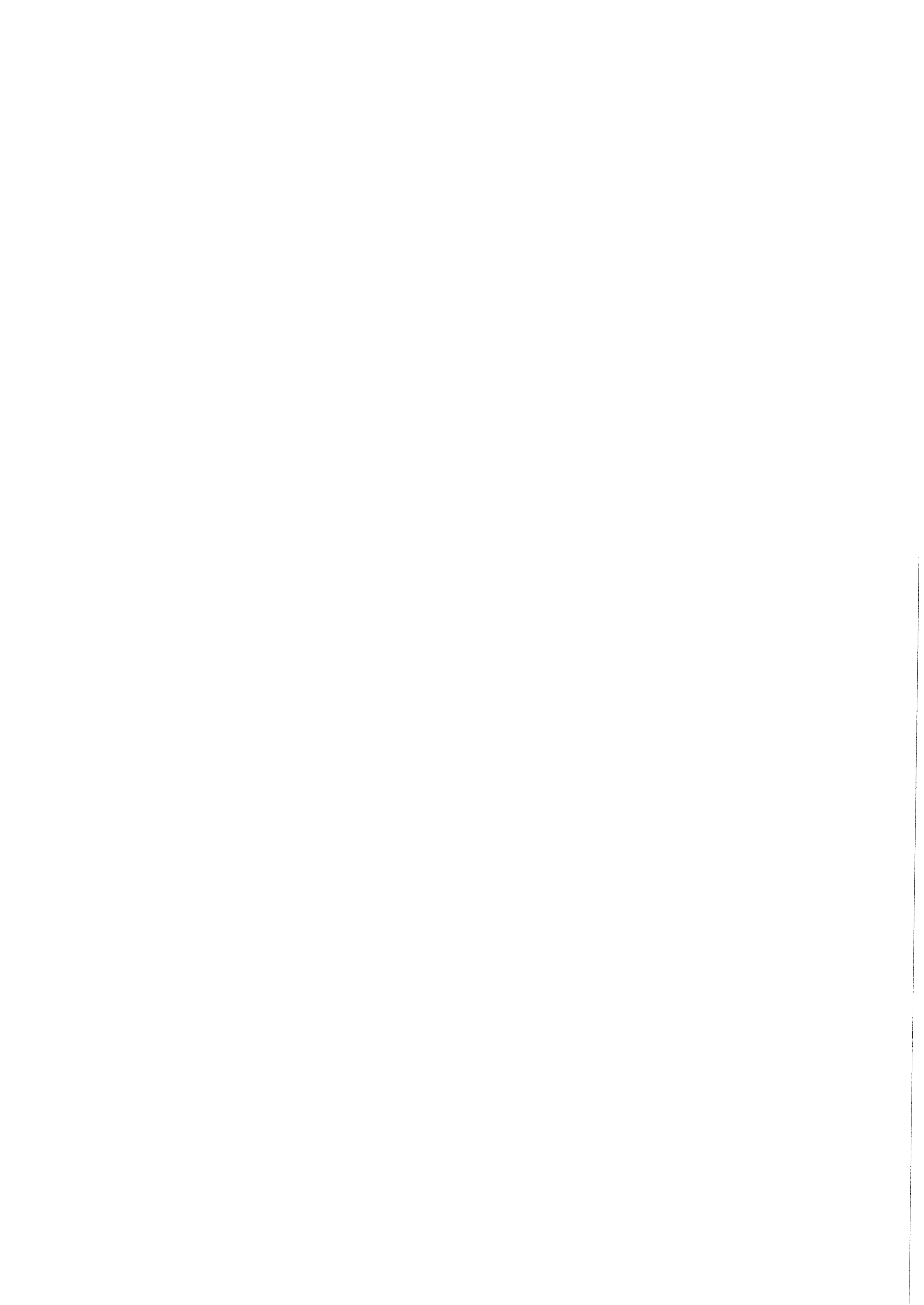
Australian Unix systems User Group Newsletter

Volume 9 - Number 3

June 1988

20 JUL 1988

Registered by Australia Post Publication No. NBG6524



The Australian UNIX* systems User Group Newsletter

Volume 9 Number 3

June 1988

CONTENTS

AUUG General Information 3

Editorial 4

Ross Nealon, 1958-1988 6

A Tribute to Ross Nealon 7

AUUG 88 Winter Conference And Exhibition 9

AUUG 88 Key Speaker Biographies 10

AUUG 88 Programme 12

AUUG 88 Registration 13

AUUG 88 Contact Details 14

Adelaide UNIX Users Group Information 15

Western Australian UNIX systems Group Information 16

Book Review - The AWK Programming Language 17

Book Review - Software Configuration Management 18

Report from the Fifth Workshop on Real-Time Software and Operating Systems 19

PostScript On Screen: Here is the NeWS 24

Evolution of the SunOS Programming Environment 26

SunOS Virtual Memory Implementation 42

From the *;login:* Newsletter - Volume 13 Number 3 59

- Know Your Board and Staff 60
- Call for Papers - UNIX Security Workshop 61
- Call for Papers - Workshop on UNIX and Supercomputers 62
- Call for Papers - C++ Conference 63
- Call for Papers - Workshop on Large Installation Systems Administration 64
- Call for Papers - EUUG Autumn Conference 65
- LOCK/ix: An Implementation of UNIX for the LOCK TCB 66
- An Update on UNIX Standards Activities 80
- Local User Groups 85

From the *;login:* Newsletter - Volume 13 Number 3 *continued* 87

Future Events 87

Publications Available 87

Management Committee Meeting Minutes - February 1988 88

AUUG Membership Catorgories 95

AUUG Forms 97

Copyright © 1988. AUUGN is the journal of the Australian UNIX* systems User Group. Copying without fee is permitted provided that copies are not made or distributed for commercial advantage and credit to the source must be given. Abstracting with credit is permitted. No other reproduction is permitted without prior permission of the Australian UNIX systems User Group.

* 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 **John Lions**

johnl@cheops.eecs.unsw.oz
School of Electrical Engineering
and Computer Science,
University of New South Wales,
New South Wales

Secretary **Robert Elz**

kre@munnari.oz
Department of Computer Science,
University of Melbourne,
Victoria

Treasurer **Chris Maltby**

chris@softway.sw.oz
Softway Pty. Ltd.,
New South Wales

Committee Members

Chris Campbell

chris@comperex.oz
Comperex Pty. Limited,
New South Wales

Piers Lauder

piers@basser.cs.su.oz
Basser Department of Computer Science,
Sydney University,
New South Wales

Tim Roper

timr@labtam.oz
Labtam Limited,
Victoria

Peter Wischart

pjw@anucsd.oz
NEC Information Systems,
Canberra

Next AUUG Meeting

The next meeting will be held in Melbourne at the Southern Cross Hotel from the 13th to the 15th of September 1988. Futher details are provided in this issue.

AUUG Newsletter

Editorial

Welcome to another issue of the AUUG Newsletter.

As reported to you in the last issue, I have moved to Webster Computers, and have put a lot of effort into making the transision as smooth as possible as far as Newsletter production is concerned. This has not been done alone, and I have had help from many people including the staff at Websters, Monash Computer Centre, and the AUUG Committee. I wish to thank them very much.

There are still minor production, distribution, renewal problems occuring with AUUGN and the Committee and myself are working to rectify them.

The AUUG is hosting its major conference at the Southern Cross Hotel in Melbourne during September. Details appear in this issue and you will be receiving forms in the mail in the near future.

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 12th of August 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 Chris Maltby.

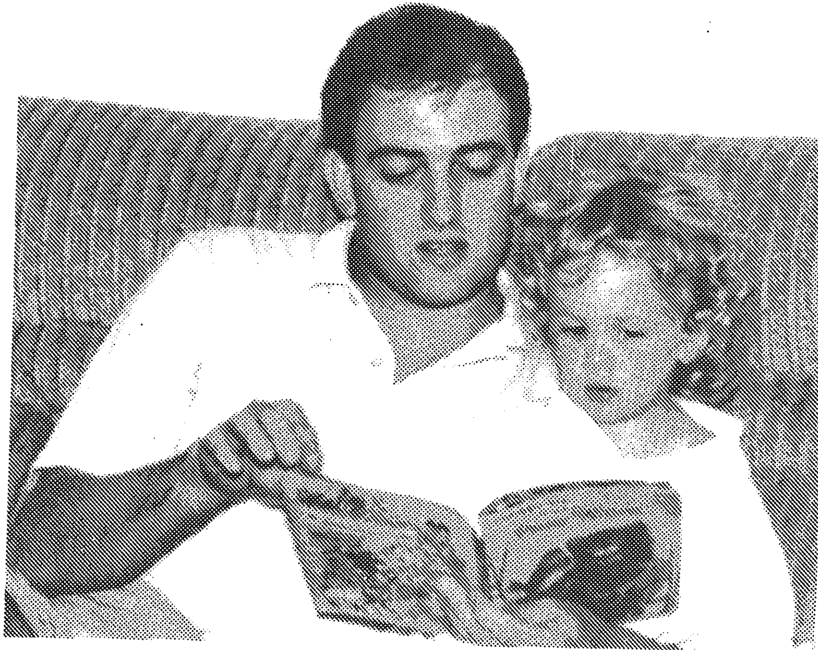
Acknowledgement

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.

Ross Nealon, 1958-1988



AUUG recently lost one of its founding members, and one of its staunchest supporters in Ross Nealon, who passed away in the early hours of Sunday, March 27, 1988, after a fifteen-month struggle with Hodgkinson's disease. Ross had accomplished much in thirty years, but there was still much that he wanted to do.

As the University of Wollongong's first Honours Graduate in Computing Science, through his participation in the original port of UNIX from the PDP/11 to the Interdata 7/32 computer, and through all his subsequent activities, he had a legitimate claim to be one of Australia's pioneers and foremost experts on the UNIX system. He attended all the early meetings of AUUG and most of the recent ones. He was always cheerful, always willing to help out and always able to provide sound opinions and good advice.

A tribute to Ross provided by his guide and mentor over many years, Professor Juris Reinfelds of the University of Wollongong, appears on the next pages.

Ross was a 'good bloke' in the most genuine sense of the term. He will be missed by his family, colleagues and many friends.

John Lions
University of New South Wales

A Tribute to Ross Nealon

Ross joined the University of Wollongong as a first year student in 1976, the year that the University first offered a course sequence for Computing Science majors. As a student with an avid interest in Computing Science and a naturally gifted programmer, Ross participated in the course development more than he realized at the time. His actions and responses to each new subject were of great value in the development of the subject and in the integration of the subjects into a coherent course with a unique flavour that was entirely our own.

Ross completed our BSc. programme in 1978 and took out First Class Honours in 1979. Every Computing Science major at Wollongong University in his or her final year has to participate in a major programming project that draws on, and ties together, the knowledge acquired in the previous two and a half years. For his third year project Ross designed and implemented a universal cross-assembler for all then available microcomputers. The concept and code were very good, so we placed the the cross assembler on the Usenix distribution tape. It was so good in fact that a couple of years later, GenRad Inc., a well-known maker of circuit board testing devices, called me up from the USA and volunteered a payment for the use of Ross's software in their products. This generous action by a far away company gave Ross enough funds for his first microcomputer and it restored my faith in industry-university relations.

An old proverb says: 'Tell me what your friends are, and I will tell you what you are'. This can be paraphrased for system programmers: 'Tell me what code you read, and I will tell you what sort of systems expert you are'. Ross enjoyed reading others' programs as much as he enjoyed writing his own. He was only a first year student when, in December, 1976, Richard Miller and I decided to port UNIX to the Interdata 7/32. For this port — which became the first successful port of UNIX world-wide — Richard read all the available UNIX source code as well as the Interdata system code, and started writing. I did my best to sustain our enthusiasm and confidence against the collected wisdom and scepticism of all our colleagues. Ross, not having been taught what is impossible, offered to help and was made responsible for the UNIX editor, *ed*, which he successfully ported to the Interdata 7/32 computer by January 10, 1977, a most impressive feat for a first year student! Ross completed many other support tasks for the first UNIX port, and we were not at all surprised that after graduation Ross joined our department as a Professional Officer to continue with the development and evolution of the UNIX system and other projects.

A Tribute to Ross Nealon

In 1979, I initiated a series of yearly computing summer schools for the best year 11 students in NSW and interstate. In 1981, Ross took over the editing of the proceedings and, from assorted lecture notes every year for three years, produced a book of lasting value to the participants. As a perfectionist with an excellent sense of beauty and a splendid sense of humour, Ross put a lot work and energy into improvements of the *nroff* and *troff* suite of formatting tools of UNIX. In 1985, when the Australian Computer Journal was experiencing difficulty in obtaining suitable typesetting services, the editor, John Lions, converted a number of articles that were already available in machine readable form to *troff* format, and arranged with Ross to typeset them on the Wollongong University Computer Centre's phototypesetter. There were more than a few software hurdles to cross before all worked successfully because Ross was working with the early version of *troff* (i.e. pre-*ditroff*) which he had converted to work with the Compugraphic typesetter. Since November, 1985, many of the articles that have appeared in the ACJ were typeset by Ross. The high spot was the August, 1986 issue that was entirely set at the University of Wollongong with his assistance.

In 1980, the Department of Computing Science decided to develop a platform for studies in distributed systems. The basis for this platform was to be a commercially available Cambridge Ring with all the software and some interfaces developed by the department. Without earmarked resources for such developments, the project was the the mercy of graduate student interest and academic staff enthusiasm. These waxed and waned without much progress for a number years until one day at an especially low point of the project, Ross volunteered to take it on in addition to all his other duties. He finished it in a remarkably short period of time, and it turned out to be his last major completed project. Ross's Cambridge Ring software will connect Computing Science terminals to computers without extra wiring or expensive crossbar switches for several years to come.

Ross's devotion to duty and his professionalism were total; his honesty and integrity were impeccable. Whether he was making sure that all lecture materials, office tools and accessories were available at lectures or seminars in locations away from Wollongong or whether he was arranging the system or developing the software for a new course or project for our students or staff, one could always count on everything being in place and working when needed. Ross's UNIX expertise was in demand in Australia and overseas. He conducted several lecture courses and workshops in Sydney, Singapore and the Philippines.

Ross helped to shape Computing Science in Wollongong from its beginnings. His style, his dedication, his professionalism and his personality have all contributed to the development and future direction of our department, and through this he will always be with us and with future generations of Wollongong Computing Science graduates.

Juris Reinfelds
University of Wollongong

AUUG 88 Winter Conference and Exhibition

The **1988 Winter Conference and Exhibition** of the Australian UNIX† systems User Group will be held on **Tuesday 13th – Thursday 15th September 1988** at the Southern Cross Hotel in Melbourne, Australia. AUUG has concentrated its efforts in 1988 into this major event:

- the largest conference and exhibition ever held by AUUG
- the first to offer so many renowned overseas speakers
- the first to be held in a first class venue
- the first to offer a three day, wide ranging programme

We expect this to be the most significant public event featuring the UNIX system in Australia in 1988. It will be accompanied by the biggest exhibition of computers running the UNIX system ever seen at an AUUG conference.

The conference theme is

Networking – Linking the UNIX World.

We are delighted to have **Michael Lesk** delivering the keynote address. Whilst at Bell Laboratories, Michael invented *uucp*, the original and still the most widespread UNIX networking software. Michael will also give a presentation on human interface and natural language issues.

John Mashey started working with the UNIX system in 1973 at AT&T and has continued to do so with Convergent Technologies and MIPS Computer Systems. His expertise in software engineering and RISC architectures as well as his highly regarded talent for public speaking should make for an informative and entertaining time. John will speak on hardware/software tradeoffs with RISC architectures and on software engineering issues.

Major work has been done at the Computer Systems Research Group, University of California at Berkeley, on the UNIX system and particularly its networking features. **Mike Karels** is one of the leaders of that group and as a noted expert on networking will be a welcome speaker at the conference. He will be speaking on current and future developments of the BSD UNIX system and on its support of OSI/ISO networking.

AUUG is honoured that **Ken Thompson** will be attending and speaking at the conference. Ken was one of the principal designers of the UNIX system at Bell Laboratories.

In addition to talks by the the invited speakers, there will be presentations of refereed papers. These are on various aspects of the UNIX system, with some but not all relating to the *Networking* theme. A Conference Proceedings will be published as a special issue of the group's newsletter *AUUGN*. This will be available at the conference and is included in the registration fee.

A traditional highlight of AUUG conferences is the Conference Dinner. This year it will be held at the Southern Cross on the evening of the second day, Wednesday 14/9/88. A new feature this year will be a Cocktail Party on the evening of the first day, Tuesday 13/9/88.

---

† UNIX is a trademark of Bell Laboratories.

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 88 Programme

Day 1 - Tuesday 13/9/88

0900-1000	Registration
1000-1100	Keynote: Michael Lesk *Unix Networks: Why Bottom-Up Design Beats Top-Down*
1100-1130	Coffee & Exhibition Viewing
1130-1300	Windowing Systems
1300-1430	Lunch
1430-1500	Future Telecommunication Developments
1500-1530	Wide Area Networks 1
1530-1600	Coffee & Exhibition Viewing
1600-1700	ISO/OSI under UNIX
1730-1900	COCKTAIL RECEPTION

Day 2 - Wednesday 14/9/88

0830-0900	Registration
0900-1000	*Futures for UNIX Software*: Larry Crume, AT&T UNIX Pacific
1000-1100	The Open Software Debate
1100-1130	Coffee & Exhibition Viewing
1130-1230	Future Berkeley UNIX Developments: Mike Karels
1230-1400	Lunch
1400-1430	Distributed File Systems
1430-1500	Legal and Social Issues
1500-1530	Security Issues
1530-1600	Coffee & Exhibition Viewing
1600-1700	Human Interface Issues: Michael Lesk
1700-1730	AUUG General Meeting
1900-2300	CONFERENCE DINNER

Day 3 - Thursday 15/9/88

0830-0900	Registration
0900-0930	AUUG ACSnet SIG Meeting
0930-1030	Wide Area Networks 2
1030-1100	Coffee & Exhibition Viewing
1100-1200	*RISC and the Motion of Complexity*: John Mashey
1200-1230	Local Area Networks
1230-1400	Lunch
1400-1430	Programming Language Issues
1430-1500	*A New C Compiler*: Ken Thompson
1500-1600	Operating System Issues
1600	Close

AUUG 88 Registration

Registration forms will be mailed to all AUUG members or may be obtained by contacting the AUUG 88 Secretariat (see below). Accommodation at the Southern Cross or the Victoria Hotel and extra tickets for the Conference Dinner should also be booked on that form if required.

Registration Fees

3 Day Registration		
	members	$200
	non-members	250
	Includes coffees, lunches, Proceedings, Cocktails, Conference Dinner	
Full-time Student 3 Day Registration		80
	Includes coffees and Proceedings	
Single Day Rate		100
	Includes coffees and lunch on the day and Proceedings	
Late Fee		50
	Added to the above for payment after 26/8/88	

AUUG 88 Contact Details

Registration, Accommodation and Exhibition

AUUG88 Secretariat
c/o ACMS
26 Hopewell St
Paddington
NSW 2021
Australia

Phone:	International	+61 2 3324622
	National	02 3324622
Fax:	International	+61 2 3324066
	National	02 3324066
Telex:	AA176765	
Viatel Mailbox No:		233246220
Minverva/Dialcom:		07:WDF001

Programme Committee

Timothy Roper
AUUG 88
c/o Labtam Limited
43 Malcolm Road
Braeside
Victoria 3195
Australia

Phone:	International	+61 3 5871444
	National	03 5871444
Fax:	International	+61 3 5805581
	National	03 5805581
Telex:	LABTAM AA33550	
ACSnet:	timr@labtam.oz	
UUCP:	uunet!munnari!labtam.oz!timr	
ARPA:	timr%labtam.oz@uunet.uu.net	

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

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**

Book Review

Tim Roper

The AWK Programming Language

Alfred V. Aho, Brian W. Kernighan, Peter J. Weinberger
Addison-Wesley

Anyone maintaining a library of significant books relating to UNIX† operating systems will buy this one. The status of *awk* as part of the UNIX toolkit and the standing of the authors ensures that. Their technical authority can hardly be questioned and previous works of (at least two of) the authors are well known.

Otherwise, this book can be purchased confidently by

1. new awk users who want a gentle introduction
2. managers assessing the language for suitability to their cause
3. existing awk users who want a complete reference manual or a refresher course

This structure of this book as explained in the Preface (that should be read) accommodates each group. The first chapter is a tutorial introduction with many small examples that the new user will want to read alongside their terminal. The second chapter is a complete language description, without quite being a reference manual, as it uses examples and some repetition. The rest of the book consists of examples grouped into subject areas, one per chapter. Suitable approaches for the three readerships are

1. read the first chapter with terminal at the ready, read one or two example chapters to gain breadth, study the second chapter for depth, and complete the remaining example chapters in time.
2. read the first chapter and whichever example chapters are of interest
3. read the second chapter, explore the example chapters for new ideas, and then refer to the summary appendix and second chapter as required.

There is a problem when using the book as a reference manual. It incorporates "new awk" (as distributed in System V Release 3.1) and hence some features may be missing from other implementations. This fact is noted in the Preface but the likely differences are not noted in the text. Differences could have been noted in the summary appendix if not in the second chapter.

My copy cost $31-95 at McGill's Newsagency, Melbourne.

† UNIX is a trademark of Bell Laboratories.

Book Review

Tim Roper

Software Configuration Management

Wayne A. Babich
Addison-Wesley

I liked this book. It was immediately relevant to what I was doing at the time and the writer's style and the book's length of 162 pages makes it easy to read.

Babich builds on a theme of F. P. Brooks' *The Mythical Man-Month*, namely the worse-than-linear relationship between productivity and number of programmers which Brooks attributes to problems of communication between staff members. Babich defines his title:

> *The art of coordinating software development to minimize this particular type of confusion is called* **software configuration management**.

He develops his arguments through a series of presumably fictitious but realistic and often amusing anecdotes. Sketches of headless chooks loose in a computer room add to the atmosphere. Some of the problem scenarios and their multiple choice answers have a humour that you may find dry, sarcastic or trite. (Many will find them all too familiar.)

One aspect of the book that I found useful was that it offered an extensive set of terms for naming many elements and problems of software management. Unfortunately it does not include a glossary.

One chapter discusses configuration management under the UNIX† operating system and its tools such as *make*, SCCS and RCS. Although Babich compares SCCS with RCS he emphasises them as primitives providing a possible basis for implementing higher level procedures. He outlines but does not detail how this might be done. The next chapter discusses the Ada Language System of which Babich was an designer. He claims some bias in this respect I don't think that any is significant. I was interested to read about a system that is more ambitious and better integrated than *make* and SCCS or RCS.

The final chapter *Applying the Principles* Babich suggest applying his principles of configuration management throughout the software development process starting with the system design phase. This seems reasonable. However he appears to gives the configuration manager excessive influence over the detailed design phase, to the point where minimising configuration management problems may be stressed to the exclusion of others.

This book is relevant to project leaders managing the development of software products involving several programmers, several alternative versions of the product or where the software has or will have a long life cycle. Other programmers may also find it useful, interesting or entertaining. If you are a lone developer not using a source code control system and surviving or using one and finding it adequate, the book may just offer promises of things to come. If you don't use a source code control system and are in a mess, you should read it. It may explain some of the reasons for your trouble. If you are using one but find that it does not solve problems due, for example, to double maintenance, shared data or simultaneous update, you should read it. If you are about to embark on a significant project involving a team of programmers producing a product with many versions and a long life cycle and think that SCCS or RCS is all you need, you definitely should read it.

Price was about $A28-00.

---

† UNIX is a trademark of Bell Laboratories.

The Australian National University

G.P.O. Box 4, Canberra A.C.T. 2601, Australia

C. J. Fidge
Department of Computer Science
cjf@anucsd.oz
(062) 49–4725

10th June 1988

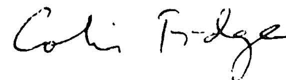
J. Carey,
AUUGN Editor,
Webster Computer Corporation,
1270 Ferntree Gully Road,
Scoresby, Victoria 3179.

Dear Dr. Carey,

Enclosed please find a contribution for the next issue of the Australian Unix systems User Group Newsletter.

As the only Aussie at the real-time workshop, and considering that your last issue included the workshop announcement, I felt that a brief description of the meeting would be timely.

Yours sincerely,



Colin Fidge

Report from the Fifth Workshop on Real-Time Software and Operating Systems

C. J. Fidge
Department of Computer Science
Australian National University
cjf@anucsd.oz

ABSTRACT

This article briefly summarises the issues discussed at the Fifth Workshop on Real-Time Software and Operating Systems, held in Washington D.C., 12–13 May 1988. The workshop was sponsored by the IEEE Computer Society and the USENIX Association.

INTRODUCTION

During a recent U.S. trip (to present a paper at the ACM Workshop on Parallel and Distributed Debugging) I took the opportunity to attend RTSOS'88 in Washington D.C. My interests are principally centred around parallel programming environments and the concept of time in distributed systems, rather than operating systems *per se*, and this report reflects that bias.

The workshop was held in the luxurious Omni-Shoreham Hotel and run in a very businesslike manner. Despite the conscious decision to include more panel sessions than in previous years, the meeting still felt more like a short conference than a "workshop", with an emphasis on formal presentations.

In all 20 extended abstracts were selected to appear in the proceedings from 76 submissions. The following is an attempt to briefly summarise the major issues discussed.

OPENING ADDRESS

John Stankovic (Univ. of Massachusetts) opened the workshop with a summary of the problem area. He characterised the field as one in which program corectness depends not only on the results of a computation, but also on the speed at which these results are produced. He identified the following application areas, in increasing order of difficulty:

- simple experiments,
- data acquisition,
- process control,
- avionics,
- expert systems in real-time,
- space station,
- mobile robots in hazardous environments, and
- SDI.

Most of these areas, including the last (!), received attention during the course of the workshop.

Session I: REAL-TIME SCHEDULING TECHNIQUES

The principal problem discussed during this session was how to schedule a number of tasks given that each has a priority and/or deadline. Deadlines may be either *hard*, i.e. specifying an absolute time limit, or *soft*, i.e. an average response time is given for tasks of this type. For example, assuming that all deadlines cannot be met, is it better to let one high-priority task miss a deadline, or several low-priority tasks? Another major issue was the question of *fairness* vs. *priority* scheduling. In the first case all tasks get roughly equal access to the CPU, making progress at their own pace. This is obviously a very simply scheduling policy. In the latter case low-priority tasks are starved, in favour of those with higher priorities.

Hideyuki Tokuda (Carnegie Mellon University) opened proceedings with an interactive tool for schedulability analysis called *Scheduler 1-2-3*. This includes a graphics interface, with histograms representing activity in each task, for checking timing constraints in the *design* phase of program development. Currently the tool only supports a single scheduling policy; future work will allow for greater flexibility.

Insup Lee (Univ. of Pennsylvania) took a more theoretical approach. He presented an extension to the 1985 version of Hoare's CSP language, with explicit features for specifying the real-time duration of events. Formal axioms and proof rules for this model have been defined.

Sara Biyabani (Univ. of Massachusetts) analysed the performance of two new scheduling algorithms which differed in the way low-priority tasks are "bumped" by higher priority tasks. In the first, the lowest is always selected, and in the second, any lower priority task is chosen. Both alternatives were shown to perform better than several popular algorithms, even under adverse conditions such as

"bursty" task requests. The point was made that in many applications the "value" of a task vs. elapsed time is a step-function; after the deadline has been missed the results are worthless and further computation is just a waste of CPU time.

Ragunathan Rajkumar (Carnegie Mellon Univ.) presented a solution to the "priority inversion problem" in which high-priority tasks are blocked by low-priority tasks for an indefinite period of time. This took the form of a variant of a priority *inheritance* protocol where each executing task inherits the highest priority of those tasks currently waiting. This prevents several low-priority tasks from continually pre-empting each other to the detriment of waiting high-priority tasks. Priority scheduling was felt to be better than fairness since, although low-priority tasks may be starved, *important* deadlines will be met. He also noted that "time" should be a consideration throughout the entire software development lifecycle.

Panel I: MYTHS IN REAL-TIME SYSTEMS - FAIRNESS CONSIDERED HARMFUL

The deliberatively provocative title of the first panel session triggered a lively discussion. Using fairness as the run-time scheduling policy was seen to lead to a system that maximised CPU usage, but made it difficult to predict beforehand which tasks would meet their deadlines. Prof. Stankovic advocated the reverse approach—sacrificing some speed for the sake of *predictability*. Program correctness was thus easier to guarantee.

Douglass Locke (IBM) emphasised the danger of *fragility* in real-time systems undergoing maintenance. Modifying one task may unexpectedly lead to missed deadlines in other, unchanged tasks, even when there is no logical connection between them, due to the effects of scheduling. An unanswered question raised during a later session related to this: should context switching be seen as a synchronisation event?

Lui Sha (Carnegie Mellon Univ.) advocated priority scheduling and asserted that it had proved its value in solving "real life" problems in the past, but admitted that more work was required.

Aloysius Mok (Univ. of Texas) pointed out that fairness is needed to guarantee progress when the program is nondeterministic, but conceded that it cannot be used in general when we must guarantee a bounded response time.

Finally, Pat Watson (IBM) noted that FIFO queueing, like fairness, will lead to unpredictable real-time deadline satisfaction behaviour (presumably since it *arbitrarily* interleaves events based on arrival time).

Session II: LANGUAGES AND O/S

Among the issues considered during this session was the effect of process "weights" at the applications program level, i.e. fine vs. large grained parallelism, on the desirable o/s primitives.

David Jameson (IBM) presented a lively talk on the ORE language and run-time environment, designed for programming high-speed, real-time systems. The current application area, "the juggling robot", caused as much comment as the language itself. It was pointed out that the robot is a good testbed for experimentation since it is difficult, real-time and, most importantly, *safe*. The language included lightweight parallelism, efficient synchronisation primitives, and a "restructurable" scheduler. The latter means that the programmer explicitly defines the scheduling policy. This was done since it was felt that scheduling is not understood well enough as yet to be "hard-wired" into the system. Non-preemptive scheduling is used so that memory requirements, stack offsets, etc. can be calculated at compile-time, thus maximising run-time speed. The language, and robot, are both still under construction.

Kwei-Jay Lin (Univ. of Illinois) treated time as an operating system resource that, like any other, may become "unavailable". Granted that deadlines may be variable, techniques were presented for generating *imprecise* results that made the best use of available time. Of primary importance was the concept of *monotonicity*—an algorithm must not oscillate around the desired result, but must continually approach it so that whenever the program runs out of time, the result produced is the best possible. Progress towards a goal was measured either as data-oriented (e.g. the number of input items used), or operation-oriented (e.g. the number of statements executed). Termination could be classified either as completion of the algorithm, reaching a fixed point, stopping on an error, out of time, etc. These two measures, progress and termination, define the precision of the result.

Ahmed Gheith (Ohio State Univ.) described an object oriented operating system for real-time applications, CHAOS. An industrial robot application was used to critically analyse CHAOS and suggest improvements.

Mok formally analysed the "Earliest Deadline" algorithm for enforcing hard deadline constraints, and concluded that a "heap of heaps" data struc-

ture for managing control blocks improved on the traditional single heap implementation.

Session III: REAL-TIME OPERATING SYSTEMS

This brief session discussed operating system support for real-time concepts.

John Barr (Motorola) described an operating system kernel that could support both UNIX* and a locally produced o/s. Features emphasised were support for scheduling, interprocess communication, interrupt handling and *fast* memory management.

Lou Salkind (New York Univ.) described the SAGE operating system kernel. The emphasis was on real-time supervisory control, with robotics applications, including the experimental "four finger manipulator".

Panel II: POSIX

Real-time extensions to the new IEEE UNIX Standard were also hotly debated. Barr emphasised the need for simpler kernels, with as few primitives as possible. Sol Kavy (Hewlett-Packard) similarly asserted that the *cost* of any proposed kernel features must be among the primary considerations.

However Marc Donner (IBM) attracted the most attention by daring to ask "why?". Acting as devil's advocate he suggested that UNIX is intended as an environment for development, not *execution*. Real-time constructs therefore have no place in the kernel at all!

Session IV: REAL-TIME SOFTWARE DEVELOPMENT

Programming tools for real-time software development were the focus of this session.

Watson presented a number of metrics for measuring the "goodness" of architectures for real-time applications, with an eye towards embedded military systems. These included predictable response times, high utilisation of shared resources, dynamic reconfigurability, and a short software update cycle time (minimising re-testing after changes).

Venu Banda (Univ. of Michigan) pointed out the scarcity of debugging tools for real-time programs and discussed the well-known issues of reproducibility and intrusiveness in nondeterministic parallel program debugging. It was noted that the only way to avoid the "probe effect", i.e. the observer affecting the system under observation, was to use specialised hardware to satisfy our observability requirements.

Sang Son (Univ. of Virginia) emphasised the importance of timely response to data-base requests in real-time systems and proposed a prototyping tool for evaluating distributed database control mechanisms.

Hugh Sparks (MTS Systems Corp.) and Bob Chatham (BBN Advanced Computers) described a graphical programming tool for icon-based programming of real-time systems. Graphical objects are linked to produce flowchart-like programs. Successful applications to date include a laser welding robot and an automated visual inspection system. An implementation for the BBN Butterfly™ processor is under development.

Session V: REAL-TIME APPLICATIONS

A number of very lively presentations characterised this session.

Thomas Bihari (Adaptive Machine Technologies) compared two robotic control programs. The first was for the "Adaptive Suspension Vehicle". Funded by DARPA, this is a three ton, six-legged walking robot for rough terrain. Powered by an internal combustion engine, it houses 14 Intel 8086 processors on-board, pressure sensors in the hydraulic actuators and a laser range finder. A second robot project, the "High-Performance Manipulator", essentially a sophisticated robotic arm controlled by an Intel 80386, was also funded by DARPA. After experience with programming the ASV using a functional technique, it was decided to use object-oriented programming for the HPM. The latter was felt to provide a superior programming methodology.

Gianfranco Ciccarella (Univ. of L'Aquila) was the only overseas speaker (and indeed one of only a handful of overseas attendees), and presented a paper advocating the value of occam and the Transputer as a development environment for real-time controller software due to the close relationship between programming language and hardware.

Tom Ralya (IBM) described experiences with developing microprocessor control software for a LAN-network, in which the first draft only achieved 7% of the anticipated performance, but a software rewrite changed this to 430%! This was achieved by using a totally different approach to scheduling based on uninterruptible priority-scheduled "worksteps".

However, the overall prize for the most entertaining talk undoubtedly belongs to Tom Richardson (Martin Marietta Info. and Comm. Sys.) for his account of writing control programs for the Autonomous Land Vehicle. Funded by DARPA this project aims to produce a self-navigation system for

* UNIX is a registered trademark of AT&T

armoured vehicles. The prototype unit has both a colour camera (supported by a SUN workstation) and a laser range finder (supported by a WARP systolic array machine). Overall the unit houses 63 processors(!) and runs 150,000 lines of code (300,000 with vendor code). Two types of data were identified, refresh (in which case only the most recent copy need be kept), and multiple occurence. The presentation was enlivened by colour photographs, and several anecdotes of the project's early days, such as when it was necessary to wash the special test track, and to ensure that no shadows fell on it before the ALV could be "released" to guarantee that it could recognise the road! This situation has improved somewhat; the system can now navigate the course at any time (as long as it is daylight) and can avoid obstacles placed on the track.

Session VI: REAL-TIME DATABASES

The final brief session considered data base access with time constraints.

Jane Liu (Univ. of Illinois) formally looked at the problem of scheduling hard real-time tasks subject to the constraints of shared data access. A concept of temporal consistency, based on the "age" of data items, was introduced.

Susan Davidson (Univ. of Pennsylvania) cited the SDI problem of identifying a light source from space, as one in which data base accesses were subject to unpredictable deadlines. In these situations distributed relational database queries take the form of iterative computations whose accuracy improves with time. At one extreme an imprecise response returns a subset of the desired tuples, at the other, a superset. Techniques for deciding when such a partial answer was useful were discussed.

In addition to the above, two further panel sessions debated the problems associated with "Real-Time Applications" and "Building Real-Time Kernels".

CONCLUSION

Although a newcomer to this field, I found the papers presented and the discussion they sparked very interesting. The lack of concensus, and sometimes even clear-cut opinion, on many fundamental issues suggests that a considerable amount of experience still needs to be gained. Specific solutions to particular problems, rather than theory, seemed to dominate the proceedings. The number of papers with explicitly stated military goals also came as a surprise to your naïve correspondent.

The sixth workshop, to be held in Pittsburgh in 1989, will emphasise progress on the Ada* revision currently underway. One of the aims of the revision is to improve support for real-time programming in the language.

APPENDIX: Papers Presented

H. Tokuda and M. Kotera, *Scheduler 1-2-3: It's Better to be Predictable Than Ad Hoc*

I. Lee, R. Gerber and A. Zwarico, *Specifying Scheduling Paradigms for Time Dependent Processes*

S. Biyabani, J. Stankovic and K. Ramamritham, *The Integration of Deadline and Criticalness Requirements in Hard Real-Time Systems*

R. Rajkumar and J. Lehoczky, *Task Synchronisation in Real-Time Operating Systems*

D. Jameson, *ORE: Programming in Real-Time Applications*

K. Lin and S. Natarajan, *Refinement and Enhancement: Primitives for Monotonic Computations*

A. Gheith, P. Gopinath, K. Schwan and P. Wiley, *CHAOS and CHAOS-ART - Extensions to an Object-Based Kernel*

A. Mok, *Task Management Techniques for Enforcing ED Scheduling On Periodic Task Set*

J. Barr, *Co-Resident Operating System: Unix and Real-Time Distributed Processing*

L. Salkind, *The SAGE Operating System*

P. Watson, *An Overview of Architectural Directions for Real-Time Distributed Systems*

V. Banda and R. Volz, *Debugging Distributed Real-Time Software*

S. Son, *A Message Based Approach to Distributed Database Prototyping*

B. Chatham and S. Sparks, *Butterfly Hose: Graphical Programming for Parallel Systems*

T. Bihari, *Functional vs. Object-Oriented Development of Robot-Control Software (A Comparison of Two Robot-Control Programs)*

G. Ciccarella, *Design and Implementation of a Real-Time Multivariable Adaptive Controller*

T. Ralya, *Real-Time Operating System Architecture: Work-steps and Related Subjects*

T. Richardson and J. McSwain, *Real-Time Control of an Autonomous Land Vehicle*

J. Liu, K. Lin and X. Song, *Scheduling Hard Real-Time Transactions*

S. Davidson and A. Watters, *Partial Computation in Real-Time Database Systems*

* Ada is a trademark of the U.S. Department of Defence

PostScript On Screen: Here is the NeWS

by

Steve Holden
Sun Microsystems (UK) Ltd.
[Now: Desktop Connection Ltd.]

Sun Microsystems recently announced a system they call the Network/extensible Window System, or NeWS for short. There are several windowing systems currently available. The PC world uses Gem and Windows, and in the UNIX world there are packages called X Windows and Andrew.

So why bring out yet another windowing system? What's so special about NeWS? How will this help desktop publishers? To answer these questions we need to look at the way computing in general, and workstations in particular, are heading.

The Workstation Market

Windowing systems are produced for *workstations*, which in this context we define to be dedicated desktop computers with significant CPU power, local or networked storage and a high-resolution display. The high-end PCs challenge the performance of the low end of the workstation market, and desktop publishing is a growth area here too. Increasingly, workstations are installed with access to a local area network (LAN), which offers communications in the Mbits/second range.

The workstation market is growing very fast and, if we discount the MS-DOS PC low end, UNIX is the accepted operating system. Workstation vendors have traditionally provided a proprietary windowing system for their ranges of equipment, but this has disadvantages: particularly, software vendors have to maintain several different versions of their product for the various workstations, with consequent development and maintenance problems.

The logical response to this situation is to produce a standard which can be agreed by all parties. This was done for file access over a LAN about four years ago, when Sun introduced the Network File System. The "protocols", specifications of what messages need to be transferred to achieve file access, were put in the public domain, which allowed anybody to write their own NFS drivers. Sun also license their proprietary code, to make it easier to get a new NFS running. There are now over 100 NFS licensees including Dec, HP, Cray and Apollo.

Network Services

Like all good network services, NFS is independent of the hardware and the operating system it runs on. Buyers can, for example, add PCs to their network and use huge shared databases mounted from their mainframes. The independence from hardware is achieved by defining "server" processes which run on the mainframe, and "client" processes on the PCs which apply over the network to read and write files on the servers' disks.

Printing can also be thought of as a network service: a client desktop publishing system produces PostScript which is interpreted by a "print server" such as a LaserWriter. Once you realise that the image can be generated on one computer and rendered on another, it seems obvious that PostScript can be used to send images to computer screens as well as printers.

A desktop workstation running a PostScript server program can display images from many different sources on the network. NeWS allows this and offers a network service with advantages over conventional window systems. Neither Windows nor Gem offer direct support for distributed applications. X does, but it works in terms of bitmaps, and does not adapt well to the need to handle user input such as mouse movements and key depressions.

PostScript on a Network

NeWS gives the advantages of PostScript on a network. Rather than bulky bitmaps, fragments of PostScript are transmitted; these are usually much smaller than the bitmaps, and so are transmitted more quickly. Also PostScript can use tricks such as sending fragments with programmed loops rather than repeating instructions for each of a number of similar objects to be drawn.

The client program, generating PostScript image descriptions, does not need to be coded for the resolution of a specific device. The display hardware is of concern only to the server program which renders the images, and this can reasonably be assumed to handle its local display hardware correctly.

NeWS extends PostScript to make it a suitable language for responding to a mouse and keyboard. In the X way of doing things, when you click a mouse button on the server a message has to be sent across the network to the client program, which then needs to compute a response and ship bitmapped images back as a result. Even on a fast LAN this all takes time. Under NeWS the client, as a part of its initialisation, sends PostScript to the server which gets executed *on the server* when a button click occurs. The screen image can be modified locally without the need for network communications.

The server process which drives the screen can be one of many processes running on a desktop workstation, or the sole process under MSDOS or other such single-user system. The former approach gives true distributed computing, while the latter allows the PC to enter the networking world in fine style.

NeWS has several interesting wrinkles. Not only does it accommodate different screen resolutions, it can render colour images in colour on a colour screen, in dithered monochrome on a black-and-white screen, and in appropriate grey levels on a greyscale screen. A single application can thus drive all these types of hardware without modification. An arbitrary image (including scanned images) can be scaled and rotated through any angle.

Conclusion

NeWS has introduced new concepts in windowing systems. It has been demonstrated on Atari ST and Intel 80386-based hardware as well as Sun workstations, and the applicability of the ideas is now proved. The initial release offers X Windows emulation by PostScript code, although this will be moved into the server itself in later releases.

By using NeWS you do not confine yourself to using a single interface style imposed by a hardware or software vendor. The Macintosh and Windows environments can also be emulated, so programs can be run under the appropriate regime, and can even offer the user a choice of styles. The window manager is written in PostScript.

Best of all, anything that is displayed can be printed on a PostScript printer. No longer need applications struggle to produce hard copy. PostScript is set to become a standard in a previously unconsidered area.

Evolution of the SunOS Programming Environment

Robert A. Gingell

Sun Microsystems, Inc.
2550 Garcia Ave.
Mountain View, CA 94043 USA

ABSTRACT

Recent changes to Sun's implementation of the UNIX† operating system (SunOS‡) have provided new functionality, primarily file mapping and shared libraries. These capabilities, and the mechanisms used to build them, have made significant changes to the programming environment the system offers. Assimilating these new facilities presents many opportunities and challenges to the application programmer, and these are explored in this paper.

The new mechanisms also provide the application programmer with a flexibility comparable to that previously reserved for the operating system developer. Much of this flexibility is based on mechanisms for dynamic linking that support interposition. The future developments and ramifications of these mechanisms, as well as other areas for similar system refinements, are also explored.

1. Introduction

Several new capabilities have been added to Sun's implementation of the UNIX operating system (SunOS), most notably file-mapping and shared libraries. These capabilities not only introduce important new facilities, they also present new issues with which a programmer must be concerned, while at the same time enabling new performance and functional opportunities, and sometimes even making new classes of applications possible.

The manner in which these new capabilities have been incorporated into the system is also of interest. They are often applications of some more primitive and general underlying facility. These facilities are not necessarily new in the sense of being "original", all that is new is their implementation in SunOS. Their utility lies in being general, fundamental abstractions. They are interesting to the system architect because they provide a basis for a compact and efficient system implementation. However, they may be of more interest to an application developer, because the "system" functions they provide are but one of many possible applications for them – application areas now open to more than just "system programmers".

An examination of these facilities involves not only a description of how to use and work with them, but must also include the architectural and implementation decisions made in deciding how the system provides a given capability. For us, these decisions are driven by a desire to find the fundamental abstractions common to a group of related problems, and then address those problems uniformly from those fundamentals.

We do not claim there is anything particularly remarkable about these activities. Some would properly claim that they simply represent a restatement of precepts of long-standing programming disciplines or the "UNIX-philosophy." Where we believe we may be particularly successful is in our

† UNIX is a trademark of AT&T.

‡ SunOS is a trademark of Sun Microsystems, Inc.

consistency in following the abstraction process and in a willingness to reimplement extant parts of a system to maintain architectural integrity. This approach leads to a particularly powerful system architecture, one that more easily lends itself to future evolution than if we had provided a specific capability as a closed system addition. It also transcends arbitrary software boundaries such as "the kernel", instead representing a systemic philosophy of providing abstractions through basic mechanisms that are applied to create (or perhaps recreate) a specific system capability. Further, this approach helps retain the "essential character" of the UNIX system, allowing us to extend it and at the same time provide a powerful and efficient implementation of standard interfaces and thus gain from the large and growing body of UNIX software.

The sections that follow will briefly describe the new facilities present in SunOS. For the most part, these are more completely described elsewhere, and the discussion provided here is simply for background. Also examined will be the application of these facilities to recreate portions of the system based on new fundamentals, and the issues and potential difficulties the new facilities can present to application developers. Then our experiences in applying these facilities will be described, as well as a perspective on how they will evolve and the opportunities they present for application development. In addition, future opportunities for the evolution of the system will be explored.

2. New Virtual Memory System

The demands to support different forms of shared memory, mapped file access, shared libraries, and practical demands of increased system portability have led us to the development of a new Virtual Memory (VM) system for SunOS. The new system unifies all the system's operations on memory (including the "memory" in files) around the single notion of file mapping. The following summary covers the system's general concepts and how they were used to reimplement existing operations on memory. A more complete examination of the systems architecture and operations can be found in [GING 87a] and a detailed treatment of its implementation structure found in [MORA 88]. Conceptually, the system owes much to MULTICS [ORGA 72] and TENEX [MURP 72].

2.1. General Concepts

The new VM system provides a page-based facility in which the fundamental concept is file-mapping. The system's *virtual memory* consists of all its available physical memory resources, including local and remote file systems, pools of unnamed memory (*swap space*), and other random access memory devices. Named objects in the virtual memory are referenced through the UNIX file system. Previous SunOS work on file system interfaces permits many different implementations of file objects that are manipulated through an abstraction of the original UNIX *inode*, called a *vnode* [KLEI 86]. The object manager for a *vnode* is called a *Virtual File System* (VFS), and is itself an abstraction of the services required to implement a file system.

A process's *address space* is defined by mappings onto objects in the virtual memory. The mappings are constrained to be sized and aligned according to the page boundaries of the system on which the process is executing. Each page in an address space is independently mappable (or not), and thus the programmer may treat the address space as a simple vector of pages. A given process page may map to only one object, although a given object address may be the target of many process mappings. An important characteristic of a mapping is that the object to which the mapping is made need not be affected by the mere *existence* of the mapping. The implications of this are that it cannot, in general, be expected that an object has an "awareness" of having been mapped.[1] Establishing a mapping to an object simply provides the *potential* for a process to access or change the object's contents.

The establishment of mappings provides an *access method* that renders an object directly addressable within an address space. Unlike the access methods provided by *read* and *write* that require an application to operate only on a copy of object data (i.e., a program buffer), this method eliminates the inefficiency of copying while permitting the object to retain its identity during the access operation.

[1] It is not *prohibited* for an object to be aware of being mapped, it is simply not guaranteed that all objects *can* have such awareness.

The ability to directly access an object and have it retain its identity over the course of the access is unique to this access method, and promotes sharing of common code and data.

The VM system consists of programming that operates as a cache manager for data in the virtual memory. The physical resources for the cache are the processor's primary memory. References to VM objects result in either an access to a "cache entry" or the fetching of data into the cache, possibly requiring removal of other data. The latter two functions are simply "page-in" and "page-out".

An important characteristic of the system is the balance between the responsibilities of the VM system as cache manager, and those of the VFS that obtains data for filling a cache entry and to which data is passed when flushing an entry. This balance permits different handling of requests based on the object manager, where the differences may reflect either semantic changes or performance enhancements, or both. For example, predictive operations such as the old function of "read-ahead" are supported by object managers that "page-ahead" in response to sequential accesses.

2.2. Application in System Primitives

The most basic operation is that establishing a mapping between a process address and an object in the virtual memory. This operation is available through the *mmap* system call, fully specified in the system's architectural description and first described in [JOY 83]. It should be noted that the only memory that a process can address is that to which a mapping has been established.

mmap provides for two primary "types" of mappings: one, called MAP_SHARED, creates a mapping that lets store operations change the mapped object (i.e., the result of the "write" is shared with all users of the object). The second type of mapping, MAP_PRIVATE, creates a mapping that makes the changes made by store operations private to the address space containing the mapping. MAP_PRIVATE is often referred to as *copy-on-write*, reflecting a common implementation technique of intercepting the first store to a page, copying the page, and redirecting the original store and successive references to the copy.

The mapping abstraction for accessing memory has been used in a reimplementation of several UNIX kernel operations. These include *exec*, *fork*, and *brk*. Perhaps surprisingly, these also include *read* and *write*. The reimplemented operations still exist as system calls to retain a compatible interface for old applications. However, from a memory management perspective there is little motivation to implement them this way. The common uses of the *read*, *write*, and *brk* system calls in particular can easily be implemented in application code using *mmap*. And all of the functions of *exec* save the implied "overlay-and-jump" function could similarly be implemented outside of the kernel, a fact used in the development of shared libraries.

2.2.1. exec

exec overlays a process's address space with a new program to be executed by performing an internal version of *mmap* to the file containing the program.[2] The process's stack and uninitialized data areas are mapped to unnamed, zero-initialized storage. The mappings *exec* establishes are all the MAP_PRIVATE type.

The use of MAP_PRIVATE mappings simplifies the system call *ptrace*. Formerly, *ptrace* would refuse to deposit breakpoints or otherwise write on the text of a program executing in more than one process. With the changes to *exec*, this restriction has been removed: *ptrace* does its work by setting a text page writable, depositing its breakpoint, and restoring the write-protection. Because the page where the breakpoint is deposited is mapped MAP_PRIVATE, when *ptrace* stores into it, the store will really access a copy.

[2] The code and data in this file must be in the (default) "demand-page" format. Executables in other formats are handled by copying them to a paged format in mapped unnamed storage.

2.2.2. fork

Perhaps the most interesting application of the new facilities is in *fork*. In the new system, *fork* has been redefined. Formerly it would copy the address space of the parent to build the child. It now merely copies the mappings describing the address space. If a mapping was MAP_PRIVATE in the parent, it is MAP_PRIVATE in the child, and neither sees changes made by the other. Copies of address space data are made only if necessary, and then only a page at a time. Since most *forks* are followed immediately by an *exec*, this avoids wasting substantial effort in making copies that are never used.

If a parent mapping is MAP_SHARED, then the corresponding mapping in the child will also be MAP_SHARED. This provides the capability for parent and child to share memory after a *fork*, something that was not previously possible in Berkeley-based UNIX systems [JOY 83].

The new *fork* illustrates that it is possible to reimplement and even redefine standard operations while both maintaining compatibility and improving system functionality and performance. Although the definition of *fork* has changed to take advantage of a more general underlying memory management mechanism, its function to programs not using the new features of the system is completely compatible. And those programs that take advantage of functions not previously available (e.g., MAP_SHARED mappings of files) have a more powerful facility with which to work.

2.2.3. read and write

When mappable objects are accessed through the *read* or *write* system calls, the kernel performs the internal equivalent of an *mmap* to the kernel's address space to gain access to the file. This is followed by the appropriate *copyin* or *copyout* operation to drain or fill the caller's buffer. The changes to *read* and *write* unify the system's I/O functions on memory objects around the notions of mapping. Mapping access to files has not been some extra wart added on top of the system. Instead, the right abstraction has been identified and then used to ensure that comparable system functions could be expressed in terms of it – thereby simplifying the system and its implementation.

2.2.4. "Segments"

The new system provides a programmer with an address space that can be viewed as a simple vector of pages. The old notions of "text", "data", and "stack" segments no longer have any real meaning to the *implementation* of the system. The system as a whole retains the notions, but they are conventions imposed by the language tools, rather than a fundamental system-imposed constraint. For compatibility purposes, *exec* still establishes a range of memory to use as a heap, another for a stack, and *brk* adds or removes mappings to manage a "data segment". However, there is nothing to stop a program from having multiple non-contiguous heaps, or multiple text and data segments. The latter actually occurs when running programs constructed with shared libraries.

3. Dynamic Linking and Shared Libraries

The *text* of a program consists of some body of code that implements the function for which the program was written and also of code copied from libraries. Although UNIX has long supported sharing code among processes running the same program, the fact that nearly every program makes use of routines such as *printf* means that at any given time there are as many copies of these routines competing for system resources as there are different programs. As the body of UNIX programs grows, so does the percentage of system resources devoted to these copies. The notion of "shared libraries" attempts to extend the benefits of code sharing to processes executing *different* programs, by sharing the libraries common to them.

The following sections summarize Sun's approach to providing a shared library facility; a more complete treatment can be found in [GING 87b].

3.1. General Concepts

Shared libraries in SunOS are provided through the application of other mechanisms. These are:

- a revised system link editor (*ld*) that supports dynamic loading and binding;
- use of the file mapping facilities to introduce an object (i.e., a file containing a shared library) to an address space; and
- compiler changes to generate *position-independent* code (PIC).

It should be noted that the only one of these mechanisms provided by the UNIX kernel is file-mapping. This is simply the *mmap* function. There is *no* kernel support specific to the support of shared libraries.

3.1.1. ld

Many of the functions relating to shared libraries are embedded in the link-editor, *ld*. Conceptually, *ld* has been transformed from a just a batch utility that combines object files to a more persistent facility, available to perform link-editing functions at various times over the life of program. Previously *ld* built all programs *statically* – executable (*a.out*) files contained complete programs, with all code and data bound and relocated in a single batch operation. The new *ld* will build "incomplete" *a.out* files, deferring the incorporation (and binding) of certain object files until some later time (generally program execution). Still other bindings (procedure calls) are deferred until the object is first referenced.

The object files on which *ld* defers link editing are added to the address space at execution time using the system file mapping facilities to address and thus share these objects directly. These "shared object" (*.so*) files are simply executable files (demand page format) lacking an entry point.

Dynamic link editing is still the same operation as static link editing, all that has changed is the *time* at which it occurs. But, what a linker such as *ld* does is *edit*: it changes a program reference to resolve it to something a processor can execute directly. To change something involves writing on it, and to write on it means that it can no longer be shared. What has really been built with these facilities is not so much "shared libraries", but "*dynamic* libraries". "Shared" becomes a property of code that can be added to a program directly without any change, rather than a functional characteristic.

3.1.2. PIC

To make code more sharable, the C compiler was enhanced with an option to generate position-independent code (PIC). PIC does not require relocation to be incorporated into an address space, and is thus inherently sharable. This is accomplished by generating references to static storage as indirections through a *linkage table*. When *ld* link edits a PIC module, it builds this linkage table, initializing it with pointers that will eventually themselves be relocated.[3] The PIC references to the linkage table itself are relocated by *ld*, leaving pure ready-to-execute code behind.

Generally, *.so* files are built as PIC. At execution time all that need be relocated from these executables are their linkage tables. These are usually small relative to the entire object file. However, the use of PIC to avoid dynamic link editing operations is simply a *performance* optimization, *not* a functional one. While it will be shown that this optimization is important for effective system operation, not constraining the mechanisms to enforce sharing provides an important flexibility advantage.

3.2. Use of Shared Libraries

In addition to supporting static and dynamic linking, *ld*'s conventions regarding the interpretation of its -l *option were augmented. -l* is the shorthand reference for a library name, implying both a search path and a name format. For instance, a command line such as

[3] *ld* really builds two "linkage tables", one for data references and the other for procedure calls. The procedure call table is (usually) loaded with code and more properly thought of as a "jump table".

```
% cc -o ... -lx ...
```

formerly meant search for library x in a file named libx.a, located in one of several directories, specified explicitly or implicitly. The new *ld* performs this function as well, but expands on it to allow the library to have a different name, specifically libx.so. *If ld is enabled to perform dynamic linking (now the default), it will search for either the .a or .so, preferring the .so form if both are present. A "shared library" is thus simply a .so file containing the objects comprising the library, named according to the format of library file names, and placed in one of the standard directories.*

Since these mechanisms are applied after a program has been compiled or assembled, it follows that building a program with shared libraries involves no change to either the program nor the manner in which it is built. All that is required is to install a form of the library suitable for dynamic linking. In our system, for instance, there are both libc.a *and libc.so*. The *.so* form is almost universally used.

3.3. Version Control

To handle the independent evolution of shared libraries and the programs that use them, a version control scheme has been established. The *.so* files used as shared libraries really have a more complex name, involving a suffix that describes the version of the library contained in the file. *Interface version* "2" of the C library, in its third compatible *revision* would be placed in a *.so* having the name libc.so.2.3. *The suffix may be an arbitrary string of numbers in Dewey-decimal format, although only the first two components are significant to the operation of the link editors at present.*

The first component of the string (often called the "major version number") describes the library's interface, and the second component ("minor version number") documents implementation revisions to that interface. When an application is linked by *ld*, the interface numbers of each of the library *.so* files *ld* processed are recorded in the dynamic linking information retained in the resulting executable. At execution time, this information is used by the dynamic link editor to determine the "best" library to use in an environment that may contain multiple versions of a given library. The rules followed are:

- **Interfaces identical**: the interface used at execution time must exactly match the version found at *ld*-time. If an exact match cannot be found, the dynamic load will fail.
- **Most recent revision**: in the presence of multiple revisions of a given interface, the one with the highest revision number will be used. A warning is issued if a revision appears to have been deleted since the application was built, although execution will continue.

4. Issues

As with most changes in technology, the developments in SunOS bring both benefits and challenges. The benefits are largely in programmer abstractions that provide improved performance, increased flexibility, or greater functionality. The challenges are often less clear, and pose subtle (and therefore insidious) issues that, if ignored, can trap the programmer trying to exercise a newly-available tool. Such issues include:

- **Illusions of Compatibility.** A pair of implementations are often considered "compatible" if they present the same "interface". However, interfaces generally only describe a narrow set of positive assertions about a facility: what a user *can* assume about its use. It is rare, however, for an interface to specify what its users may *not* assume. Further, interface descriptions are generally devoid of non-functional issues such as performance.
- **Inflation of Responsibility.** More powerful programming tools, perhaps paradoxically, often bring with them the obligation to use increased thought and skill in their application. Correct and skilled use brings increased benefits, careless use merely brings more spectacular disasters.
- **Threats to Conventional Wisdom.** Programming habits are acquired through a series of experiences that train the programmer in "what works". However, new tools and facilities, particularly those that generalize or refine an abstraction or the environment in which they are applied, threaten those habits. A practice once considered acceptable may no longer be adequate because the problems to which it is applied may have scaled beyond its

applicability.

4.1. Working Set Size

One of the significant side-effects of the availability of shared libraries has been an increase in working set size [DENN 72]. This effect was expected, and has appeared in other implementations of shared libraries [ARNO 86]. The effect can be explained by the nature of a shared library: rather than including in the address space of a program only the library functions actually required during its execution, the *entire* library is provided. The (usually proper) subset of the library really needed is likely to be distributed over more pages, and thus the memory demand imposed by the program is greater.

Although more pages are required, as long as the pages are shared among multiple processes the incremental cost is decreased. This emphasizes the need to minimize the amount of *private* working set size, and suggests that the priorities for minimizing size are to first:

- reduce the percentage of the working set that involves physical memory unique to this process; and then
- reduce the shared physical memory requirements.

Experience has shown that the system is effective at handling pages that are truly shared, in that they are rarely removed from memory and thus impose little per-access cost. However, the per-process private pages must compete for the remaining physical memory resources with a comparable set of pages from other processes.

Although it has always been important for programs to impose a reasonable resource demand on the underlying system, the "working set" inflation created by shared libraries makes this a consideration not just of the application builder, but of the library builder as well. Although the functional requirements of the library have not changed (the interfaces are *compatible*), the performance side effects of this library "program" are not compatible with previous engineering considerations. It is, for instance, perfectly acceptable to ignore these issues and create a library that *works*, however the global impact of something like a poorly shared C library is potentially devastating.

4.2. Improving Sharing

The programmer can use several tools to increase the degree of sharing in a program or library and thus address the issues of working set size. Although these tools have always been available, the programmer may not have been sufficiently motivated to use them. In some (more insidious) cases, the changes to the system have caused the tools to be "broken" or otherwise changed in some non-functional way. For instance, the SunOS C compiler has supported an option (-R) to enter **initialized data in the text segment of an object file. When programs were linked with archive libraries containing such objects, the read-only data was shared among all the users of a program. However, with position-independent shared libraries, the use of an option such as -R is no longer as simple: a PIC module built with -R and containing an initialized array of character pointers will actually *worsen* the sharing of code, since the pointers in the initialized data will require relocation when the object is actually added to the program at execution.[4] While it is important to move the truly invariant data to sharable memory, it is also important to recognize what is really a piece of invariant data.**

Invariant data can also be obtained by recoding programs to use position-independent data (PID). Although a compiler can be changed to emit PIC, the position-dependence of static storage is a function of the program's algorithms. Clearly C is a language that favors a coding style using pointers, but often substantial efficiences can be gained if a data structure is accessed as PID, using relative addresses (array indices) to describe the desired address. The C treatment of array names as pointers makes this coding simple, if a little unnatural. Languages such as C++ [STRO 86] that contain support for overloaded operators such as '[]' **can make this more aesthetically satisfying.**

[4] An alternative to -R is to have a C compiler that supports the const storage class: when generating PIC for a const data definition the compiler could issue the pointers and invariant data under separate relocation counters.

The use of PID is also important for databases that are expected to be accessed through mapping operations. Such databases might include, for instance, character font descriptions for a bit-map display, and be used by many processes simultaneously. Having the data be PID allows the client applications to structure their address space around their application, rather than the requirements imposed by a common piece of data.

Finally, private data storage requirements can sometimes be lessened through the use of dynamically allocated global storage. This can be accomplished by removing static declarations and changing their references to access lazily *malloc*ed data. In a body of code such as the C library, where the average program uses little of the entire library, this can represent a substantial space savings. Further, the data that is used is generally allocated contiguously with other used data, often on the same page of the *malloc* heap.

4.3. Interfaces and Configuration Management

The availability of a version control mechanism for dynamically linked objects places a requirement on the programmer to recognize and manage library interfaces. The programmer's responsibilities range from simply updating version numbers appropriately to designing the interface to be more suitable for dynamic linking. In addition, the ability to selectively use the dynamic linking facilities may require some decisions about program configuration management with respect to dynamic linking.

4.3.1. Interfaces

Although the management of library interfaces has always been important, before the availability of dynamic linking an erroneous or unanticipated incompatibility would not break *existing* programs. With dynamically loaded objects, such errors are possible. However, they are also generally easier to detect earlier in the development cycle. Because the new code can be easily inserted into almost every program in a system, it becomes easier to perform large-scale testing.

A problem with interface management is that the languages and tools most programmers use do not provide much support for it. Further, the interface is more diffuse than just the functions, arguments, and results that one usually associates with a library. It extends to the shape (and sometimes content) of data structures shared by, or defined in, both applications and libraries. For example, if the size of a jmp_buf (as defined in the file setjmp.h) changes, then this constitutes a C library interface change. If the interface did not change, the extant applications containing instances of jmp_buf structures might fail if run with a C library that expected the differently sized jmp_buf. If the size the library expects is greater than that built in to the program, then the application will most likely malfunction as the library routines overwrite the area reserved in the application.

4.3.2. Configuration Management

The provider of a given application (or set of applications) may not wish to expose itself to changes made to libraries on which the application depends. Yet, the application may consist of application-specific libraries that are dynamically linked to simplify maintenance. Further, when an application *does* fail, the problem of determining the environment in which the failure occurred is more complex. Not only must one determine exactly what version of an application failed, it is important to know what versions of shared objects were involved. We have found it important to develop tools such as the program *ldd* (*l*ist *d*ynamic *d*ependencies) that displays the shared objects used to execute a given application. We have also modified other tools, notably debuggers, to interpret a program's dynamic configuration.

The apparent complexity of the configuration management issues is not so much a problem as it is a manifestation of an opportunity: vendors and computing suppliers now have a vehicle that supports multiple interfaces simultaneously. It has become more practical to ship "field-replaceable" software units as libraries, because a mechanism exists that no longer requires an "all or nothing" form of program replacement.

Further, the specification of interfaces at a dynamic linking level encourages innovation. Programmers modifying or enhancing the UNIX kernel have benefited for years from the ability to replace

the code behind an interface that was dynamically linked with applications. This flexibility has now been extended to more than just kernel developers – now virtually any programmer producing a general purpose facility has in the concept of a library much more than "a related collection of object files", there is an *interface* that exists independent of its implementation.

4.4. Foiled Assumptions: "Unexec"

Some facilities exist that are predicated on old conventions and practices. They presume a limited model in such a highly constrained manner that any attempt at generality foils them. An example is "unexec" utilities common in the building of programs that are "saved" in a memory-image initialized state. UNIX examples of such programs include TeX and some versions of the EMACS text editor.

These facilities were built on assumptions that the three segment memory model of a process was all there could ever be, something the use of shared libraries and dynamic binding invalidates. It is not that this function cannot be provided, simply that the considerations on which it is based need to be changed. It can be argued that "unexec" is simply the ability to save a program image, something that *ld* performs when finished link editing a program. This suggests that the dynamic abstraction of *ld* is incomplete, and that future work to complete it would include the capability for saving a program image.

5. Experiences and New Capabilities

The changes to the system brought about by the new VM and dynamic linking facilities, while creating sometime subtle changes in the programming environment that require increased programmer vigilance, have mostly brought about a more flexible environment in which it has become possible to vastly simplify or improve applications. In some cases, the changes have made it just possible to write a given application. In this section, we examine a few of these advantages to illustrate both their immediate value and their potential future impact.

5.1. Address Space Freedom

One of the goals of the new memory management facilities was to provide programmers with an address space that could be used flexibly. No longer does the operating system enforce any particular structure to the address space, nor does it impose any semantic requirement on any specific area of it. *Requirements* on address space structure have become *conventions* applied by language tools and utilities.

The resulting flexibility has simplified the implementation of shared libraries. A dynamically linked program consists of multiple text and data segments: one for each *a.out* file mapped into the program's address space. The notions of a "text" or "data" segments remain useful for describing the assembly of executable files, but no longer have a system interpretation. Without this generality, the mechanisms might have required specialized kernel support. We have often found that the imposition of a fixed semantic interpretation by a lower software layer eventually forms a barrier to flexibility.

Other uses of the unstructured address space have involved disjoint collections of data. Coroutine libraries creating multiple stacks now create them surrounded by "holes" in the address space that form *red zone* protection areas. Multiple storage heaps (either to improve locality or to isolate data for placement in stable storage) can also be easily established. The stable storage heap is simpler still: the stable storage is simply mapped into the address space for direct access.

5.2. Incremental Maintenance and Development

The dynamic linking of libraries permits easy incremental maintenance. A bug-fix to a library is easily incorporated into programs that use the library simply by installing the repaired version. This effect can be extended naturally to the program development process as well.

Developing a complex program involves many compile, link, and debug cycles. If the time taken to perform any of these operations can be shortened, programmer productivity will be improved. With the dynamic linking facilities in the new system, a crude form of incremental development is easily practiced. Consider a program consisting of many individual object files, such that the program is built

with

% cc -o prog main.o ... *many other .o files* ...

Using shared objects, it is possible to build a "linking hierarchy" such that for any one edit and compilation of a single object file, only a few objects need be processed by *ld* rather than the entire set.

By way of illustration, assume a program built from 100 object files. For the purposes of this example, let each object file be named from 00.o to 99.o. A possible hierarchy might be to collect the ten *.o* files with the same leading digit into a single *.so* file, such as with:

% ld -o 0.so 0?.o

The program would then be built with:

% cc -o prog 0.so 1.so ... 9.so

that creates prog from 10 *.sos* dynamically linked at execution. If at some later time, a bug requires the recompilation of 23.o, *then only the modules comprising 2.so* (or just 10% of the total number of *.o* files) need be relinked to incorporate the change.

Of course, each time that prog executes, it incurs the cost of dynamic linking. However, this cost is often negligible when compared with static linking, since the dynamic linking process does not require that a new output file be built and written to the file system. Judicious grouping of *.o* files may provide further efficiencies.

Other groupings are possible of course. In the limit, each single object could be built as its own *.so* file. In practice, there is most likely some middle ground between the one-to-one extreme and complete relinking.

5.3. Application Performance

The access method provided by mapping has two fundamental performance advantages over its counterparts *read* and *write*:

1.) a buffer copy operation is avoided, reducing the demand for processing power; and

2.) not having to support access to a second block of memory to contain the same information reduces the demand on the system's memory resources.

This suggests that programs in which buffer copying represents a significant fraction of the execution time can benefit from optimizing out the copy with mapped accesses.

cat and *cp* have been reimplemented to incorporate this optimization, and use *mmap* rather than *read* when accessing a mappable file. This has had the dual advantage of removing both copy overhead and reducing the number of system calls. These programs map a large section of the file being read (one megabyte) and write it out in a single operation. A code fragment from a very simplified version of a *cat* that simply maps the entire file is:

```
...

int     fd;             /* file descriptor */
struct  stat sb;        /* file status */
caddr_t cp;             /* file pointer */

...

/*
 * Take "fd" (opened for read), and map the entire length of
 * the file. Write it to standard output with a single write.
 */
(void) fstat(fd, &sb);
cp = mmap(0, sb.st_size, PROT_READ, MAP_SHARED, fd, 0);
if (cp != (caddr_t)-1)
        (void) write(1, cp, sb.st_size);
```

...

An interesting side effect of the use of mapping in these programs is that it provides a useful illustration of the "lazy evaluation" properties of the VM system. Directing their output to /dev/null will cause the program to take practically *no* time to execute, no matter how big the input file is. The reason for this is that *mmap* really performs no access on the data mapped, it merely describes to the system the potential for an access to occur. The driver for /dev/null *on the other hand, never performs the access. The above program, when asked to do nothing, really does nothing!*

5.4. Multithreaded I/O

If an application required the ability to perform multiple I/O operations in parallel, it had to be programmed using non-blocking I/O facilities and the *select* system call. Even with this, many I/O operations remained synchronous, since facilities such as the file systems did not support non-blocking I/O. With mapped files, the problems of initiating parallel I/O are even more acute, as I/O is only performed in response to a page fault, and such faults occur only on a single page at a time.

However, multiple processes can be created that are identically mapped (or have a significant overlap) in the construction of their address spaces. These processes can each perform their own I/O requests (or page faults) by dividing up the work between them. The maximum amount of parallelism is limited by the number of concurrent processes available.

This illustrates a crude example of multiple processes executing from a single (or heavily overlapped) set of memory objects. A more refined example is the notion of *lightweight processes* [KEPE 85], addressed in more detail further on. However, this example illustrates an important facet of the new system's architecture: the independence of address space contents from specific processes.

5.5. Global Performance Analysis

To support the development of the dynamic link editor, it became desirable to profile its execution. The granularity of the profiling process (generally at a line-clock rate), coupled with the short execution of the linker, required many samples. The problems of initializing and saving of profile buffers also presented a number of issues that were easily addressed with mapped files. A special version of the dynamic link editor was created that would map a specific file into its address space as part of its initialization. The kernel's profiling facilities were then directed to the mapped area. Other parts of the link editor accumulated other statistical information into the mapped area.

By mapping the file into the address space *shared*, all instances of the link editor in the system (nearly every process) contributed to the statistics collection in the shared file. At the end of a sample period of several hours, a significant set of information had been collected, one that almost certainly reflected an excellent description of how the link editor's time had been spent. The use of a mapped file obviated the need for the linker to engage in end-of-program activities to dump and merge the information with previous execution, and thus avoided any issues of saving the information from programs that did not end normally. Further, the information collected came from a large variety of programs, summed over the course of a long period of system execution. Although this example describes an implementation unique to the dynamic link editor, it suggests a general means of performing global instrumentation of a given piece of code, one that could benefit the construction of tools to improve the performance of programs on a global basis.

5.6. Interposition

A powerful aspect of the dynamic linking facilities is the ability to *interpose* targets for symbolic references. For example, consider the building of a program that uses a library, libinterpose, *supplied as a .so file:*

```
% cc -o prog prog.o -linterpose
```

This command invokes *ld*, and invisibly includes a reference to the C library after libinterpose. *libinterpose* defines entry points for *read* and *write* that in addition to performing the required system calls take statistics on the use of these system calls.

In this case, both libinterpose and libc define entry points for *read* and *write*, the latter being the standard entry points for the "stub" routines performing the respective system calls. In the presence of multiple definitions of a symbol, the link editor resolves the issue of which to select by using the ordering established when prog *was linked: since libinterpose* was specified first, its entry points are used for references to *read* and *write*. These entry points are used globally, so that *read* and *write* references made internal to the C library (such as from standard I/O routines) also use libinterpose.

Interposition offers the opportunity for programmers to provide "added value" to a "standard" system function by providing a new implementation of the function. In this respect, the act of interposing is the *control-flow* analog to the effect of UNIX I/O redirection on *data-flow*. An example of such a "value-added" feature is an instrumented *malloc* function, used to drive a graphical display termed a "mallometer". One approach to implementing this would be to replace *malloc* and build a new shared C library. While this would be more than sufficient, it presumes that it is possible to replace a single module in a shared library, which it is not. Further, the obligation to rebuild the library is unnecessarily cumbersome.

An easier approach is to "insert" a modified *malloc* into an already running program. This could be accomplished by building a *.so* out of just the modified *malloc* (say, malloc.so*) and providing it to programs with a sequence such as:*

```
% setenv LD_PRELOAD malloc.so
% prog
```

where LD_PRELOAD is interpreted by the dynamic link editor as a list of objects to be loaded before loading those requested by the program itself. The *malloc* defined by malloc.so *would thus take precedence over that in the C library.*

An even more flexible capability would be to allow a new *malloc* to simply "jacket" the "real" *malloc*, thereby not requiring any rewriting or reimplementation of the real function. The use of an interposing routine in this fashion can be viewed as the control-flow analog of a *filter* [RITC 74].

6. Trends and Future Developments

Development follows cyclical patterns: new developments bring new understanding that in turn brings further developments. The new facilities have brought with them both new capabilities and new requirements for their evolution. In areas such as dynamic linking, the application potential has barely been explored, as shared libraries represent but one potential use of the mechanisms. And, there are yet other areas of the system beyond memory and binding that can benefit from such evolution.

6.1. VM

Future developments of the VM system are likely to focus on the management policies of the system, such as the global page replacement algorithms and processor and memory scheduling. The likely outcomes of these developments include a page replacement policy supportive of a high degree of memory sharing, and an integrated process and memory scheduler. Support for program-directed performance functions such as "advising" the system as to expected process behavior (as in "these pages expected to be needed soon"), or "commanding" the system to have some behavior (as in page locking), are expected to be created from these efforts.

6.2. A Global View of Binding

Many problems in operating systems and programming languages reduce to issues of "binding". For example, the VM system implements a per-reference address binding operation using hardware assists such as memory management units supplemented with software interpretation on exceptional conditions (page faults). The "logical name" the program uses is simply a process address, and the VM facilities translate the name to some physical storage address, a binding that occurs on each memory reference. The existing link-editing mechanisms provide a "global symbol to process address" binding. The dynamic linking facilities have changed this activity from one that is "compiled" to one that is "interpretive". Finally, many problems in distributed computing, such load balancing, server selection, and failure containment, are problems of binding a client to a specific instance of a server.

The binding mechanisms implementing shared libraries do not yet offer any direct functional interface to the programmer, whose interface is through command-line interaction with *ld*. When contrasted with the VM changes, which provided functions to programmers in the form of system calls such as *mmap*, the dynamic binding mechanisms have at present but one application: shared libraries. The mechanisms themselves have not been made directly available – though this reflects product and business priorities rather than an architectural limitation. The global use of shared libraries suffices to establish the architecture into which new capabilities and interfaces can be easily introduced.

The existence of an interpretive binding facility in the programming environment architecture presents many opportunities for innovation not just by UNIX system vendors, but also by independent parties and individual programmers. These opportunities begin with the availability of a generalized set of link-editing functions and the definition of a programmatic interface by which they are accessed. Such an interface would include, but not be limited to functions that supported:

- **Object loading**: the addition of an object file to an ''address space''.
- **Image saving**: the *ld* function of producing an executable is generalized. This would obviate the need for special functions such as ''unexec'' for programs that wish to preserve a running image.
- **Symbol lookup**: obtain the value of a given symbol (perhaps syntactically as a pointer), or in the presence of multiple definitions of a given symbol, the value of any one or all of them.
- **Object ''unbinding''**: removal of an object from an address space.
- **Exception handling**: programmatic control of link editor exceptions such as references to undefined symbols.
- **Extensibility**: programmatic interpretation of link editor functions such as relocation.

In addition to their programmatic availability, such functions might also be invoked implicitly by the establishment of conditions through environment variable settings. The functions of interpretation could be enhanced with the (perhaps selective) use of interface descriptions maintained in upgraded object file formats.

These facilities could vastly simplify the use of link-editor properties such as interposition, thereby enabling them to be a more useful tool for the application developer. Consider the ''mallometer'' example described previously. With a function that performs symbol lookup, the ''value-added'' version of *malloc* could be written as:

```
char *
malloc(n)
        unsigned int n;
{
        char *cp;
        extern void *ld_lookup();

        ... accumulate statistics about n ...

        /*
         * Look up and call the "next" malloc().
         */
        cp = (char *)(*ld_lookup("malloc", "_next_"))(n);

        ... accumulate statistics about result ...
        return (cp)
}
```

This interposed *malloc* skeleton is an example of use of interposition to invoke the control-flow analog of a UNIX filter. ld_lookup *is a link editor function that returns a pointer to a (qualified)*

*function name. In this case, the qualifier was _next_*, a special qualifier meaning the "the *malloc* found by searching dynamically linked objects *after* this one in the symbol precedence order". For the sake of simplicity, disposition of failures has been omitted from the example.

However, since the link editor is interpretive, the reference to the "next" *malloc* might as easily have been coded as:

```
cp = _next_$malloc(n)
```

An unqualified reference to *malloc* would simply invoke the normal symbol lookup rules. Still other qualifiers could be used to identify "absolute" or "fully qualified" function or symbol names. For instance, if a program wanted to call the C language library routine called *malloc*, the reference could be:

```
cp = _c_$malloc(n)
```

Qualifiers need not be exclusively string based of course; these have merely been used in these examples as a suggestion of possible program references.

The interpretive nature of the binding process can also be integrated the activity with other program development facilities. At present, the exception handling facilities in the dynamic link editing process handle errors solely through program termination. Consider a program fragment containing an invocation of *printf* that is erroneously typed in as *pintf*. The execution of such a call would today cause the program to be terminated.[5] An alternative exception-handling facility could be used to intercept such references, and dispatch them to a debugger, perhaps automatically. The programmer could redirect the erroneous reference to *printf*, and with the assistance of sophisticated software-engineering tools update the source at the same time. In the event that the program referenced some truly undefined (as opposed to misspelled) symbol, an even more sophisticated software engineering environment could support the incremental addition of the missing symbol and the code or data it labeled to the program.

The interpretive nature of the binding process not only permits deferred and possibly interactive bindings, it also provides the opportunity to defer the *implementation* of the binding. For instance, using information either supplied through the environment or programmatically, the dynamic link editor could bind subroutine references not to the final target of the reference, but instead through some intermediate that performed some instrumentation such as "call graph profiling".

Alternatively, a function reference could be implemented as something other than a simple subroutine call instruction. This would allow system calls to be defined entirely as library function interfaces. Or, in a general network environment, and if coupled with additional support information such as interface descriptions in object files or configuration databases, the call could be implemented with a Remote Procedure Call (RPC) facility, thereby making the use of distributed resources transparent to the coding of the application and treating it as a problem in application configuration. In this case, the programming environment architecture has treated a symbolic reference as an abstraction that has its semantics defined through execution-time interpretation. In this respect, the symbolic reference is serving the same role for distributed computation as the UNIX file descriptor played in the implementation of transparently networked file systems: a handle for interpreted semantics.

6.3. Future Evolution: Asynchrony

UNIX systems have traditionally provided only one form of asynchrony: the process. Although Berkeley-based systems have introduced a form of asynchronous activity in the support of non-blocking I/O operations and the *select* system call, these are at best crude approximations to truly asynchronous operation. Several applications environments, notably those requiring response to multiple input stimuli (such as window systems or server processes of various kinds) or perhaps requiring real-time constraints, can profit from full support of asynchronous activities.

The traditional approach to providing asynchronous services in a system is to provide specialized interfaces that provide asynchronous operation. For instance, the support of asynchronous I/O

[5] In reality, *ld* anticipates this eventuality and reports it at the time the program is link-edited, thus it is difficult for this scenario to occur in the current system.

operations would be provided by variants on the system calls *read* and *write*. The variants would accept additional arguments describing the disposition of completed I/O requests, and would return almost immediately after being called. The actual transfer would complete some time later.

The implementation of such facilities is generally built to capture the parameters of the requested operation in some form of control block that is used to manage the real I/O activities. On completion, the control block contains a description of how to notify the invoking process that the operation has completed, and what its status was. Other asynchronous interfaces would be similarly constructed, each requiring a specialized control block to capture the required operation and allowing the invoking process to return.

The asynchronous "control blocks" created in support of these interfaces can be viewed as the representation of an extremely specialized "process". These specialized entities record the state of such processes throughout their "execution" (e.g., progress of I/O). Each new form of asynchrony to be added to the system usually involves the creation of a new form of such "processes". The specialized nature of their representation often makes each implementation ill-suited to the needs of any later requirement, and over time the system becomes an accretion of such specialized structures.

A view proving popular is that the notion of a "process" in UNIX is excessively "heavy": a "process" is more than just the desired parallel thread of control, it is also an address space, a range of descriptors, and a description of event handling among other things. An alternative notion is that of *lightweight processes* (*lwps*) [KEPE 85] [TEVA 87], in which the notion of a thread of control is broken out of its UNIX semantics and treated as a distinct entity in itself. A *lwp* can be created efficiently, and multiple *lwps* can coexist in the address space associated with a single "heavyweight" UNIX process.

In an environment supporting *lwp* constructs, it is not necessary to provide specialized interfaces to selectively support asynchronous operation. Any operation can be accessed asynchronously with inexpensively created and managed threads of control. The resulting system is more open to application expansion, as new asynchronous abstractions do not require the services of a "kernel programmer" or clumsy user-program approximations to avoid kernel changes. Note that use of a *lwp*-facility can be "hidden" behind implementations of asynchronous interfaces, and thus any present or future standards requiring the more traditional approach can be easily accommodated.

The programming facilities supported by the *lwp* model of computing provide additional benefits beyond a simple abstraction of asynchrony. *lwps* are "lighter" than traditional UNIX processes because they share heavyweight structures such as address spaces. With such sharing comes mechanisms to manage that sharing, such as *monitors* for the implementation of critical sections, *messages* (often using shared memory) for asynchronous interchange, and *condition variables* for synchronization. In turn, these mechanisms enable the building of reentrant and preemptable code, and render the resulting programming more suitable for execution in a multiprocessor, and more responsive to the preemption demands of real-time and interactive environments.

7. Conclusion

Several changes to the SunOS programming environment have been described. These changes represent a functionally compatible reimplementation of standardized or generally accepted UNIX interfaces. In addition, the changes enrich the environment by providing new capabilities that simplify, and sometimes even render possible, various applications. In still other cases, system and application performance is improved.

The architectural approach of providing simple, fundamental, and general abstractions as primitive mechanisms that are consistently applied has been quite successful. The system has been conceptually simplified while presenting applications programmers with more powerful facilities. In many cases, these programmers have obtained a flexibility previously available only to the "systems programmer". The increased flexibility is demonstrated particularly well with the interpretive binding facilities used to provide shared libraries. The general abstraction of binding mechanisms promises to provide an architectural cornerstone for a wide range of new applications.

These changes show that new implementations of standard functions based on new abstractions are both possible and effective. They also show that the standardizing of clean system *interfaces* is not necessarily a barrier to innovation and, in areas such as dynamic binding, even promotes innovative activity. The resulting and anticipated evolutions have substantially enriched the application programming environment available under SunOS for both ourselves and our user community.

8. Acknowledgements

The ideas and implementations presented here are the product of discussions with and work by many people at Sun. In addition to their contributions to the work itself, Xuong Dang, Jon Kepecs, Joe Moran, and in particular Glenn Skinner provided a great deal of support and assistance through their review of this paper. Other notable participants have included Evan Adams, Bill Shannon, and Richard Tuck.

9. References

[ARNO 86] Arnold, J. Q., "Shared Libraries on UNIX System V", *Summer Conference Proceedings, Atlanta 1986*, USENIX Association, 1986.

[DENN 72] Denning, P., S. C. Schwartz, "Properties of the working set model", *Communications of the ACM*, Volume 15, No. 3, March 1972.

[GING 87a] Gingell, R. A., J. P. Moran, W. A. Shannon, "Virtual Memory Architecture in SunOS", *Summer Conference Proceedings, Phoenix 1987*, USENIX Association, 1987.

[GING 87b] Gingell, R. A., M. Lee, X. T. Dang, M. S. Weeks, "Shared Libraries in SunOS", *Summer Conference Proceedings, Phoenix 1987*, USENIX Association, 1987.

[JOY 83] Joy, W. N., R. S. Fabry, S. J. Leffler, M. K. McKusick, *4.2BSD System Manual*, Computer Systems Research Group, Computer Science Division, University of California, Berkeley, 1983.

[KEPE 85] Kepecs, J. H., "Lightweight Processes for UNIX: Implementation and Applications", *Summer Conference Proceedings, Portland 1985*, USENIX Association, 1985.

[KLEI 86] Kleiman, S. R., "Vnodes: An Architecture for Multiple File System Types in Sun UNIX", *Summer Conference Proceedings, Atlanta 1986*, USENIX Association, 1986.

[MORA 88] Moran, J. P., "SunOS Virtual Memory Implementation", *Spring Conference Proceedings, London 1988*, European UNIX Users Group, 1988.

[MURP 72] Murphy, D. L., "Storage organization and management in TENEX", *Proceedings of the Fall Joint Computer Conference*, AFIPS, 1972.

[ORGA 72] Organick, E. I., *The Multics System: An Examination of Its Structure*, MIT Press, 1972.

[RITC 74] Ritchie, D. M., K. Thompson, "The UNIX Time-Sharing System", *Communications of the ACM*, Volume 17, No. 7, July 1974.

[STRO 86] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley Publishing Company, 1986.

[TEVA 87] Tevanian, A., R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper, M. W. Young, "Mach Threads and the UNIX Kernel: The Battle for Control", *Summer Conference Proceedings, Phoenix 1987*, USENIX Association, 1987.

SunOS Virtual Memory Implementation

Joseph P. Moran

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, CA 94043 USA

ABSTRACT

The implementation of a new virtual memory (VM) system for Sun's implementation of the UNIX† operating system (SunOS‡) is described. The new VM system was designed for extensibility and portability using an object-oriented design carefully constructed to not compromise efficiency. The basic implementation abstractions of the new VM system and how they are managed are described. Some of the more interesting problems encountered with a system based on mapped objects and the resolution taken to these problems are described.

1. Introduction

In December 1985 our group at Sun Microsystems began a project to replace our 4.2BSD-based VM system with a VM system engineered for the future. A companion paper [1] describes the general architecture of our new VM system, its goals, and its design rationale. To summarize, this architecture provides:

- Address spaces that are described by mapped objects.
- Support for shared or private (copy-on-write) mappings.
- Support for large, sparse address spaces.
- Page level mapping control.

We wanted the new VM system's implementation to reflect the clean design of its architecture and felt that basing the implementation itself on the proper set of abstractions would result in a system that would be efficient, extensible to solving future problems, readily portable to other hardware architectures, and understandable. Our group's earlier experience in implementing the *vnode* architecture [2] had shown us the utility of using object-oriented programming techniques as a way of devising useful and efficient implementation abstractions, so we chose to apply these techniques to our VM implementation as well.

The rest of this paper is structured as follows. Section 2 provides an overview of the basic object types that form the foundation of the implementation, and sections 3 through 6 describe these object types in detail. Sections 7 through 9 describe related changes made to the rest of the SunOS kernel. The most extensive changes were those related to the file system object managers. A particular file system type is used to illustrate those changes. Section 10 compares the performance of the old and new VM implementations. Sections 11 and 12 discuss conclusions and plans for future work.

2. Implementation Structure

The initial problem we faced in designing the new VM system's implementation was finding a clean set of implementation abstractions. The system's architecture suggested some candidate abstractions and examining the architecture with an eye toward carving it into a collection of objects suggested others.

---

† UNIX is a trademark of Bell Laboratories.

‡ SunOS is a trademark of Sun Microsystems.

We ultimately chose the following set of basic abstractions.

- The architecture allows for page-level granularity in establishing mappings from file system objects to virtual addresses. Thus the implementation uses the *page* structure to keep track of information about physical memory pages. The object managers and the VM system use this data structure to manage physical memory as a cache.
- The architecture defines the notion of an "address space". In the implementation, an *address space* consists of an ordered linked list of mappings. This level defines the external interface to the VM system and supplies a simple procedural interface to its primary client, the UNIX kernel.
- A *segment* describes a contiguous mapping of virtual addresses onto some underlying entity. The corresponding layer of the implementation treats segments as objects, acting as a class in the C++ [3] sense[1]. Segments can map several different kinds of target entities. The most common mappings are to objects that appear in the file system name space, such as files or frame buffers. Regardless of mapping type, the segment layer supplies a common interface to the rest of the implementation. Since there are several types of segment mappings, the implementation uses different *segment drivers* for each. These drivers behave as subclasses of the segment class.
- The *hardware address translation* (*hat*) layer is the machine dependent code that manages hardware translations to pages in the machine's memory management unit (MMU).

The VM implementation requires services from the rest of the kernel. In particular, it makes heavy demands of the *vnode* [2] object manager. The implementation expects the *vnode* drivers to mediate access to pages comprising file objects. The part of the *vnode* interface dealing with cache management changed drastically. Finding the right division of responsibility between the segment layer and the *vnode* layer proved to be unexpectedly difficult and accounted for much of the overall implementation effort.

The new VM system proper has no knowledge of UNIX semantics. The SunOS kernel provides UNIX semantics by using the VM abstractions as primitive operations [1]. Figure 1 is a schematic diagram of the VM abstractions and how they interact. The following sections describe in more detail the implementation abstractions summarized above.

3. *page* Structure

The new VM architecture treats physical memory as a cache for the contents of memory objects. The *page* is the data structure that contains the information that the VM system and object managers need to manage this cache. The *page* structure maintains the identity and status of each page of physical memory in the system. There is one *page* structure for every interesting[2] page in the system.

A *page* represents a system page size unit of memory that is a multiple of the hardware page size. The memory page is identified by a <*vnode*, offset> pair kept in the *page* structure. Each page with an identity is initialized to the contents of a page's worth of the *vnode*'s data starting at the given byte offset. A hashed lookup based on the <*vnode*, offset> pair naming the page is used to find a page with a particular name. The implementation keeps all pages for a given *vnode* on a doubly-linked list rooted at the *vnode*. Maintaining this list speeds operations that need to find all a *vnode*'s cached pages. *page* structures can also be on free lists or on an "I/O" list depending on the setting of page status flags. The *page* structure also contains an opaque pointer that the *hat* layer uses to maintain a list of all the active translations to the page that are loaded in the hardware. In the machine independent VM code above the *hat* layer, the only use for this opaque pointer is to test for NULL to determine if there are any active translations to the page. When the machine-dependent *hat* layer unloads a translation it retrieves the hardware reference and modified bits for that translation to the page, and merges them into machine-independent versions of these bits maintained in the *page* structure.

---

[1] Actually, as a class whose public fields are all virtual, so that subclasses are expected to define them.

[2] Pages for kernel text and data and for frame buffers are not considered "interesting".

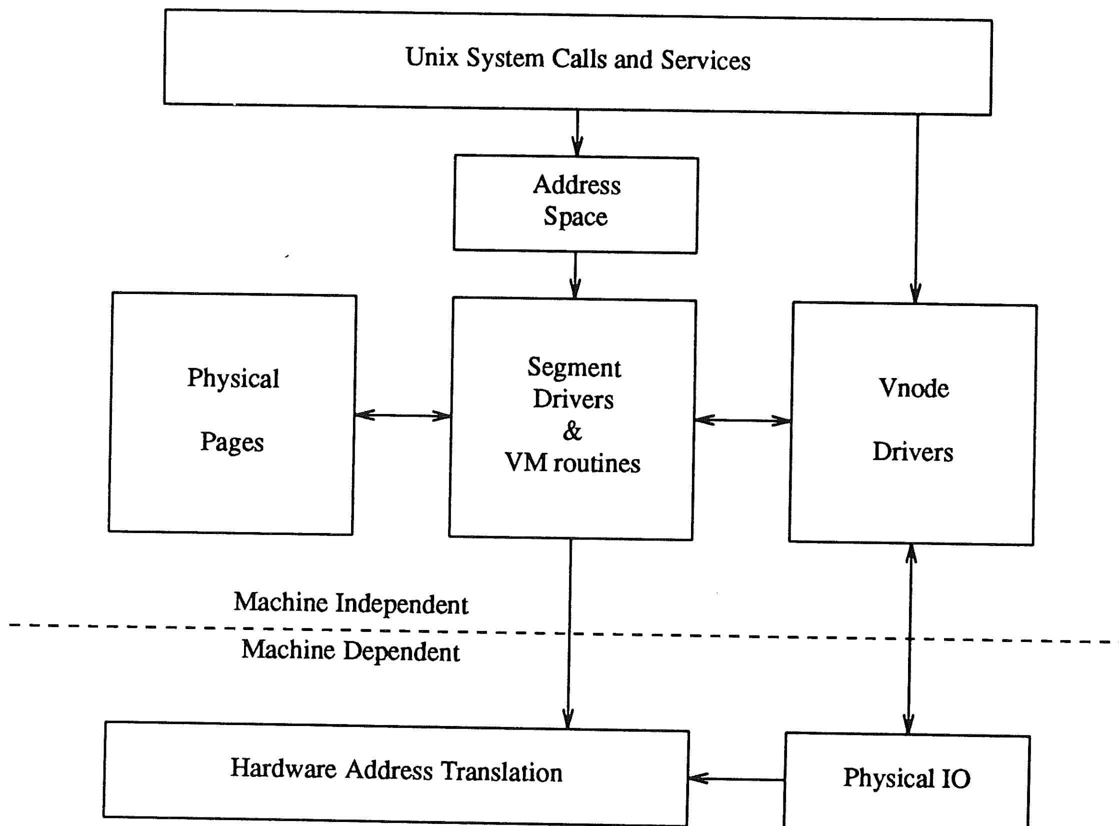


Figure 1

4. Address Space

The highest level abstraction that the VM system implements is called an *address space* (*as*), which consists of a collection of mappings from virtual addresses to underlying objects such as files and display device frame buffers. The *as* layer supports a procedural interface whose operations fall into two basic classes. Procedures in the first class manipulate an entire address space and handle address space allocation, destruction, duplication, and "swap out". Procedures in the second class manipulate a virtual address range within an address space. These functions handle fault processing, setting and verifying protections, resynchronizing the contents of an address space with the underlying objects, obtaining attributes of the mapped objects, and mapping and unmapping objects. Further information on these functions may be found in [1].

The implementation must maintain state information for each address space. The heart of this information is a doubly linked list of contiguous mappings (termed *segments* for lack of a better name) sorted by virtual address. Section 5 describes segments in detail. The *as* layer implements its procedural interface by iterating over the required range of virtual addresses and calling the appropriate segment operations as needed.

In addition, the *as* structure contains a *hardware address translation* (*hat*) structure used to maintain implementation specific memory management information. Positioning the *hat* structure within the *as* structure allows the machine dependent *hat* layer to describe all the physical MMU mappings for an address space, while the machine independent *as* layer manages all the virtual address space mappings. The *hat* structure is opaque to the machine independent parts of the system and only the *hat* layer examines it. Section 6 describes the *hat* layer in detail. The *as* structure also includes machine independent address space statistics that are kept separately from the machine dependent *hat* structure for convenience.

4.1. Address Space Management

The implementation uses several techniques to reduce the overhead of *as* management. To reduce the time to find the segment for a virtual address, it maintains a ''hint'' naming the last segment found, in a manner similar to the technique used in Mach [4]. Any time the *as* layer translates a virtual address to a segment, this hint is used as the starting point to begin the search.

Another optimization reduces the total number of segments in a given address space by allowing segment drivers to coalesce adjacent segments of similar types. This reduces the average time to find the segment that maps a given virtual address within an address space. By using this technique, the common UNIX *brk* (2) system call normally reduces to a simple segment extension within the process address space.

4.2. Address Space Usage

SunOS uses an *as* to describe the kernel's own address space, which is shared by all UNIX processes when operating in privileged (supervisor) mode. A UNIX process typically has an *as* to describe the address space it operates in when in non-privileged (user) mode[3]. An *as* is an abstraction that exists independent of any of its uses. Just as several UNIX processes share the same kernel address space when operating in supervisor mode, an *as* can have multiple threads of control active in a user mode address space at the same time. Future implementations of the operating system will take advantage of these facilities [5].

Most UNIX memory management system calls map cleanly to calls on the *as* layer. The *as* layer does not have knowledge of the implementation of the segment drivers below it, thus making it easy to add new segment types to the system. The *as* design provides support for large sparse address spaces without undue penalty for common cases, an important consideration for the future software demands that will be placed on the VM system.

5. Segments

A *segment* is a region of virtual memory mapped to a contiguous region of a memory object[4]. Each segment contains some public and private data and is manipulated in an object-oriented fashion. The public data includes the base and size of the segment in page-aligned bytes, pointers for the next and previous segments in the address space, and a pointer to the *as* structure itself. Each segment also contains a reference to a vector of pointers to operations (an ''ops'' vector) that implement a set of functions similar to the *as* functions, and a pointer to a private per-segment type data structure. This is similar to the way the SunOS *vnode* and *vfs* abstractions are implemented [2]. Using this style of interface allows multiple segment types to be implemented without affecting the rest of the system.

To most efficiently handle its data structures, a segment driver is free to coalesce adjacent segments of the same type in the virtual address space or even to break a segment down into smaller segments. Individual virtual pages within a segment's mappings may have varying attributes (e.g. protections). This design allows the segment abstraction control over the attributes and data structures it manages.

Of equal importance to what a segment driver does is what it does not do. In particular, we found that having the segment driver handle the page lookup operation and call the *vnode* object manager only when a needed page cannot be found was a bad idea. After running into some problems that could not be solved as a result of this split, we restructured the VM system so that the segment driver always asks the object manager for the needed page on each fault. Having the *vnode* object manager be responsible for the page lookup operation allows it to take action on each new reference.

5.1. Segment Driver Types

The implementation includes the following segment driver types:

seg_vn Mappings to regular files and anonymous memory.

---

3 Some processes run entirely in the kernel and have no need for a user mode address space.

4 Note that the name ''segment'' is not related to traditional UNIX text, data, and stack segments.

seg_map Kernel only transient <*vnode*, offset> translation cache.

seg_dev Mappings to character special files for devices (e.g. frame buffers).

seg_kmem Kernel only driver used for miscellaneous mappings.

The seg_vn and seg_map segment drivers manage access to *vnode* memory objects and are the primary segment drivers.

5.2. *vnode* Segment

The **seg_vn** *vnode* segment driver provides mappings to regular files. It is the most heavily used segment driver in the system.

The arguments to the segment create function include the *vnode* being mapped, the starting offset, the mapping type, the current page protections, and the maximum page protections. The mapping type can be shared or private (copy-on-write). With a shared mapping, a successful memory write access to the mapped region will cause the underlying file object to be changed. With a private mapping the first write access to a page of the mapped region will cause a copy-on-write operation that creates a private page and initializes it to a copy of the original page.

The UNIX *mmap* (2) system call, which sets up new mappings in the process's user address space, calculates the maximum page protection value for a shared mapping based on the permissions granted on the *open* of the file. Thus, the *vnode* segment driver will not allow a file to be modified through a mapping if the file was originally opened read-only.

5.2.1. Anonymous Memory

An important aspect of the VM system is the management of ''anonymous'' pages that have no permanent backing store. An anonymous page is created for each copy-on-write operation and for each initial fault to the anonymous clone object[5]. For a UNIX executable, the uninitialized data and stack are set up as private mappings to the anonymous clone object.

The mechanism used to manage anonymous pages has been isolated to a set of routines that provide a service to the rest of the VM system. Segment drivers that choose to implement private mappings use this service. The *vnode* segment driver is the primary user of anonymous memory objects.

5.2.1.1. Anonymous Memory Data Structures

The *anon* structure serves as a name for each active anonymous page of memory. This structure introduces a level of indirection for access to anonymous pages. We do not wish to assume that anonymous pages can be named by their position in a storage device, since we would like to be able to have anonymous pages in memory that haven't been allocated swap space. The *anon* data structure is opaque above the anonymous memory service routines and is operated on using a procedural interface in an object-oriented fashion. These objects are reference counted, since there can be more than one reference to an anonymous page[6]. This reference counting allows the *anon* procedures to easily detect when an anonymous page and corresponding resident physical page (if any) are no longer needed.

The other data structure related to anonymous memory management is the *anon_map* structure. This structure describes a cluster of anonymous pages as a unit. The *anon_map* structure consists of an array of *anon* structure pointers with one *anon* pointer per page. Segment drivers that wish to refer to anonymous pages do so by using an *anon_map* structure to keep an array of pointers to *anon* structures for the anonymous pages. These segment drivers lazily allocate an *anon_map* structure with NULL *anon* structure pointers at fault time as needed (i.e., on the first copy-on-write for the segment or on the first fault for an all anonymous mapping).

[5] The name of this object in the UNIX file system name space is **/dev/zero**.

[6] Typically from an address space duplication resulting from a UNIX *fork* (2) system call.

5.2.1.2. Anonymous Memory Procedures

There are two *anon* procedures that operate on the arrays of *anon* structure pointers in the *anon_map* structure. *anon_dup*() copies from one *anon* pointer array to another one, incrementing the reference count on every allocated *anon* structure. This operation is used when a private mapping involving anonymous memory is duplicated. The converse of *anon_dup*() is *anon_free*(), which decrements the reference count on every allocated *anon* structure. If a reference count goes to zero, the *anon* structure and associated page are freed. *anon_free*() is used when part of a privately mapped anonymous memory object is unmapped.

There are three *anon* procedures used by the fault handlers for anonymous memory objects. *anon_private*() allocates an anonymous page, initializing it to the contents of the previous page loaded in the MMU. *anon_zero*() is similar to *anon_private*(), but initializes the anonymous page to zeroes. This routine exists as an optimization to avoid having to copy a page of zeroes with *anon_private*(). Finally, *anon_getpage*() retrieves an anonymous page given an *anon* structure pointer.

5.2.2. *vnode* Segment Fault Handling

Page fault handling is a central part of the new VM system. The fault handling code resolves both hardware faults (e.g., hardware translation not valid or protection violation) and software pseudo-faults (e.g., lock down pages). The *as* fault handing routine is called with a virtual address range, the fault type (e.g., invalid translation or protection violation), and the type of attempted access (read, write, execute). It performs a segment lookup operation based on the virtual address and dispatches to the segment driver's fault routine, which is responsible for resolving the fault.

The *vnode* segment driver takes the following steps to handle a fault.

- Verify page protections.
- If needed, allocate an *anon_map* structure.
- If needed, get the page from the object manager.
  Call the *hat* layer to load a translation to the page.
- If needed, obtain a new page by performing copy-on-write.
  Call the *hat* layer to load a writable translation to the new page.

Some specific examples of *vnode* segment fault handling and how anonymous memory is used are given below.

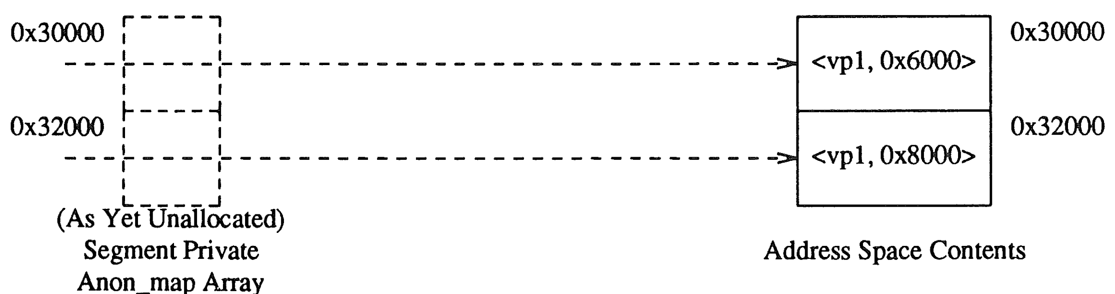


Figure 2
<vp1, 6000> Mapped Private to Address 0x30000 for 0x4000 Bytes

Figure 2 depicts a private mapping from offset 0x6000 in *vnode* vp1 to address 0x30000 for a length of 0x4000 bytes using a system page size of 0x2000. If a non-write fault occurs on an address within the segment, the *vnode* segment driver asks the *vnode* object manager for the page named by <vp1, 0x6000 + (*addr* - 0x30000)>. The *vnode* object manager is responsible for creating and initializing the page when requested to do so by a segment driver. After obtaining the page, the *vnode* segment driver calls the *hat* layer to load a translation to the page. The permissions passed to the *hat* layer from the *vnode* segment driver are for a read-only translation since this is a private mapping for which we want to catch a memory write operation to initiate a copy-on-write operation.

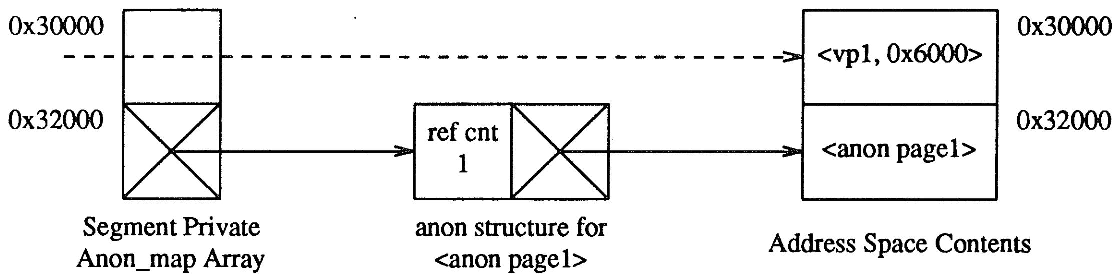


Figure 3
After Copy-On-Write Operation to Address 0x32000

Figure 3 shows the results of a copy-on-write operation on address 0x32000 in Figure 2. The *vnode* segment driver has allocated an *anon_map* structure and initialized the second entry to point to an allocated *anon* structure that initially has a reference count of one. The *anon_private* () routine has allocated the *anon* structure in the array, returned a page named by that *anon* structure, and initialized to the contents of the previous page at 0x32000. After getting the anonymous page from *anon_private* (), the *vnode* segment driver calls the *hat* layer to load a writable translation to the newly allocated and initialized page.

Note that as an optimization, the *vnode* segment driver is able to perform a copy-on-write operation, even if the original translation was invalid, since the fault handler gets a fault type parameter (read, write, execute). If the first fault taken in the *segment* described in Figure 3 is a write fault at address 0x32000 then the first operation is to obtain the page for <vp1, 0x8000> and call the *hat* layer to load a read-only translation. The *vnode* segment driver can then detect that it still needs to perform the copy-on-write operation because the fault type was for a write access. If the copy-on-write operation is needed, the *vnode* segment driver will call *anon_private* () to create a private copy of the page.

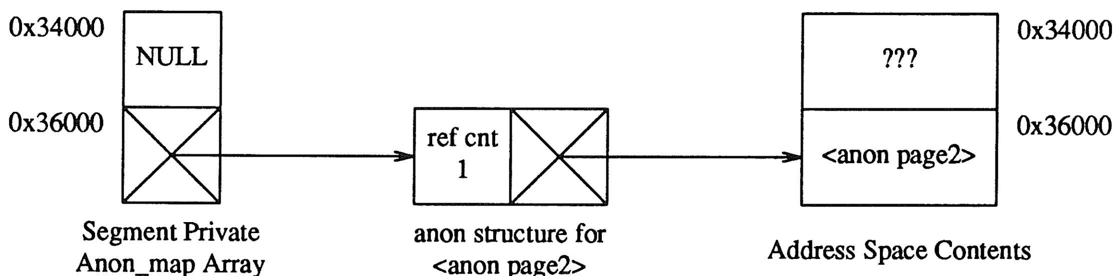


Figure 4
Private Mapping to /dev/zero for 0x4000 bytes at Address 0x34000
After a Page Fault at Address 0x36000

Figure 4 depicts a private mapping from the anonymous clone device **/dev/zero** to address 0x34000 for length 0x4000 after a page fault at address 0x36000. Since there is no primary *vnode* that was mapped, the *vnode* segment driver calls *anon_zero* () to allocate an *anon* structure and corresponding page and initialize the page to zeroes.

Figure 5 shows what happens when the mapped private *vnode* segment shown in Figure 4 is duplicated. Here both segments have a private reference to the same anonymous page. When the segment is duplicated, *anon_dup* () is called to increment the reference count on all the segment's allocated *anon* structures. In this example, there is only one allocated *anon* structure and its reference count has been incremented from one to two. Also, as part of the *vnode* segment duplication process for a privately mapped segment, all the *hat* translations are changed to be read-only so that previously writable anonymous pages are now set up for copy-on-write.

Figure 6 shows the result after the duplicated segment in Figure 5 handles a write fault at address 0x36000. When the segment fault handler calls *anon_getpage* () to return the page for the given *anon*

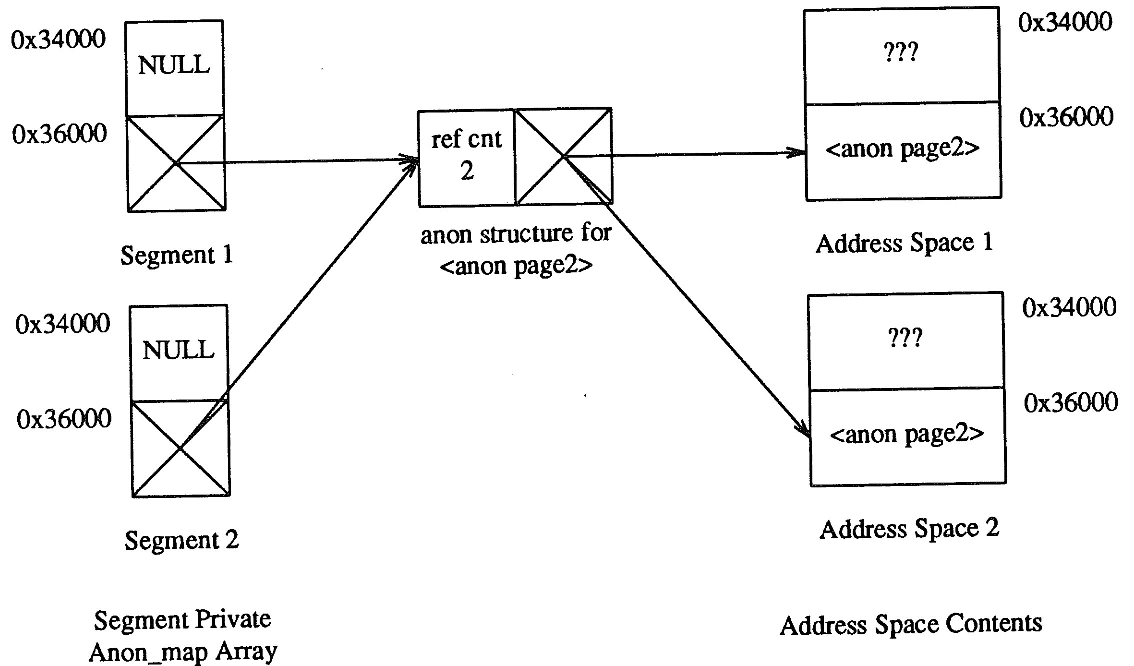


Figure 5
After the Private Mapped Vnode Segment is Duplicated

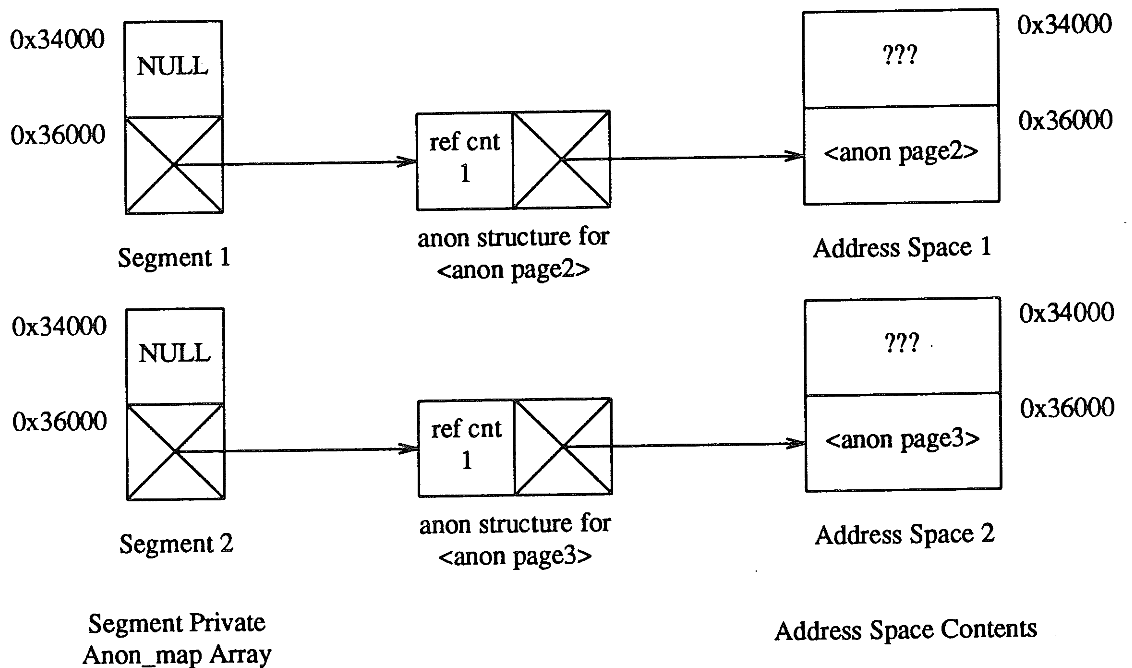


Figure 6
After Write Fault on Address 0x36000 in Address Space 2

structure, it will return protections that force a read-only translation since the reference count on the *anon* structure is greater than one. The segment driver fault handler will then call *anon_private* () to allocate a new *anon* structure and *page* structure and to initialize the page to the contents of the previous page loaded in the MMU. In contrast to the case depicted in Figure 3, *anon_private* () is copying from another

anonymous page and will decrement the reference count of the old *anon* structure after the *anon* pointer in the segment's *anon_map* array is changed to point to the newly allocated *anon* structure. Since the reference count on the original *anon* structure reverts to one, this means that the original segment will no longer have to do a copy-on-write operation for a subsequent write fault at address 0x36000. If a fault were to occur at 0x36000 in the original segment, *anon_getpage*() would not enforce a read-only mapping, since the reference count for the *anon* structure is now one.

5.3. Kernel Transient *vnode* Mapping Segment

The **seg_map** segment driver is a driver the kernel uses to get transient <*vnode*, offset> mappings. It supports only shared mappings. The most important service it provides to the *as* layer is fault resolution for kernel page faults. The seg_map driver manages a large window of kernel virtual space and provides a view onto a varying subset of the system's pages. The seg_map driver manages its own virtual space as a cache, so that recently referenced <*vnode*, offset> pairs are likely to be loaded in the MMU and no page fault will be taken when the virtual address within the seg_map segment are referenced.

This segment driver provides fast map and unmap operations using two segment driver-specific subroutines: *segmap_getmap*() and *segmap_release*(). Given a <*vnode*, offset> pair, *segmap_getmap*() returns a virtual address within the seg_map segment that is initialized to map part of the *vnode*. This is similar to the traditional UNIX *bread*() function used in the "block IO system" to obtain a buffer that contains some data from a block device. The *segmap_release*() function takes a virtual address returned from *segmap_getmap*() and handles releasing the mapping. *segmap_release*() also handles writing back modified pages. *segmap_release*() performs a similar function to the traditional UNIX *brelse*() / *bdwrite*() / *bwrite*() / *bawrite*() "block IO system" procedures depending on the flags given to *segmap_release*().

The seg_map driver is simply used as an optimization in the kernel over the standard *vnode* driver. It is important to be able to do fast map and unmap operations in the kernel to implement *read*(2) and *write*(2) system calls. The basic algorithm for the *vnode* read and write routines is to get a mapping to the file, copy the data from/to the mapping, and then unmap the file. Note that the kernel accesses the file data just as user processes do by using a mapping to the file. The *vnode* routines that implement read and write use *segmap_getmap*() and *segmap_release*() to provide the fast map and unmap operations within the kernel's address space.

5.4. Device Driver Segment

The **seg_dev** segment driver manages objects controlled by character special ("raw") device drivers that provide an *mmap* interface. The most common use of the seg_dev driver is for mapped frame buffers, though it is also used for mappings to machine-specific memory files such as physical memory, kernel virtual memory, Multibus memory, or VMEbus memory. This driver currently only supports shared mappings and does not deal with anonymous private memory pages. The driver is simple since it doesn't have to worry about a many operations that don't make sense for these types of objects (e.g., swap out). To resolve a fault, it simply calls a function to return an opaque "cookie" from the device driver, which is then handed to the machine-specific *hat* layer to load a translation to the physical page denoted by the cookie.

5.5. Kernel Memory Segment

The **seg_kmem** segment driver is an example of the use of a machine independent concept to solve a machine dependent problem. The kernel's address space is described by an *as* structure just like the user's address space. The seg_kmem segment driver is used as a catch-all to map miscellaneous entities into the kernel's address space. These entities includes the kernel's text, data, bss, and dynamically allocated memory space. This driver also manages other machine dependent portions of the kernel's address space (e.g. Sun's Direct Virtual Memory Access space [6]).

The seg_kmem driver currently only supports non-paged memory whose MMU translations are always locked[7]. In the previous 4.2BSD-based VM system, the management of the kernel's address space

[7] This means that the *hat* layer cannot remove any of these translations without explicitly being told to do so by the seg_kmem driver.

for things like device registers was done by calls to a *mapin*() procedure that set up MMU translations using a machine-dependent page table entry. For kernel and driver compatibility reasons, the seg_kmem driver supports a *mapin*-like interface as a set of segment driver-specific procedures.

6. Hardware Address Translation Layer

The *hardware address translation* (*hat*) layer is responsible for managing the machine dependent memory management unit. It provides the interface between the machine dependent and the machine independent parts of the VM system. The machine independent code above the *hat* layer knows nothing about the implementation details below the *hat* layer. The clean separation of machine independent and dependent layers of the VM system allows for better understandability and faster porting to new machines with different MMUs.

The *hat* layer exports a set of procedures for use by the machine independent segment drivers. The higher levels cannot look at the current mappings, they can only determine if any mappings exist for a given page. The machine independent levels call down to the *hat* layer to set up translations as needed. The basic dependency here is the ability to handle and recover from page faults (including copy-on-write). The *hat* layer is free to remove translations as it sees fit if the translation was not set up to be locked. There exists a call back mechanism from the hat layer to the segment driver so that the virtual reference and modified bits can be maintained when a translation is take away by the *hat* layer. This ability is needed for alternate paging techniques in which per address space management of the working set is done.

6.1. *hat* Procedures

Table 1 lists the machine independent *hat* interfaces. All these procedures must be provided, although they may not necessarily do anything if not required by the *hat* implementation for a given machine.

Operation	Function
`hat_init()`	One time *hat* initialization.
`hat_alloc(as)`	Allocate *hat* structure for *as*.
`hat_free(as)`	Release all *hat* resources for *as*.
`hat_pageunload(pp)`	Unload all translations to page *pp*.
`hat_pagesync(pp)`	Sync ref and mod bits to page *pp*.
`hat_unlock(seg, addr)`	Unlock translation at *addr*.
`hat_chgprot(seg, addr, len, prot)`	Change protection values.
`hat_unload(seg, addr, len)`	Unload translations.
`hat_memload(seg, addr, pp, prot, flags)`	Load translation to page *pp*.
`hat_devload(seg, addr, pf, prot, flags)`	Load translation to cookie *pf*.

Table 1
***hat* operations**

6.2. *hat* Implementations

Several *hat* implementations have already been completed. The first implementations were for the Sun MMU [6]. The MMUs in the current Sun-2/3/4 family are quite similar. All use a fixed number of context, segment, and page table registers in special hardware registers to provide mapping control. The Sun-2 MMU has a separate context when running in supervisor mode whereas the Sun-3 and Sun-4 MMUs have the kernel mapped in each user context. The maximum virtual address space for the Sun-2, Sun-3, and Sun-4 MMUs are 16 megabytes, 256 megabytes, and 4 gigabytes respectively.

Some machines in the Sun-3 and Sun-4 families use a virtual address write-back cache. The use of a virtual address cache allows for faster memory access time on cache hits, but can be a cause of great

trouble to the kernel in the old VM system [7]. Since the *hat* layer has information about all the translations to a given page, it can manage all the details of the virtual address cache. It can verify the current virtual address alignment for the page and decide to trade translations if an attempt to load a non-cache consistent address occurs. In the old 4.2BSD-based VM system the additional support needed for the virtual address cache permeated many areas of the system. Under the new VM system, support for the virtual address cache is isolated within the *hat* layer.

Other *hat* implementations have been done for more traditional page table-based systems. The Motorola 68030 has a flexible on-chip MMU. The *hat* layer chooses to manage it using a three level page table to support mapping a large sparse virtual address space with minimal overhead. The Intel 80386 also has an on-chip MMU, but it has a fixed two level translation scheme of 4KB pages. The problem with the 80386 MMU is that the kernel can write all pages regardless of the page protections (i.e., the write protection only applies to non-supervisor mode accesses)! This means that explicit checks must be performed for kernel write accesses to read-only pages so that kernel protection faults can be simulated. Another implementation has been done for IBM 370/XA compatible main frames. The biggest problem with this machine's architecture for the new VM system is that an attempted write access to a read-only page causes an protection exception that can leave the machine in an unpredictable state for certain instructions that modify registers as a side effect. These instructions cannot be reliably restarted thus breaking copy-on-write fault handling. The implementation resorts to special work arounds for the few instructions that exhibit this problem[8].

7. File System Changes

The VM system required changes to several other parts of the SunOS kernel. The VM system relies heavily on the *vnode* object managers, and required changes to the *vnode* interface as well as to each *vnode* object type implementation. It took us several attempts to get the new *vnode* interface right.

Our initial attempt gave the core VM code responsibility for all decisions about operations it initiated. We repeatedly encountered problems induced by not having appropriate information available within the VM code at sites where it had to make decisions, and realized that the proper approach was to make decisions at locations possessing the requisite information. The primary effect of this shift in responsibility was to give the *vnode* drivers control on each page reference. This allows the *vnode* drivers to recognize and act on each new reference. These actions include validating the page, handling any needed backing store allocation, starting read-ahead operations, and updating file attributes.

7.1. File Caching

Traditionally, buffers in the UNIX buffer cache have been described by a device number and a physical block number on that device. This use of physical layout information requires all file system types implemented on top of a a block device to translate (*bmap*) each logical block to a physical block on the device before it can be looked up in the buffer cache.

In the new VM system, the physical memory in the system is used as a *logical* cache; each buffer (page) in the cache is described by an object name (*vnode*) and a (page-aligned) offset within that object. Each file is named as a separate *vnode,* so the VM system need not have any knowledge of the way the *vnode* object manager (file system type) stores the *vnode*. A segment driver simply asks the *vnode* object manager for a range of logical pages within the *vnode* being mapped. The file system independent code in the segment drivers only has to deal with offsets into a *vnode* and does not have to maintain any file system-specific mapping information that is already kept in the file system-specific data structures. This provides a much cleaner separation between the segment and *vnode* abstractions and puts few constraints on the implementation of a *vnode* object manager[9].

The smallest mapping unit relevant to the VM system is a system page. However, the system page size is not necessarily related to the block sizes that a file system implementation might use. While we

[8] Such instructions are highly specialized and the standard compilers never generate them.

[9] We have taken advantage of this and have implemented several interesting *vnode* object managers that are nothing like typical file systems.

could have implemented a new file system type that used blocks that were the same size as a system page, and only supported file systems that had this attribute, we did not feel this was an acceptable approach. We needed to support existing file systems with widely varying block size. We also did not feel that it was appropriate to use only one system page size across a large range of machines of varying memory size and performance. We decided it was best to push the handling of block size issues into each file system implementation, since the issues would vary greatly depending on the file system type.

The smallest allocatable file system block is potentially smaller than the system page size, while the largest file system block may be much larger than the system page size. The *vnode* object manager must initialize each page for a file to the proper contents. It may do this by reading a single block, multiple blocks, or possibly part of a block, as necessary. If the size of the file is not a multiple of the system page size, the *vnode* object manager must handle zeroing the remainder of the page past the end of the file.

Using a logical cache doesn't come without some cost. When trying to write a page back to the file system device, the VOP_PUTPAGE routine (discussed below) may need to map the logical page number within the object to a physical block number, or perhaps to a list of physical block numbers. If the file system-specific information needed to perform the mapping function is not present in memory, then a read operation may be required to get it. This complicates the work the page daemon must do when writing back a dirty page. File system implementations need to be careful to prevent the page daemon from deadlocking waiting to allocate a page needed for a *bmap*-like operation while trying to push out a dirty page when there are no free pages available.

7.2. *vnode* Interface Changes

We defined three new *vnode* operations for dealing with the new abstractions of mappings in address spaces and pages. These new *vnode* operations replaced ones that dealt with the old buffer cache and the 4.2BSD-based VM system [2]. The primary responsibility of the *vnode* page operations is to fill and drain physical pages (page-in and page-out). It also provides an opportunity for the managers of particular objects to map the page abstractions to the representation used by the object being mapped.

The VOP_MAP() routine is used by the *mmap* system call and is responsible for handling file system dependent argument checking, as well as setting up the requested mapping. After checking parameters it uses two address space operations to do most of the work. Any mappings in the address range specified in the *mmap* system call are first removed by using the *as_unmap*() routine. Then the *as_map*() routine establishes the new mapping in the given address space by calling the segment driver selected by the *vnode* object manager.

The VOP_GETPAGE() routine is responsible for returning a list of pages from a range of a *vnode*. It typically performs a page lookup operation to see if the pages are in memory. If the desired pages are not present, the routine does everything needed to read them in and initialize them. It has the opportunity to perform operations appropriate to the underlying *vnode* object on each fault, such as updating the reference time or performing validity checks on cached pages.

As an optimization, the VOP_GETPAGE() routine can return extra pages in addition to the ones requested. This is appropriate when a physical read operation is needed to initialize the pages and the *vnode* object manager tries to perform the I/O operation using a size optimal for the particular object. Before this is done the segment driver is consulted, using a "kluster" segment function, so that the segment driver has the opportunity to influence the *vnode* object manager's decisions. The VOP_GETPAGE() routine also handles read-ahead if it detects a sequential access pattern on the *vnode*. It uses the same segment kluster function to verify that the segment driver believes that it would be worthwhile to perform the read-ahead operation. The I/O klustering and read-ahead conditions allow both the *vnode* object manager and the segment driver controlling a mapping onto this object to have control over how these conditions are handled. Thus, for these conditions we have set up our object-oriented interfaces to allow distributed control among different abstractions that have different pieces of knowledge about a particular problem. The *vnode* object manager has knowledge about preferred I/O size and reference patterns to the underlying object, whereas the segment driver has the knowledge about the view established to this object and may have advice passed in from above the address space regarding the expected reference pattern to the virtual address space.

The other new *vnode* operation for page management is VOP_PUTPAGE(). This operation is the complement of VOP_GETPAGE() and handles writing back potentially dirty pages. A flags parameter controls whether the write back operation is performed asynchronously and whether the pages should be invalidated after being written back.

The VOP_GETPAGE() and VOP_PUTPAGE() interfaces deal with offsets and pages in the logical file. No information about the physical layout of the file is visible above the *vnode* interface. This means that the work of translating from logical blocks to physical disk blocks (the *bmap* function) is all done within the *vnode* routines that implement the VOP_GETPAGE() and VOP_PUTPAGE() interfaces. This is a clean and logical separation of the file object abstractions from the VM abstractions and contrasts with the old 4.2BSD-based implementation where the physical locations of file system blocks appeared in VM data structures.

8. UFS File System Rework

Another difficult issue pertinent to the conversion to a memory-mapped, page-based system is how to convert existing file systems. The most notable of these in SunOS is the 4.2BSD file system [8], which is known in SunOS as the UNIX File System (UFS). The relevant characteristics of this file system type include support for two different blocking sizes (a large basic block size for speed, and a smaller fragment size to avoid excessive disk waste), the ability to have unallocated blocks (''holes'') in the middle of a file which read back as zeroes, and the need to *bmap* from logical blocks in the file to physical disk blocks.

8.1. Sparse UFS File Management

ufs_getpage () is the UFS routine that implements the VOP_GETPAGE() interface. When a fault occurs on a UFS file, the segment driver fault routine calls this routine, passing it the type of the attempted access (e.g., read or write access). It uses this access type information to determine what to do if the requested page corresponds to an as yet unallocated section of a sparse file. If a write access to one of these holes in the file is attempted, *ufs_getpage* () will attempt to allocate the needed block(s) of file system storage. If the allocation fails because there is no more space available in the file system, or the user process has exceeded its disk quota limit, *ufs_getpage* () returns the error back to the calling procedure which then propagates back to the caller of address space fault routine.

When *ufs_getpage* () handles a read access to a page that does not have all its disk blocks allocated, it zeroes out the part of the page that is not backed by an allocated disk block and arranges for the segment driver requesting the page to establish a read-only translation to it. Thus no allocation is done when a process tries to read a hole from a UFS file. However, an attempted write access to such a page causes a protection fault and *ufs_getpage*() can perform the needed file system block allocation as previously described.

8.2. UFS File Times

Another set of problems resulted from handling the file access and modified times. The obvious way to handle this problem is to simply update the access time in *ufs_getpage* () any time a page is requested and to update the modification time in *ufs_putpage* () any time a dirty page is written back. However, this approach has some problems.

The first problem is that the UFS implementation has never marked the access time when doing a *write* to the file[10]. The second problem is related to getting the correct modification time when writing a file. When doing a *write* (2) system call, the file is marked with the current time. When dirty pages created by the *write* operation are actually pushed back to backing store in *ufs_putpage* (), we don't want to override the modification time already stored in the *inode* [11].

To solve these problems, *inode* flags are set in the ''upper layers'' of the UFS code (e.g., when doing a file read or write operation) and examined in the ''lower layers'' of the UFS code (*ufs_getpage*() and *ufs_putpage*()). *ufs_getpage* () examines the *inode* flags to determine whether to update the *inode* 's access

[10] The ''read'' that is sometimes needed to perform a *write* operation never causes the file's access time to be updated.

[11] The *inode* is the file system private *vnode* information used by the UFS file system [2].

time based on whether a read or write operation is currently in progress. *ufs_putpage* () can use the *inode* flags to determine whether it needs to update the *inode*'s modification time based on whether the modification time has been set since the last time the *inode* was written back to disk.

8.3. UFS Control Information

Another difficult issue related to the UFS file system and the VM system is dealing with the control information that the *vnode* driver uses to manage the logical file. For the UFS implementation, the control information consists of the *inodes*, indirect blocks, cylinder groups, and super blocks. The control information is not part of the logical file and thus the control information still needs to be named by the block device offsets, not the logical file offsets. To provide the greatest flexibility we decided to retain the old buffer cache code with certain modifications for optional use by file system implementations. The biggest driving force behind this is that we did not want to rely on the system page size being smaller than or equal to the size of control information boundaries for all file system implementations. Other reasons for maintaining parts of the old buffer cache code included some compatibility issues for customer written drivers and file systems. In current versions of SunOS, what's left of the old buffer cache is used strictly for UFS control buffers. We did improve the old buffer code so that buffers are allocated and freed dynamically. If no file system types choose to use the old buffer cache code (e.g., a diskless system), then no physical memory will be allocated to this pool. When the buffer cache is being used (e.g., for control information for UFS file systems), memory allocated to the buffer pool will be freed when demand for these system resources decreases.

9. System Changes

With the conversion to the new VM system, many closely related parts of the SunOS kernel required change as well. For the most part time constraints persuaded us to retain the old algorithms and policies.

9.1. Paging

The use of the global clock replacement algorithm implemented in 4.2BSD and extended in 4.3BSD is retained under the new VM system. The "clock hands" now sweep over *page* structures, calling *hat_pagesync* () on each eligible *page* to sync back the reference and modified bits from all the hat translations to that page. If a dirty page needs to be written back, the page daemon uses VOP_PUTPAGE() to write back the dirty page.

9.2. Swapping

We retained the basic notion of "swapping" a process. Under the new VM system there is much more sharing going on than was present in 4.2BSD where the only sharing was done explicitly via the *text* table. Now a process's address space may have several shared mappings, making it more difficult to understand the memory demands for an address space. This fact is made more obvious with the use of shared libraries [9, 10].

The address space provides an address space swap out operation *as_swapout* () which the SunOS kernel uses when swapping out a process. This procedure handles writing back dirty pages that the *as* maps and that no longer have any MMU translations after all the resources for the *as* being swapped are freed. The *as_swapout* () operation returns the number of bytes actually freed by the swap out operation. The swapper saves this value as a working set estimate[12], using it later to determine when enough memory has become available to swap the process back in. Also written back on a process swap out operation is the process's user area, which is set up to look like anonymous memory pages.

The *as* and segment structures used to describe the machine independent mappings of the address space for the process are currently not swapped out with the process since we don't yet have the needed support in the kernel dynamic memory allocator. This differs from the 4.2BSD VM implementation where the page tables used to describe the address space are written back as part of the swap out operation.

---

12 Unfortunately, a poor one; this is an opportunity for future improvement.

9.3. System Calls

We rewrote many traditional UNIX system calls to manipulate the process's user address space. These calls include *fork, exec, brk,* and *ptrace.* For example, the *fork* system call uses an address space duplication operation. An *exec* system call destroys the old address space. For a demand paged executable it then creates a new address space using mappings to the executable file. For further discussion on how these system calls were implemented as address space operations see [1].

Memory management related system calls based on the original 4.2BSD specification [11] that were implemented include *mmap, munmap, mprotect, madvise,* and *mincore.* In addition, the *msync* system call was defined and implemented. For further discussion on these system calls see [1].

9.4. User Area

The UNIX user area is typically used to hold the process's supervisor state stack and other per-process information that is needed only when the process is in core. Currently the user area for a SunOS UNIX process is still at a fixed virtual address as is done with most traditional UNIX systems. However, the user area is specially managed so context switching can be done as quickly as possible using a fixed virtual address. There are several reasons why we want to convert to a scheme where the user areas are at different virtual addresses in the kernel's address space. Among them are faster context switching[13], better support for multi-threaded address spaces, and a more uniform treatment of kernel memory. In particular, we are moving toward a **seg_u** driver that can be used to manage a chunk of kernel virtual memory for use as u-areas.

10. Performance

A project goal for the new VM work was to provide more functionality without degrading performance. However, we have found that certain benchmarks show substantial performance improvements because of the much larger cache available for I/O operations. There is still much that can be done to the system as a whole by taking advantage of the new facilities.

Table 2 shows some benchmarks that highlight the effects of the new VM system and dynamically linked shared libraries [9, 10] over SunOS Release 3.2. *Dynamic* linking refers to delaying the final link edit process until run time. The traditional UNIX model is based on *static* linking in which executable programs are completely bound to all their libraries routines at program link time using *ld*(1).

Running a new VM kernel with same 3.2 binaries clearly shows that the new VM system and its associated file system changes has a positive performance impact. The effect of the larger system caching effects can be seen in the read times.

One way that the system uses the new VM architecture is a dynamically linked shared library facility. The *fork* and *exec* benchmarks show that the flexibility provided by this facility is not free. However, the benefits of the VM architecture that provides copy-on-write facilities more than compensate for the cost of duplicating mappings to shared libraries in the *fork* benchmark. The *exec* benchmark is the only test that showed performance degradation from dynamically linked executables over statically linked executables run with a SunOS Release 3.2 kernel. These numbers show that the startup cost associated with dynamically linking at run time is currently about 74 milliseconds. These results are preliminary and more work will be undertaken to streamline startup costs for dynamically linked executables. We feel that the added functionality provided by the dynamic binding facilities more than offsets the performance loss for programs dominated by start up costs.

11. Future Work

The largest remaining task is to incorporate better resource control policies so that the system can make more intelligent decisions based on increased system knowledge. We plan to investigate new management policies for page replacement and for better integration of memory and processor scheduling. We believe that the VM abstractions already devised will provide the hooks needed to do this. SunOS

---

[13] This is especially true with a virtual address cache and a fixed user area virtual address, since the old user area must be flushed before the mapping to the new user area at the same virtual address can be established.

Kernel Tested	SunOS 3.2	Pre-release New VM	Pre-release New VM
Binaries Executed	3.2	3.2	Dynamically Linked
Tests Performed	**Time (secs)**	**Time (secs)**	**Time (secs)**
exec 112k program 100 times	7.3	3.3	10.7
fork 112k program 200 times	8.8	4.4	7.7
Recursive stat of 125 directories	4.9	1.4	1.3
Page out 1 Mb to swap space	2.0	2.0	0.8
Page in 1 Mb from swap space	4.6	3.8	3.5
Demand page in 1 Mb executable	1.7	0.9	0.8
Sequentially read 1 Mb file (1st time)	1.6	1.5	1.5
Sequentially read 1 Mb file (2nd time)	1.6	0.4	0.4
Random read of 1 Mb file	5.7	0.7	0.8
Create and delete 100 tmp files	6.3	4.7	4.7

Table 2
System Benchmark Tests on a Sun-3/160
with 4 Megabytes of Memory and an Eagle Disk

kernel ports to different uniprocessor and multiprocessor machine bases will provide further understanding of the usability of the abstractions and our success in isolating machine dependencies.

Other future work involves taking advantage of the foundation established with the new VM architecture — both at the kernel and user level. Specialized segment drivers can be used at the kernel level to more elegantly support various unique hardware devices and to support new functionality such as mappings to other address spaces. Shared libraries are an example of the usefulness of mapped files at the user level. We expect to find the features of the new VM system used in various new facilities yet to be imagined. As new uses for the VM system are better understood, we can refine and complete the interfaces that have not yet been fully defined.

12. Conclusions

From our experience in implementing the new VM system, we draw the following conclusions.

- **Object oriented programming works.** The design of the new VM system was done using object-oriented techniques. This provided a coherent framework in which we could view the system.
- **The balance of responsibility is important.** When partitioning a problem amongst different abstractions, it is critical that the system be structured so that each abstraction has the right level of responsibility. When an abstraction gets control at the right time it has the opportunity to recognize and act on events that make sense for that abstraction.
- **The layering in the new VM system is effective.** For example, the *hat* layer provides all the machine dependent MMU translation control and has been found to be easily ported to new hardware architectures. The use of segment drivers has proven to make the system more extensible.
- **Performance did not suffer.** Although the new VM system provides considerably more functionality, it did so without any performance loss. Performance often improved because the new VM system better uses memory resources as a cache. By carefully designing the abstractions with optimizations for critical functions, we reduced the cost sometimes associated with object-oriented techniques to provide clean abstractions that are still efficient.

13. Acknowledgements

I would like to thank Rob Gingell, Dave Labuda, Bill Shannon, and especially Glenn Skinner, for reviewing this paper and helping to make it presentable and for their work with the new VM system. And most of all I would like to give a big thank you to my understanding wife Laurel, who continued to tolerate me during the countless extra hours I put into this project.

14. References

[1] Gingell, R. A., J. P. Moran, W. A. Shannon, "Virtual Memory Architecture in SunOS", *Summer Conference Proceedings, Phoenix 1987*, USENIX Association, 1987.

[2] Kleiman, S. R., "Vnodes: An Architecture for Multiple File System Types in Sun UNIX", *Summer Conference Proceedings, Atlanta 1986*, USENIX Association, 1986.

[3] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley Publishing Company, 1986.

[4] Rashid, R., A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, J. Chew, "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures", *Operating Systems Review*, Volume 21, No. 4, October, 1987.

[5] Kepecs, J. H., "Lightweight Processes for UNIX Implementation and Applications", *Summer Conference Proceedings, Portland 1985*, USENIX Association, 1985.

[6] Sun Microsystems Inc., *Sun-3 Architecture: A Sun Technical Report*, 1985.

[7] Cheng, R., "Virtual Address Cache in UNIX", *Summer Conference Proceedings, Phoenix 1987*, USENIX Association, 1987.

[8] McKusick, M. K., W. N. Joy, S. J. Leffler, R. S. Fabry, "A Fast File System for UNIX", *Transactions on Computer Systems*, Volume 2, No. 3, August, 1984.

[9] Gingell, R. A., M. Lee, X. T. Dang, M. S. Weeks, "Shared Libraries in SunOS", *Summer Conference Proceedings, Phoenix 1987*, USENIX Association, 1987.

[10] Gingell, R. A., "Evolution of the SunOS Programming Environment", *Spring Conference Proceedings, London 1988*, EUUG, 1988.

[11] Joy, W. N., R. S. Fabry, S. J. Leffler, M. K. McKusick, *4.2BSD System Manual*, Computer Systems Research Group, Computer Science Division, University of California, Berkeley, 1983.

;login:

The USENIX Association Newsletter

Volume 13, Number 3 **May/June 1988**

CONTENTS

Results of the Election for the USENIX Association Board of Directors 3
Know Your Board and Staff 4
San Francisco Conference June 20-24 5
Best Student Paper 5
Fifth Annual Computer GO Tournament 5
1988-89 USENIX Scholarship Winner 5
Call for Papers: UNIX Security Workshop 6
Call for Papers: Workshop on UNIX and Supercomputers 7
Call for Papers: C++ Conference 8
Call for Papers: Workshop on Large Installation Systems Administration 9
EUUG Autumn Conference 10
Addendum to the Computer Graphics Workshop Proceedings 10
LOCK/ix: An Implementation of UNIX for the LOCK TCB 11
Mark A. Schaffer and Geoff Walsh
An Update on UNIX Standards Activities 25
Shane P. McCarron
Additional Corrigenda to the 1987 Graphics Proceedings 29
Summary of the Board of Directors' Meeting Dallas, TX, February 7, 8 & 11, 1988 30
FYI 31
Financial Statements for the USENIX Association for Fiscal Year 1987 32
Future Events 35
Publications Available 35
4.3BSD Manuals 36
4.3BSD Manual Reproduction Authorization and Order Form 37
Local User Groups 38

The closing date for submissions for the next issue of *;login:* is June 24, 1988

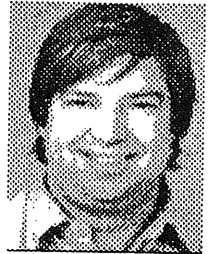
USENIX THE PROFESSIONAL AND TECHNICAL UNIX® ASSOCIATION

Know Your Board and Staff

Alan G. Nemeth
President

Deborah K. Scherrer
Vice President

Stephen C. Johnson
Treasurer


Rob Kolstad
Secretary

M. Kirk McKusick

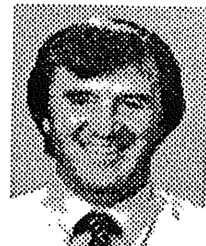
Sharon Murrel

Michael O'Dell

John S. Quarterman

Peter H. Salus
Executive Director

Judy DesHarnais
Conference Coordinator


John Donnelly
Tutorial/Exhibit Manager

Betty Madden
Office Manager

Emma Reed
Membership Secretary

Call for Papers
UNIX Security Workshop

Marriott Hotel
Portland, Oregon

August 29-30, 1988

Matt Bishop is chairman for the UNIX Security Workshop to be held in Portland, OR, on Monday and Tuesday, August 29th and 30th, 1988. This workshop will bring together researchers in computer security dealing with UNIX and system administrators trying to use UNIX in environments where protection and security are of vital importance. It is believed these people battle many of the same problems repeatedly and can share their unique solutions to some problems in order to avoid duplication of effort in making UNIX secure enough for their needs. It is intended that each participant will present briefly unique attributes of his/her environment and/or research and contribute a short (five minute) discussion (and paper) detailing some solution from their environment or work.

Some topics to be considered include: password security (password file integrity, enforcing choice of a safe password, spotting and handling crackers), network security (problems arising from logins over an unprotected ethernet, containing a break-in to one machine in a networked environment), file system security (auditing packages, security in an NFS environment), new designs to obtain C-level (or better) certification, making existing UNIX systems more secure, and locating and fixing UNIX security problems.

Workshop Format

This gathering will follow a "workshop" format rather than a "paper presentation" format. Each participant submits (electronically, to the address below) a one or two page single-spaced summary describing a solution to some problem from the topics above (or something equally interesting/important). Use the first paragraph to describe the properties of the environment and anything that makes it unique (e.g., distributed, large, supercomputers, mixed-vendors). Follow with a description of the problem and a description of the solution (detailed enough that fellow researchers and administrators can implement or use it). Also, please include with your submission a set of five (or so) topics that you'd like to hear about. It is possible that some participants will not present their papers at this first workshop.

The workshop chairman will collate the papers to schedule sessions which have appropriate audiences. It is anticipated that some sessions will include all 60-100 participants; some may require breaking into smaller groups. Send your submissions to Matt Bishop by noon EDT July 1, 1988.

For further details on the workshop:

Matt Bishop
Dept. of Mathematics and Computer Science
Bradley Hall
Dartmouth College
Hanover, NH 03755

(603) 646-3267
{ihnp4,decvax}!dartvax!bear!bishop
bishop%bear.dartmouth.edu@relay.cs.net

Call for Papers
Workshop on UNIX and Supercomputers

Westin William Penn Hotel
Pittsburgh, Pennsylvania

September 26-27, 1988

A large number of supercomputers are now or will in the future be running UNIX as their primary operating system. This is the first workshop to consider the general problems of running UNIX on supercomputers, and will cover topics both practical and abstract. Areas of specific interest include but are not limited to:

- Systems administration
- Archiving
- Scheduling
- File systems
- Networking and network protocols
- Job batching systems
- Monitoring performance/parallelism
- Programming languages and environments
- Fast file I/O
- Shared memory management
- IPC
- Very large files
- Checkpoint-restart

The workshop will include both shorter presentations and full-length papers, and there will also be tours of Pittsburgh Supercomputing Center and Westinghouse Energy Center facilities and a reception at the Pittsburgh Supercomputing Center. Workshop proceedings will be available at the Workshop.

If you are interested in presenting either a full paper or a brief discussion of your current work, please send an abstract of your paper or presentation to Melinda Shore by **July 15, 1988**. If you are sending your submission by US Mail, please send three copies. All submissions will be acknowledged.

Program Co-chairs:

Lori Grob
NYU Ultracomputer Research Lab
715 Broadway, 10th Floor
New York, NY 10003

(212) 998-3339
grob@lori.ultra.nyu.edu

Melinda Shore
Pittsburgh Supercomputing Center
4400 Fifth Avenue
Pittsburgh, PA 15213

(412) 268-5125
shore@reason.psc.edu

Call for Papers
C++ Conference

Denver, Colorado, October 17-21, 1988

USENIX is pleased to host its first full C++ conference in Denver, Colorado, Monday through Thursday, October 17-20. A one-day limited-enrollment implementor's workshop will follow the conference on Friday, October 21.

Papers are solicited on all aspects of C++, including:

applications
libraries
new or improved implementations
programming environments
case studies

We intend this conference to be interesting to a broad range of C++ users and potential users. Even if you have never written a C++ program, you will probably be able to learn enough from the tutorials to follow the technical sessions.

Tutorials, Monday and Tuesday

The tutorial program is ideal for people who have been thinking about using C++ but haven't had the opportunity to learn it. In addition, we expect to cover selected advanced topics for people who are already using C++.

Please contact the program chair if you are interested in giving a tutorial or have a topic you would particularly like to see covered.

Technical sessions, Wednesday and Thursday

One characteristic of C++ that stands out is the diversity of its applications and users. They range from microcomputers to large systems, from graphics and databases to real-time process control, from single-programmer efforts to large projects.

The technical sessions will present a cross-section of this diverse and rapidly growing community.

Implementor's Workshop, Friday

This small workshop is intended for people who are actively involved in C++ implementation. The workshop fee covers hotel accommodations for Thursday and Friday nights, meals through Friday lunch, and round-trip transportation leaving Denver after the main technical sessions and returning Saturday morning.

The workshop is primarily for speakers at this and last year's C++ meetings. If space permits, a few others will be admitted. If you don't want to speak at the conference, but wish to attend the workshop, let us know and submit a paper or abstract of your relevant work just as you would if you were interested in speaking at the conference. Details about the workshop will be sent with acceptance notices.

Program Committee

Andrew Koenig, AT&T, chair

Keith Gorlen, National Institutes of Health
Mark Linton, Stanford University
Richard Myers, Apple Computer
Peggy Quinn, AT&T
Mark Rafter, University of Warwick
Michael Tiemann, MCC

Extended abstracts (2-4 pages) or papers (9-12 pages) must be received, either electronically (preferred) or on paper, by Tuesday, **June 14.** Authors will be notified of acceptance by August 1 and must submit a full paper electronically (preferred) or in camera-ready form by August 30.

Send all submissions to Peter H. Salus at *usenix!peter* or the Association office.

Direct all queries about the technical program to:

Andrew Koenig
Room 4N-R12
AT&T Bell Laboratories
184 Liberty Corner Road
Warren, NJ 07060-0908

ark@europa.att.com or {attmail,research}!ark
+1 201 580 4127 (FAX)
+1 201 580 4883 (last resort)

Call for Papers
Workshop on Large Installation Systems Administration

Monterey, California

November 17-18, 1988

In light of last year's successful workshop on Large Installation Systems Administration, Alix Vasilatos will again be chairing a workshop on this subject in Monterey, CA on Thursday and Friday, November 17th and 18th, 1988. There is demonstrable benefit in bringing together system administrators of sites with 100 or more users (on one or more processors) to compare notes on solutions that they have found for a variety of common problems. These include but are not limited to:

Large file systems (dumps, networked file systems)
Password file administration
Large mail system administration
USENET/News/Notes administration
Heterogeneous environments (mixed vendor and/or version)
Load control and batch systems
Monitoring tools
Software release to multiple systems
Output device management

We are particularly interested in technical solutions to problems involving changes which directly affect users.

The workshop will focus on short papers and presentations. Please submit (electronically, to alix@athena.MIT.EDU) a one or two page single-spaced summary describing the solution to a problem. Include a description of the unique characteristics of the site, an outline of the problem, and a description of the solution (detailed enough that fellow administrators can implement it). Workshop proceedings will be available at the workshop.

The deadline for submissions is **September 15, 1988**. For further details about the workshop, contact:

Alix Vasilatos
MIT Project Athena
E40-357
1 Amherst Street
Cambridge, MA 02139

alix@athena.MIT.EDU
(617) 253-0121

For details about registration, contact the USENIX Conference Office.

EUUG Autumn Conference

Portugal

October 3-7, 1988

The Autumn '88 European UNIX systems User Group Technical Conference will be held in southern Portugal. Technical tutorials will be held on October 3 & 4, followed by the three day conference.

The theme of the conference is "New Directions for UNIX." Papers are expected on a wide variety of topics.

For further information about this and future EUUG events, contact the Secretariat.

Secretariat
EUUG
Owles Hall
Owles Lane
Buntingford, Herts. SG9 9PL UK
Phone: (+44) 763 73039
Fax: (+44) 763 73255 (G2)
Email: euug@insct.uucp

Tutorial Officer
Neil Todd
IST
60 Albert Court
Prince Consort Road
London SW7 2BH UK
Phone: (+44) 1 581 8155
Fax: (+44) 1 581 5147 (G3)
Telex: 928476 ISTECH G
Email: neil@ist.co.uk

Programme Chair
Peter Collinson
Computing Laboratory
University of Kent
Canterbury, Kent CT2 7NF UK
Phone: (+44) 227 764000, x7619
Email: pc@ukc.ac.uk

[This paper was inadvertently omitted from the Winter 1988 Conference Proceedings –PHS]

LOCK/ix: An Implementation of UNIX for the LOCK TCB

Mark A. Schaffer
Honeywell
Secure Computing Technology Center
2855 Anthony Lane South - Suite 130
Minneapolis, MN 55418
(612) 782-7134

Geoff Walsh
R&D Associates
4640 Admiralty Way
Marina del Rey, CA 90295
(213) 822-1715

ABSTRACT

The LOgical Coprocessing Kernel (LOCK) is a Trusted Computing Base (TCB) that is designed to meet and exceed the requirements for a Class A1 secure system. This paper describes the results of a study that determined how to port the UNIX System V Operating System to the LOCK TCB, while maintaining maximum compatibility with the System V Interface Definition (SVID) [SVID86].

1. Background of the Problem

Over the years, UNIX has gained widespread acceptance as the de facto standard Operating System (OS) within the U.S. Government and private industry. During the same time that UNIX has gained in popularity, a demand for secure computing systems has developed. Recently, the demands for these two technologies have created a demand for secure UNIX systems within the user community.

To meet the demand for secure UNIX systems, we decided to port UNIX to LOCK rather than develop a new OS. This is very appealing from a practical point of view because a large amount of portable UNIX applications already exists.

1.1. Background of the Solution

Traditional approaches to providing Multi-Level Secure (MLS) computing systems have emphasized implementing software security kernels that run in the target processor's privileged mode. In some cases, security has been provided by redesigning the OS. These purely software approaches to providing multi-level security have four primary disadvantages:

1. DECREASED ASSURANCE since a software malfunction could cause total security failure,

2. DECREASED PERFORMANCE to usually unacceptable levels because of the high overhead incurred by performing the security access checks in software,

3. LOSS OF EXISTING APPLICATION SOFTWARE because of the extensive redesign of the operating system, and

4. INABILITY TO FUNCTIONALLY ENHANCE the OS without requiring expensive and time-consuming re-verification and revaluation [SAYD87].

The LOCK TCB is a MLS computing system currently being prototyped at Honeywell Secure Computing Technology Center. It has been designed to meet and exceed the requirements for a Class A1 system as defined in the DoD Trusted Computer System Evaluation Criteria (the Orange Book) [TCSEC85].

This effort has been supported by National Computer Security Center contract MDA904-87-C-6011.

LOCK is the third phase of a continuing project previously called the Secure Ada Target (SAT), which was started by Honeywell in 1982. The first phase of the SAT program (SAT-0) resulted in a high-level requirements specification [HONE83]. The second phase (SAT-I) resulted in an intermediate specification [HONE86]. The third phase (SAT-II), renamed LOCK, will result in a detailed design specification and MLS mini-computer prototype in 1990 [SAYD87].

1.1.1. The LOCK Solution to Multi-Level Security

The LOCK system takes a hardware-oriented approach to providing a MLS computing system which should enable the system to overcome the disadvantages associated with purely software approaches.

The security policy of the system is enforced by a physically separate, multi-processor, coprocessing unit called the System-Independent, Domain-Enforcing, Assured, Reference Monitor (SIDEARM). The SIDEARM has its own processors, memory, and mass storage. All security-related data is stored on the SIDEARM mass storage unit. All access decisions and computations are performed by the SIDEARM.

The physical separation of the protection-critical from the non protection-critical elements in the LOCK system makes it physically impossible for a user process to access or tamper with the SIDEARM firmware or its data, giving the LOCK system a high degree of assurance.

The host processor provides TCB-mediated resource management and computing power for user applications. Since it performs no security access checks, the performance degradation imposed on the system by the security mechanisms should be minimal.

The OS for LOCK will not be responsible for enforcing the security policy of the system, and therefore, it will not be part of the TCB and not have to be verified or evaluated when it is updated.

Since UNIX is not part of the TCB, we will not have to modify it to provide MLS features. These capabilities are provided by the underlying TCB. Because of this, we should be able to maintain a great deal of compatibility with the SVID and, hence, with the existing base of applications.

1.2. The Study Goals and Results

During 1987, we performed a study of the UNIX kernel to determine if it could be (relatively easily) ported to the LOCK system, and if so, determine what the effect on the interfaces be. To enable us to determine if it would be worthwhile to port UNIX, we established the following research goals:

- The number of modifications to the UNIX kernel should not be extensive.
- The TCB could not be modified to "tailor" it to running UNIX.
- UNIX had to be able to service many concurrent users running at different security levels without becoming part of the TCB.
- The file system had to be able to manage data at different security levels requiring trusted servers and without introducing covert channels.
- The resultant system must maintain a maximum compatibility with the SVID.

The results of our study indicate that these goals can be met. The application-visible interface to the LOCK implementation of UNIX (LOCK/ix) is nearly identical to that of a standard implementation of UNIX System V. The security policy enforced by the underlying LOCK TCB should have little, if any, impact on the majority of existing UNIX applications.

We feel one major result of the study is our approach for implementing an untrusted file system that will manage the multi-level data. Internally, our file system implementation will be quite different than in a standard UNIX kernel. However, users and applications should not notice the differences.

1.3. Overview of the LOCK Architecture

The LOCK system consists of two computing units: the SIDEARM and the host processor. The majority of the TCB functionality resides in the SIDEARM, whose firmware coordinates with a small (TCB) software kernel (the Supervisor) that runs on the host processor.

The resultant TCB provides low-level services for subject, object, and device management. The LOCK TCB is restricted, for reasons of verifiability, to minimum functionality. It is intended to support, not replace, traditional OS services, such as a hierarchical file system.

The Supervisor (see Figure 1) functions as a low-level resource manager, and provides an application visible interface to the TCB's capabilities. The Kernel Extensions are a set of verified, security-related utilities whose capabilities cannot be provided by the SIDEARM in a generic fashion.

1.3.1. The SIDEARM

The SIDEARM implements what is called the Reference Monitor (RM) concept (see Figure 2). In general, an RM can be thought of as a guard between people and the information they would like to access. There are three important criteria for an RM:

1. It must always be invoked.
2. It must be verified to be correct (i.e., properly enforce the security policy of the system).
3. It must be tamperproof.

The LOCK hardware-oriented approach (see Figure 3) provides a good match to the RM model [SAYD87].

When the system is booted, the SIDEARM is booted and initialized before the host processor begins to run and continues to run until the system is shut down. All security-related data and most of the security functionality is implemented in the SIDEARM, thus making it possible to verify that it is correct. And finally, since the SIDEARM is physically separate (see Figure 4) and maintains its own memory, there is no (physical) way for a user process to tamper with its firmware or data. It is unbypassable since it is the SIDEARM, and not the Host processor, that has exclusive control over the Memory Management Unit (MMU).

1.3.2. The Host Processor

As mentioned previously, a small software kernel (which is part of the TCB) runs on the host processor. This software kernel is responsible for preserving the security policy of the system by performing correct, low-level resource management. This software kernel, called the Supervisor, consists of code that runs in both privileged and user mode of the host processor.

The portion of the Supervisor that runs in the privileged mode is only that code which is forced there by the hardware, such as the interrupt handlers. Other code, such as the subject scheduler, runs in the processor's user mode.

All code that runs in privileged mode will be placed in Read-Only Memory (ROM)) that is addressable only when the processor is running in privileged mode, thereby making it tamperproof. Other software, such as the OS, will run in user mode on the host processor.

2. Overview of the LOCK Security Model

The LOCK TCB enforces a MLS policy. The policy is enforced by mediating access between subjects, the active entities of the system, and objects, the inactive entities of the system.

To enforce this policy, the SIDEARM maintains a large database called the Global Object Table (GOT). Each time a subject or object is created, it is assigned a unique identifier (UID). A GOT entry is then created for the new entity where the UID is used as the primary key. A GOT entry will contain additional information, such as the level and the creator.

The LOCK TCB provides Discretionary Access Control (DAC) and Mandatory Access Control (MAC) mechanisms to enforce the system's security policy. In order for a subject to be granted access to an object, the request

must be allowed by both the DAC and MAC mechanisms of the system.

2.1. Discretionary Access Control Policy

A DAC policy is discretionary because its administration is up to the discretion of the system users. The LOCK TCB provides Access Control Lists (ACLs) as the mechanisms for providing DAC.

ACLs allow a user to specify, for each named object he owns, a list of named individuals and a list of groups of named individuals and their respective modes of access to the object. Additionally, for each named object a user owns, he may specify a list of named individuals and a list of groups of named individuals for which no access to the object is to be given. The currently supported modes of discretionary access are read (r), write (w), execute (x), and null (n).

2.2. Mandatory Access Control Policy

A MAC policy is mandatory because it is always enforced by the system. Unlike a DAC policy, the system users have no say in how the policy is administered. The LOCK MAC policy is enforced by Labeled Security Protection and Type Enforcement mechanisms.

2.2.1. Labeled Security Protection

The LOCK TCB enforces Labeled Security Protection as required by the Orange Book. The policy is enforced over all system resources (e.g., subjects, objects, and I/O devices) that are directly or indirectly accessible by subjects external to the TCB.

The LOCK TCB maintains a SIDEARM-resident data structure that is a partially ordered set (POSet) of all security levels known to the system. When a subject and/or object is created, it is assigned one of the levels from the POSet. Access is then computed using the level of the subject requesting access and the level of the object being accessed in the following manner:

- To read an object, the level of the subject must dominate the level of the object (the Simple Security Property).
- To write an object, the level of the subject must be dominated by the level of the object (the *-Property).

As used in the rules above, the term "dominate" means less than or equal to. The POSet is consulted to determine if one level dominates another.

2.2.2. Type Enforcement

Type enforcement is a mechanism that is unique to the LOCK TCB. Not required by the Orange Book, it is this mechanism that will allow the LOCK TCB to exceed the Orange Book Class A1 requirements. Type enforcement is based on two attributes:

- The domain of execution of a subject
- The type of the object a subject is attempting to access.

A domain is similar in concept to rings in ringed architecture machines. Unlike rings, though, there is no hierarchical relationship between domains. Moving from one domain to another does not necessarily imply an accumulation of increasing system privilege. Rather, each domain has a set of privileges different from other domains.

To represent the domains and the privileges allowed in them, the TCB maintains a SIDEARM-resident data structure called the Domain Table. It contains the following information:

- The UID of the domain
- The human-readable name of the domain
- A list of special privileges
- A list of domains that can be transitioned to.

The special permissions that are allowed in domains are the ability for a subject to take exception to the DAC and/or the Labeled Security Protection mechanisms of the system. Since it is the type enforcement mechanism that allows a subject to have these special privileges, a subject may never take exception to the type enforcement rules of the system.

When a subject is said to have the ability to transition to another domain, this means that it can create another subject in the domain that is allowed to be transitioned to.

The domain of execution is an attribute of a subject that remains constant throughout its lifetime. In other words, a subject can only execute in one domain.

All objects have a type associated with them. The concept of type is similar in nature to types in high level programming languages. The TCB restricts operations on objects of specific types based on the domain of execution of the subject attempting the access.

To represent types and the operations allowed on them, the TCB maintains a SIDEARM-resident data structure called the Type Table. It contains the following information:

- The UID of the type
- The human-readable type name
- Allowable object sizes (minimum and maximum)
- List of access vectors for the type in existing domains
- Default ACL.

The list of access vectors defines all operations allowed on objects of a specific type for each domain defined in the Domain Table.

When a subject requests access to (or attempts to create) an object, the TCB consults the Domain and Type Tables to determine if the access, based on the domain of execution and object type, is allowed.

Both the Domain Table and Type Table are initialized at sysgen time by the System Security Officer (SSO) and are inaccessible to user processes.

As mentioned earlier, type enforcement can be used to grant special privileges. For example, it may be necessary to implement an application that is allowed to downgrade files. The list of special privileges in the Domain Table is used to grant such privileges. The list of access vectors in the Type Table is used to restrict which object types can be read and written in the downgrade process.

Type enforcement is also useful for integrity reasons. For example, the system may grant subjects running in a system administration domain read and write access to objects of type "password file." Subjects running in the OS domain may be granted only read access to objects of type password file. With the Domain and Type Tables established in this fashion, the system will prevent unauthorized modification (integrity) of objects of the type password file.

The type enforcement mechanism can be used to support a variety of integrity models such as the Clark-Wilson [CLARK87] model, and as described in [BOEB85], the Biba [BIBA75] model.

2.3. Subjects

The basic execution (active) entity in LOCK is the subject. A subject is a process that executes in a particular security context. The security context comprises the level of the subject, the domain of execution, and the user on whose behalf the subject is executing. In many ways, a subject is like a UNIX process: it shares the processor with other subjects through timeslicing, it has access to a "file system" that other subjects also have access to, it can open and operate on "files," and it has limited capabilities for communicating with other subjects.

There are some notable differences between subjects and UNIX processes. There are no hierarchical parent/child relationships; each subject is independent of the subject that created it. For UNIX processes with effective user IDs of superuser, the entire system is accessible; there is no corresponding notion of superuser in the LOCK UNIX world. Under UNIX, multiple processes can be writing to the process control terminal simultaneously; LOCK allows only one subject to perform terminal I/O at a time.

All subjects have associated with them a Subject Translation Table (STT). The STT contains an entry for each object that the subject has opened (see Figure 5). In LOCK/ix objects are data files, text segments, data segments, stack objects, and kernel level data structures. The STT is similar in nature to the UNIX per-user open file table. Each entry in the STT identifies an object and the current access that the subject has to it. The STT is resident in the host processor's memory and provides the first level of address translation for the MMU.

Subjects within the LOCK system are characterized by the following:

- Each subject is uniquely identified within the system's security database (the GOT). A subject is uniquely identified by the UID the SIDEARM assigned to it when it was created.

- The subject manager within the TCB maintains an Active Subject Table. The Active Subject Table contains an entry for each active subject within the system. Subject scheduling and multiplexing is performed by the TCB's subject manager.

- A user subject may execute instructions if and only if the host processor is operating in user mode. (Only TCB subjects may operate when the host processor is in privileged mode.)

User subjects are created as a result of a TCB Create Subject request. They come into existence as the result of a user action, perform their function, and are terminated by a TCB Destroy Subject request at some later time. When a subject is destroyed, the subject ceases to exist within the system. All objects allocated by the subject (contained within its STT) are closed, and all resources (e.g., memory) previously allocated to the subject are released.

2.3.1. Relation to UNIX Virtual Machines/UNIX Processes

The differences between UNIX processes and LOCK subjects strongly influenced the way UNIX processes are represented in LOCK/ix. To cleanly support UNIX process functionality, each subject represents the equivalent of an abstract UNIX virtual machine.

The LOCK subject has capabilities not provided by UNIX processes. To support operating systems built on top of LOCK, as well as multitasking applications such as an Ada run-time environment, a subject has a periodic interrupt, similar to a timeslice interrupt, available to it. A "beginning timeslice" signal is sent to all subjects from the TCB when they begin to execute in a new timeslice. To take advantage of this feature, a subject must enable a signal handler, in much the same way as is done for UNIX signal handlers. If a subject does not wish to take advantage of this signal and does not define a handler for it, the signal is ignored and the subject is allowed to run without the knowledge of receipt of the signal.

Unlike the UNIX signal handling mechanism, the LOCK signal handling mechanism provides its context to the subject in the form of register, stack pointer, and program counter values when the signal occurred. However, there is no way to control the frequency of this signal. The frequency of occurrence of this signal is directly related to the system load.

The use of this feature allows a subject to perform its own process multiplexing. Each subject can run its own process (or task) multiplexing to provide multiprocessing support within the subject.

2.4. Objects

One of the most unusual features of LOCK (at least for those accustomed to UNIX) is that there is no notion of external files or a file system; instead, there are objects. Objects are containers for visible data that reside in the virtual memory and can be (physically) stored on disk, tape, or other media such as optical disk. Every object in the system has a size, security level, owner, physical location, and (potentially) an ACL associated with it.

Objects are a generalization of the segmented memory system used by Multics. The TCB Open Object operation maps the object into the virtual address space of the subject and returns a pointer to a memory address. Data with an open object can be accessed by referencing offsets into the object's memory range. I/O is performed on objects by modifying the contents of memory addresses in the open object. One object can be open by multiple subjects simultaneously, with the object mapped to different virtual addresses in each subject's address space.

2.4.1. Relation to UNIX Virtual Memory

All memory that a subject references, even the subject itself, consists of open objects. The code and data that a subject executes consists of several open objects. The virtual memory space of a subject is the union of open object virtual addresses. LOCK imposes a limit on the number of open objects a subject is

allowed, which is currently 256. The maximum size of an object is 16Mbytes.

Disk I/O is performed by LOCK without explicitly doing I/O. The MMU provides the mapping between memory references and modifications and physical I/O. If a piece of an open object that has been paged out to disk is referenced, the MMU brings the appropriate piece of the object into memory. To an application, the entire contents of an object appear to be in memory when an object is opened, and the contents disappear when the object is closed. Referencing a memory address that is not mapped to an open object generates a bus error. A bus error will be interpreted by the TCB as an attempt to violate the security policy of the system and cause the termination of the offending subject.

LOCK object operations are analogous to UNIX memory management functions in many ways. Opening an object is similar to allocating a region, in order to obtain memory for a process. Objects can expand and shrink, as can regions. Open objects are memory regions associated with each process.

2.4.2. Relation to UNIX File System and Files

Objects provide the foundation for building a LOCK file system that will appear to operate similar to UNIX. However, from a programming standpoint, object operations are quite different than file I/O operations.

The LOCK/ix kernel is responsible for providing the functional bridge between the LOCK TCB and UNIX applications. LOCK/ix will need to provide the functionality to support a UNIX file system on top of LOCK object operations.

File creation requires that an object be created and cataloged into the file system in the proper directory, with the inode table providing the linkage between physical storage and external appearance. Open and close operations logically perform the same function in both LOCK and UNIX, allowing or removing file access from the executing process, but the implementation methods differ. LOCK/ix will map the UNIX-style access operations into their LOCK counterparts. UNIX-style I/O operations will be mapped into open object references and updates. File deletion removes a reference to an object from the file system, and if there are no references remaining, the object will be deleted.

3. Process Management in LOCK/ix

The UNIX process management services provide process creation and deletion, program execution, and synchronization between related processes. The UNIX model of process creation, using the `fork()` operation, enforces parent-child relationships between processes and ensures that a child process is initially created to be an exact copy of its parent. The UNIX model of program execution, using the `exec()` operation, provides for the inheritance by the new program of part of the environment of the process that executes the program.

In contrast to UNIX System V, in which all user processes are managed and coordinated by a single kernel entity, the LOCK/ix implementation encapsulates the management of processes for each LOCK/ix login session within a single LOCK subject (see Figure 6). Each LOCK/ix subject contains a (virtual) instance of the UNIX kernel that manages only the user processes associated with its login session. The LOCK/ix kernel is in reality a shared text segment that is used by all LOCK/ix subjects. However, at any point in time, the kernel only knows about the single LOCK/ix subject that it is currently servicing.

This approach to process management provides a `fork()` and `exec()` implementation that is compatible with UNIX System V from the viewpoint of an executing user program. Since most explicit communication between user processes provided by UNIX requires the processes to be directly related or within the same process group, such functions can be provided locally within a LOCK/ix subject.

Process management is affected by the TCB security mechanisms only with regard to access to executable files by the `exec()` function. The LOCK/ix subject performing an `exec()` must have execute access to the executable file according to both the MAC and DAC policies being enforced by the TCB.

The set-user-ID and set-group-ID modes for file execution are supported as in System

V, but with the same mandatory access restrictions imposed by the TCB. The set-user-ID execution mode affects only the UNIX-specific access permissions implied by UNIX user-IDs and does not cause a change in the LOCK user UID of the subject.

There are two problems that arise with set-user-ID and set-group-ID applications. The problems exist because UNIX maintains both an effective-user-ID and real-user-ID for each process. These user-IDs are not always the same. In contrast, the LOCK TCB maintains only a user-ID for each active subject. A LOCK subject user-ID and a process real-user-ID will be the same (the person who is actually running an application).

The first problem arises from the fact that file system objects have ACLs associated with them. The LOCK TCB evaluates ACLs based on the real user's identity. What this means is that there must be an ACL entry for the real user (who `exec()`ed the program) with the desired access rights for all files that the application will access. UNIX may allow a process to inherit the access rights of the owner of a set-user-ID application, but the TCB will not.

The second problem arises from the fact that when an object is created, the TCB assigns as the owner attribute the name of the real user who created the file. Additionally, the ACL for the file is initialized with one entry for the owner. This will result in a conflict between who UNIX thinks is the owner and who the TCB thinks is the owner.

When an object is initially created by the TCB, it is assigned a default ACL. This default ACL will contain only one entry that gives the owner read and write permission. This will have the effect (from a UNIX point of view), of giving the wrong person object access. The ACL entry will specify the name of the individual who ran the set-user-ID application, rather than the name of the individual who owns the application.

We are currently investigating several alternative approaches to solving this problem. Since the UNIX user community appears to have a strong desire to continue to run set-user-ID applications, our current design will most likely be criticized as providing unacceptable support for set-user-id applications.

There are some within the computer security community who argue that a capability such as setuid should not be provided by A1 systems. This is a debate that is likely to continue for many years to come. Our feeling is that, in general, existing UNIX protection mechanisms and in particular the setuid capability are really integrity mechanisms. If a method can be devised to support a fully functional setuid capability in a secure manner, we see no disadvantage to doing so.

3.1. Memory Management

UNIX kernel memory management functions provide user processes with an expandable data area that can be used for dynamic heap allocation. The heap allocation algorithms are supplied by the run-time library and provide a generalized memory block allocation scheme to user programs. They call on the kernel to expand the user process's data area as needed to increase the pool of memory that is available for allocation.

Although not explicitly visible at the user program interface, UNIX memory management also normally enforces protection of user process address space from access by other processes. The kernel's memory space is also protected from access by any user process. These attributes are dependent on the MMU hardware characteristics of the target machine. Since a LOCK/ix subject has no explicit control over the TCB's MMU, such protection cannot be provided by LOCK/ix.

The LOCK/ix kernel manages memory via calls to the TCB storage manager. The physical allocation of memory is replaced by requests to create, delete, open, close, and expand the LOCK objects that compose the address space of user processes. Each user process within the subject is assigned text, data and stack objects by the kernel which are open to the subject as long as the process is executable. When a process terminates, these objects are closed and deleted by the kernel.

Expansion of the data area of a user process is implemented by calling the TCB storage manager to expand the data object for the process. This TCB function automatically handles physical relocation of the object if needed and zeros the newly allocated space for LOCK/ix.

Expansion of the stack of a user process is implemented by a special LOCK/ix system call, which is used to request that the stack of the calling process be extended. This requires that the compiler generate a stack overflow check as part of every C function's entry preamble code. Although relatively inefficient, this method of automatic stack growth is used on standard UNIX machines that have no hardware support for stack overflow detection in the MMU. Memory management is not affected by the TCB security mechanisms, since all functions are local to the LOCK/ix subject.

4. The LOCK/ix File System Design

During the course of the study, three options were identified for supporting a UNIX file system and accessing files at lower levels. It became clear that either a significant amount of trusted code would be required to support the UNIX file system as currently implemented, that a totally separate file system was required for each different level, or that the file system structure would have to change, but still appear to users and applications like the UNIX file system.

The first option was discarded as being unacceptable from a design and implementation point of view. Separate file systems would require minimum changes from the current UNIX implementation, but would create problems in usability and system maintainability since lower level files would be inaccessible. A redesign of the file system to make it appear as much like the UNIX file system as possible, yet fit into the MLS framework, became the most viable alternative.

A separate file system at each level is supported. A separate inode table for each level will be stored in an object at the level of the file system. Directories map inodes to files as in existing UNIX implementations. The key difference in LOCK/ix is that directories with the same path name will exist at the multiple levels that are accessible to a subject. The directories are logically overlaid to produce a combined virtual directory. To a user process, there will always appear to be only one file system, as with UNIX. The level at which the user process is executing will determine which files are visible in its view of the file system.

A directory can only contain files that are at the same level as the directory. However, if the same directory path exists at several levels, each containing files, applications are given the illusion that the directory contains files at multiple levels. Only the directory paths need to be duplicated, not the files.

Figure 7 illustrates a simple two-level file system. Level 0 is lower in security level than Level 1. A LOCK/ix process running at Level 0 would see the files `cat` and `ls` in the `/bin` directory. A Level 1 process would see those two files, and, in addition, the "magic" file. LOCK/ix performs the logical combination of the file systems at multiple levels. The files at Level 0 that the Level 1 process is aware of will never be writable. The view of the file system presented to a Level 1 process will contain the Level 1 file system overlaid on top of the Level 0 file system, as if each were a transparent figure. A Level 0 process will have no way of determining the existence of anything in Level 1 (or beyond), because LOCK will deny access to any Level 1 (or beyond) file system objects, including the inode table and root directory.

The concept of virtual directories could also be applicable to a conventional, unsecure networked UNIX environment. If file systems resided on multiple machines, some with the same directory path names, virtual directories could be created. The issue of which network machine a file resided on would no longer be significant.

4.1. Path Inheritance

In order to support virtual directories, a method is needed whereby files can be created at the current level if the directory path exists at a lower level. It would be incompatible with UNIX to allow the creation of a directory that already exists. Therefore, LOCK/ix automatically creates ("inherits") directory paths that exist at a lower level, but not the current level, when creating a new file in a directory.

If a directory does not exist at an observable level, attempting to create a file in the directory will fail, as with UNIX. If the directory does exist at the current level, no further action needs to be taken in terms of path creation. Only when components of the path do

not exist at the current level does LOCK/ix need to automatically create them. The creation of path components at the current level is handled in a completely transparent manner. Other than noting a slight delay, a user or process will not be aware that it is occurring.

Path inheritance creates only the directory paths that are needed at a given level. A good analog is a virtual memory system. The working set is at first very small, containing only the top-level path components. As files are created, as with pages not in memory, a fault occurs and the appropriate pages are brought in from disk, or in this case, path components are created. Eventually, a stable working set is established that handles most references, for either virtual memory or the LOCK/ix file system. The major difference is that the directory paths created are permanent, real directory entries that exist in the file system. There is no way to determine afterwards whether a directory path was inherited or explicitly created.

If directories are created with names that (unknowingly and unintentionally) match those at a higher level, the higher level virtual directory view will contain all the files. The lower level view will only consist of those at the lower level.

An open issue currently is how to resolve name collisions. If identically named files exist at multiple levels in a directory, the higher level processes will need a way to determine which version gets accessed. An extension to the `namei` routine, which performs name to inode mapping, is planned for the future. The main changes in logic to support file systems at different levels that are combined to appear as one composite file system have been in the `namei` module.

4.2. File System Examples

A few simple examples will help illustrate how the file system appears and operates. The examples are built on the file system shown in Figure 7.

Figure 8 shows how the file system would appear if an application running at Level 1 created a file named `z` in the directory `/usr/user3`. Before creating the `z` file, the LOCK/ix subject was required to create the full directory path at Level 1 so that the file would end up at the correct level in the file system. In this case, only the `user3` path component was created, because the `/usr` component already existed. The inode numbers for each level can be, and typically will be, different for each level of the file system. The Level 1 application creating the file `z` is unaware that the path is being inherited. There will still appear to be one `/usr/user3` directory to both Level 0 and Level 1 applications, and they will appear to be the same, although the Level 1 application can potentially access additional files. If a file named `/usr/user3/z` already existed at Level 0, the Level 1 application would not be able to create the file, since one would already exist as a read only file. A Level 0 application can create a file named `/usr/user3/z`, since the Level 1 version would not be visible. The Level 1 application would still access the Level 1 copy. If the Level 1 application deleted its copy of the `z` file, the Level 0 file `z` would remain in the file system.

Figure 9 illustrates the file system after making the directory `/usr/user2/da` at Level 0, then creating the file `z` in that directory. There was no directory appearing to the Level 0 processes by the Level 0 name before the directory was created, although one existed at a higher level. There will still appear to be one `/usr/user2/da` directory to the Level 1 process, but there is now a read-only file named `z` in the directory.

Figure 10 shows the results of the deletion of the directory `/usr/user3` by a Level 0 process. To the Level 0 process, the directory appeared to be empty, thus it was permissible to unlink the directory. Without path inheritance, trusted code would be required to make sure that the file `z` created in the previous example was properly connected in the directory structure when the Level 0 path was deleted. With path inheritance, the file was created on a valid directory path to begin with, and the lower level process did not need to perform any special processing to account for files created at higher levels.

4.3. Integrity Considerations

The LOCK/ix kernel runs in the same address space as user processes. LOCK does not allow a single subject to execute in multiple domains. A process within a given LOCK/ix subject could gain access to any kernel file structures at its level and modify them. For this reason the file system update operations are removed from the LOCK/ix kernel and implemented as separate file system server subjects per level. Independent of a file system server, LOCK will provide protection for files from an aberrant program if the program does not have update (or write) access to the file.

The file system inode table is broken out into two parts: non critical components, such as time modified, and critical components, such as object UIDs. The non critical components consist of fields that can be updated by the LOCK/ix kernel and that would not cause any problems if they were incorrect. The critical component will be in a different object type, which can be accessed by the LOCK/ix kernel, but not modified by it. The file system server subject will run in a domain that is different than that of the LOCK/ix kernel and user processes, and will have update access to the critical inode table.

The file system server will only perform file system update operations; it will not run processes. No malicious programs that could cause unexpected and undesirable consequences will run. This particular instance shows how Type Enforcement can be used to support an integrity policy that will achieve the desired end result.

5. Conclusion

The results of our study are encouraging. We were surprised to find how much of the UNIX kernel code can be retained unmodified. However, our conceptual design has not been put to the test; it is only a paper design. We plan to begin implementation of LOCK/ix in April 1988, with the first version of the system running by April 1989.

The LOCK/ix system design is really in its infancy. There are some issues that we did not address in the study in much detail, such as performance. Our future plans for enhancements include refining the file system design, solving the setuid problem, and extending the standard interface to incorporate security-relevant functionality provided by the underlying TCB. The latter will provide the capabilities necessary to develop Multi-Level applications.

Acknowledgments

We would like to thank Bob Hartman, Jim Papke, Glen Swonk (ComputerBase), and Mike Carty (Sendona, formerly I.C.E.S. Ltd.) who also participated in the study.

References

[BIBA75] K. J. Biba, "Integrity Considerations for Secure Computer Systems," The MITRE Corporation, Bedford MA, MTR-3153, 30 June 1975.

[BOEB85] W. E. Boebert, "A Practical Alternative to Hierarchical Integrity Policies," Proceedings, 8th National Computer Security Conference, 1985.

[CLARK87] D. D. Clark and D. R. Wilson, "A Comparison of Commercial and Military Security Policies," Proceedings, IEEE Symposium on Security and Privacy, 1987.

[HONE83] Honeywell, SCTC, A-Specification, Contract MDA 904-82-C-0444, April 1983.

[HONE86] Honeywell, SCTC, B-Specification, Contract MDA 904-84-C-6011, March 1986.

[SVID86] AT&T, "System V Interface Definition," 1986.

[SYAD87] O. Sami Saydjari, et al, "LOCKing Computer Securely," Proceedings, 10th National Computer Security Conference, 1987.

[TCSEC85] "Department of Defense Trusted Computer System Evaluation Criteria," DoD 5200.28.STD, December 1985.

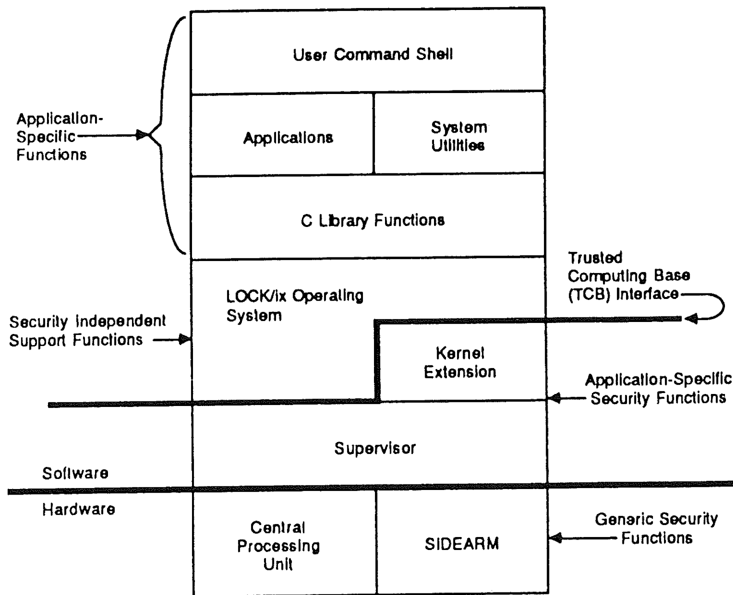
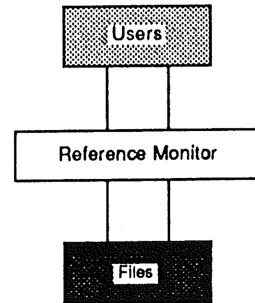


Figure 1

Reference Monitor Concept

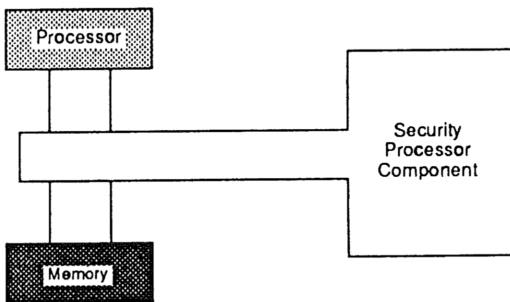


A Reference Monitor Must Be

1. Always invoked
2. Verfied correct
3. Tamperproof

Figure 2

Reference Monitor in LOCK

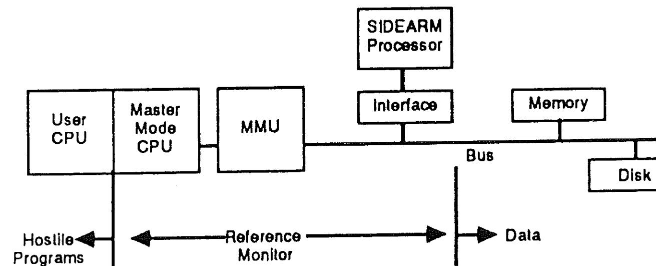


Hardware Advantages

1. Always invoked - No way to bypass
2. Verified correct - Simpler; machine indpendent
3. Tamperproof - No way to attack security processor component

Figure 3

Reference Monitor in LOCK



Hardware Advantages

1. High Assurance
2. Reasonable Performance
3. Application Portability
4. Functionality

Figure 4

Typical LOCK/ix Subject STT

LOCK/ix Kernel Text Segment
LOCK/ix Kernel Data Segment
LOCK/ix Kernel Stack Object
sh Text Segment
sh Data Segment
sh Stack Segment
• • •
emacsText Segment
emacs Data Segment
emacs Stack Segment
File 1
File 2
• • •

Figure 5

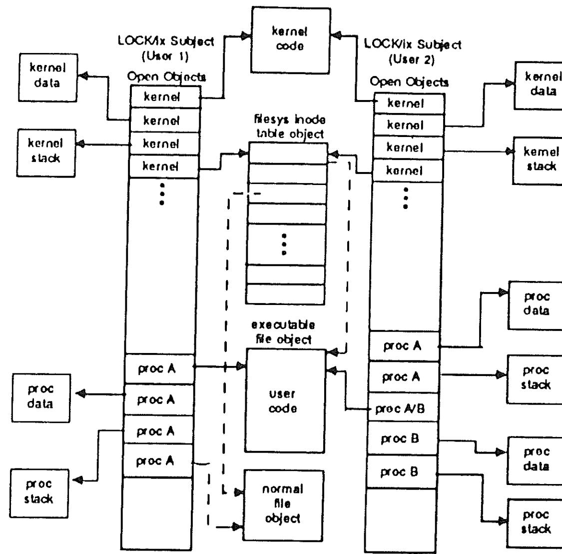
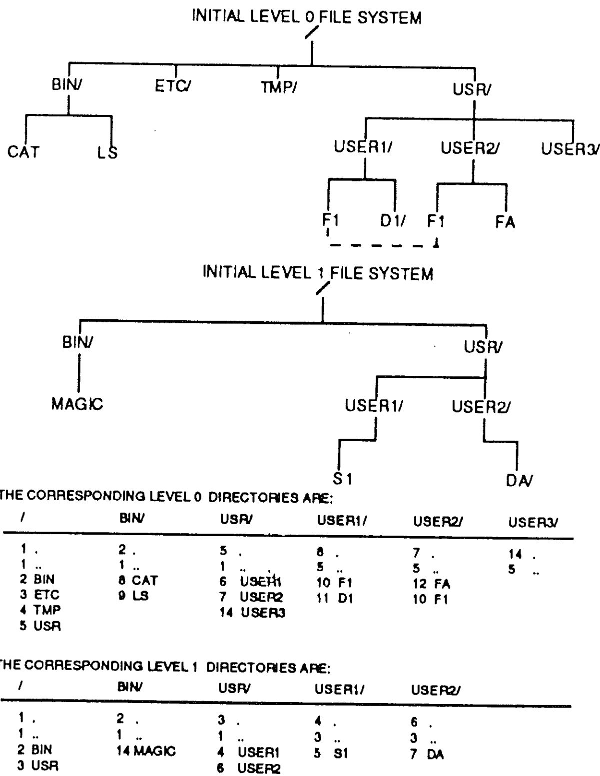


Figure 6



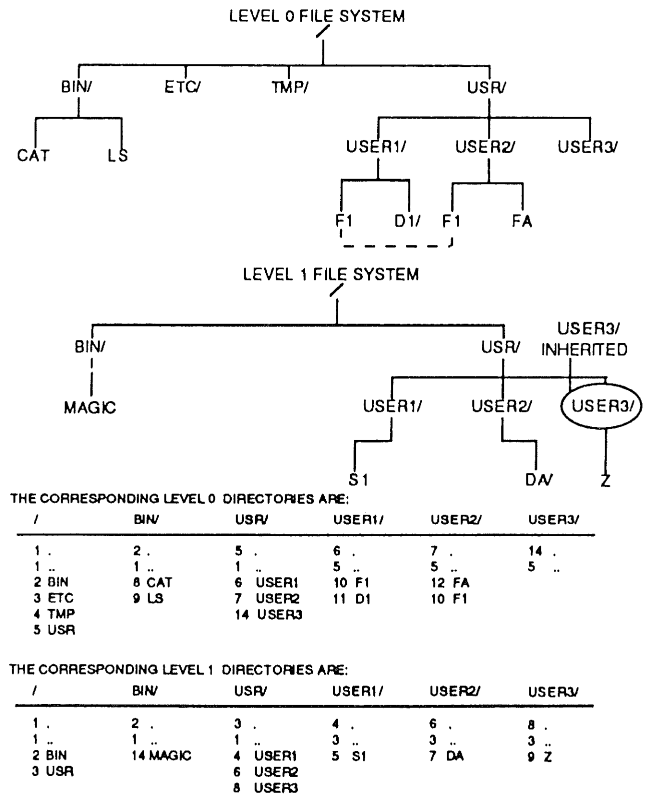
THE CORRESPONDING LEVEL 0 DIRECTORIES ARE:

/	BIN/	USR/	USER1/	USER2/	USER3/
1 .	2 .	5 .	8 .	7 .	14 .
1 ..	1 ..	1 ..	5 ..	5 ..	5 ..
2 BIN	8 CAT	6 USER1	10 F1	12 FA	
3 ETC	9 LS	7 USER2	11 D1	10 F1	
4 TMP		14 USER3			
5 USR					

THE CORRESPONDING LEVEL 1 DIRECTORIES ARE:

/	BIN/	USR/	USER1/	USER2/
1 .	2 .	3 .	4 .	6 .
1 ..	1 ..	1 ..	3 ..	3 ..
2 BIN	14 MAGIC	4 USER1	5 S1	7 DA
3 USR		6 USER2		

Figure 7



THE CORRESPONDING LEVEL 0 DIRECTORIES ARE:

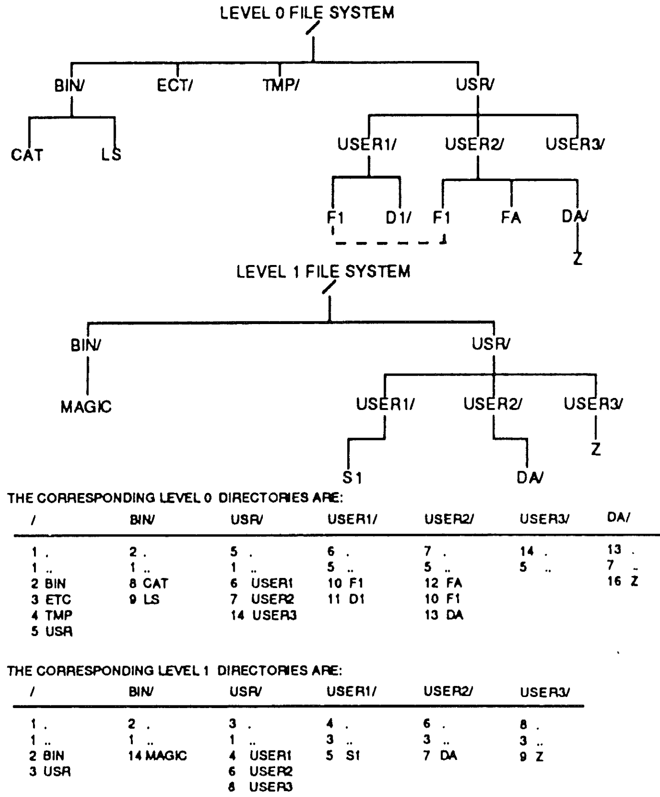
/	BIN/	USR/	USER1/	USER2/	USER3/
1 .	2 .	5 .	6 .	7 .	14 .
1 ..	1 ..	1 ..	5 ..	5 ..	5 ..
2 BIN	8 CAT	6 USER1	10 F1	12 FA	
3 ETC	9 LS	7 USER2	11 D1	10 F1	
4 TMP		14 USER3			
5 USR					

THE CORRESPONDING LEVEL 1 DIRECTORIES ARE:

/	BIN/	USR/	USER1/	USER2/	USER3/
1 .	2 .	3 .	4 .	6 .	8 .
1 ..	1 ..	1 ..	3 ..	3 ..	3 ..
2 BIN	14 MAGIC	4 USER1	5 S1	7 DA	9 Z
3 USR		6 USER2			
		8 USER3			

Figure 8

File System After Direrctory /usr/user2/da and
File /usr/user2/da/z Created by Level 0 Process



THE CORRESPONDING LEVEL 0 DIRECTORIES ARE:

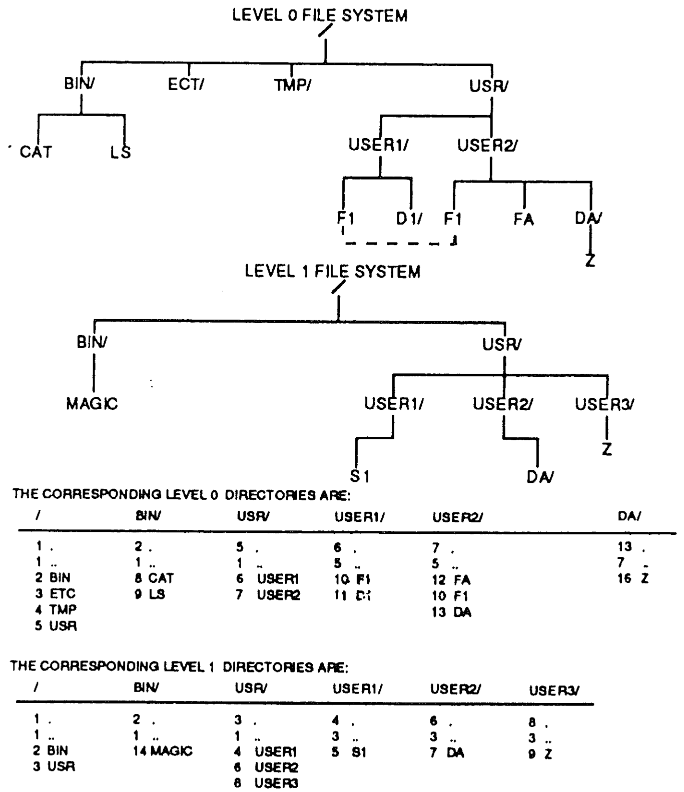
/	BIN/	USR/	USER1/	USER2/	USER3/	DA/
1 .	2 .	5 .	6 .	7 .	14 .	13 .
1 ..	1 ..	1 ..	5 ..	5 ..	5 ..	7 ..
2 BIN	8 CAT	6 USER1	10 F1	12 FA		16 Z
3 ETC	9 LS	7 USER2	11 D1	10 F1		
4 TMP		14 USER3		13 DA		
5 USR						

THE CORRESPONDING LEVEL 1 DIRECTORIES ARE:

/	BIN/	USR/	USER1/	USER2/	USER3/
1 .	2 .	3 .	4 .	6 .	8 .
1 ..	1 ..	1 ..	3 ..	3 ..	3 ..
2 BIN	14 MAGIC	4 USER1	5 S1	7 DA	9 Z
3 USR		6 USER2			
		8 USER3			

Figure 9

File System After Direrctory /usr/user3
Deleted by Level 0 Process



THE CORRESPONDING LEVEL 0 DIRECTORIES ARE:

/	BIN/	USR/	USER1/	USER2/	DA/
1 .	2 .	5 .	6 .	7 .	13 .
1 ..	1 ..	1 ..	5 ..	5 ..	7 ..
2 BIN	8 CAT	6 USER1	10 F1	12 FA	16 Z
3 ETC	9 LS	7 USER2	11 D1	10 F1	
4 TMP				13 DA	
5 USR					

THE CORRESPONDING LEVEL 1 DIRECTORIES ARE:

/	BIN/	USR/	USER1/	USER2/	USER3/
1 .	2 .	3 .	4 .	6 .	8 .
1 ..	1 ..	1 ..	3 ..	3 ..	3 ..
2 BIN	14 MAGIC	4 USER1	5 S1	7 DA	9 Z
3 USR		6 USER2			
		8 USER3			

Figure 10

An Update on UNIX Standards Activities

Shane P. McCarron, NAPS Inc.

April 17, 1988

Overview

This is the second in a series of reports on the UNIX standards community. In this article I will give you a summary of what happened at the March meeting of the POSIX committees. I will also explain what happened during the IEEE P1003.1 balloting, and why there is going to be another round of review and comment during May. In addition I will discuss what is going on with the National Bureau of Standards (NBS) Federal Information Processing Standards (FIPS), and how this will affect both implementors and programmers in the short and long term. Those of you who saw the first article in this series will remember that the title was "An update on UNIX and C Standards Activities." That changed this time because the ANSI X3J11 meeting isn't until mid-April, and there hasn't been too much going on between meetings (other than a public review). Next quarter I will return to the C arena as well.

P1003.1 Final Ballot?

Those of you who saw the first issue of this column may remember that I reported on the status of the P1003.1 balloting. At that time I stated that the standards would be fully ratified in March... Well, I was wrong. Although the IEEE review board gave the standard conditional approval, it did not pass in its first round of balloting, nor did it pass in the first recirculation for review and comment. Needless to say, I was a little surprised, but there were many factors that figured into the problem.

In the interest of clearing the air, below you will find a chronological account of the balloting procedure. I have also outlined the IEEE requirements for balloting, and how P1003.1 worked within these constraints. Even though you may finish reading the summary with an uneasy feeling about the process, please keep in mind that until recently there have been no large IEEE standards. The procedures were designed for brief documents describing the characteristics of three-phase power, not for 400 page documents specifying all the characteristics of an operating system.

On November 15th the Standard went out to the balloting group. The balloting group consists of IEEE or IEEE Computer Society members who have indicated an interest in voting on this standard. When balloters vote no, they must return a document which states their specific objections, and what can be done to resolve them. Although specific wording is not required, it is encouraged.

On December 15th (actually, a little after) balloting on the standard closed. The official IEEE length of a balloting period is 30 days, or until 75% of the balloting group members have returned a ballot, whichever is later. When 75% of the ballots had been returned, the standard did not have the necessary percentage of yes votes (75%) for approval. At this point the standard and the ballots were turned over to the Technical Reviewers for resolution.

On January 15th (or so) the committee chair started to assemble the ballot resolution documents for recirculation to the balloting group. The resulting document was a summary of all the changes made to the standard to resolve balloting objections or comments. In all there were 140 pages of changes, and (unfortunately) they were poorly organized and formatted. In my own defense (as a Technical Reviewer) I can only say that the process was rushed, and I procrastinated a little. Also, communication among the Technical Reviewers was lacking, and the guidelines for reviewing and acting on ballots were unclear. This is all kind of tragic, but it was certainly an educational experience.

On February 5th the resolution document was resubmitted to the balloting group for a 10 day review period that was to start on the 15th. Unfortunately the mail was held up until the 15th (or in some cases the 17th) and many balloting group members did not receive the recirculation document until the 20th or

later, for return to the IEEE Standards office by the 25th. Worse yet, the IEEE balloting procedures state that if the technical reviewers have resolved all objections in a ballot, that ballot automatically becomes a yes. The balloter must specifically indicate that his/her ballot is still negative. This was not made very clear to the balloting group, and many people did not resubmit a ballot.

Fortunately many people did complain about the short review period and the problems with the recirculation document. Eventually it was discovered that the 10 day period that IEEE stipulates for reviews is a minimum, not a maximum. There was a lot of finger pointing and complaining on all sides, and in the end it was decided that even though the standard had the necessary 75% approval, there would be another recirculation.

During the week of March 7th, the IEEE Standards Board met. In spite of all the problems with the standard, and all of the letters of protest that they received (including one from each of the Institutional Representatives, if I am not mistaken), the board conditionally approved the standard. [You're not mistaken: the Institutional Representatives of USENIX, /usr/group and X/OPEN all sent letters of protest to the Standards Board; I also spoke to the Standards Activities Group directly about the time limit problem. -jsq] This conditional approval is an unprecedented event (as far as I can tell) and means that the standard can become fully ratified before the next meeting of the standards board once the second recirculation has been completed and it has sufficient positive ballots. There was a lot of screaming about this as well.

During the week of March 14th the POSIX committees met in Washington D.C. Throughout the meetings the co-chair of P1003.1 met with each of the Technical Reviewers and very carefully went through their sections of the document, making sure that all objections and comments had been considered, processed, and responded to. This was an incredibly time consuming and painful process, but I believe that it resulted in a much better standard. During the last few weeks the Technical Reviewers have continued to work closely with the co-chair to get the second recirculation document put together. It should be completed and sent to the Technical Reviewers (as a safety check) in mid-April. Once the Reviewers think that it is clean enough, it will be sent out to the balloting group for a second review and comment period.

The second recirculation will be handled quite a bit differently than the first. All members of the balloting group will receive a new copy of the standard (Draft 12.3) that will have change bars only in those places where changes have been made as a result of balloting objections or comments. In addition, each balloter will receive a document detailing all of the unresolved objections, their nature, and why they were not resolved. The balloting group will have a longer period to respond to this document (> 10 days), but they shouldn't need much more time, as most of the changes in the document were already detailed in the first recirculation document (although they were not made in context – that is to say they were not in a new draft, but rather listed as changes to draft 12). At the end of this recirculation and balloting period it is believed by most members of the committee that the standard will be complete.

The time frame for all of this is late April/early May.

I apologize for the length of this summary, but I think it is important that everyone know just what happened. Of course, this is just one man's perspective, but I think that it is a fair one. I believe that the completed standard will be one which was carefully considered and designed, even if it won't make everyone happy.

NBS POSIX FIPS

As I reported last quarter, the National Bureau of Standards has specified a Federal Information Processing Standard for POSIX. This FIPS has now been called an Interim FIPS, and is based on Draft 12 of the POSIX standard (the draft that went to the balloting group). This is unfortunate, since the post balloting draft is significantly different in a number of areas. Also, the NBS has made some changes in their requirements for the FIPS since I last reported them. As of this writing the POSIX Interim FIPS for the System Services Interface is not official. It is going

through the government signature maze within the Department of Commerce, and is expected to emerge sometime in April.

This Interim FIPS will remain the standard until the P1003.1 standard is completed. Sometime after that the NBS will put together a final FIPS based on .1. Unfortunately, this may not be for several months after .1 is completed. In the meantime government agencies will be generating Requests for Procurement (RFPs) which stipulate the Interim FIPS.

What this means for systems implementors is not entirely clear. The government will be requiring (at least for a little while) a standard that is in many ways incompatible with the final P1003.1 document. Obviously implementors have two options: 1) put together POSIX conforming systems and wait until the final FIPS is complete before selling any systems, or 2) put together a FIPS conforming system and be able to start selling immediately. Fortunately implementors have an out here – many of them have release cycles lasting anywhere from 6 to 18 months. By the time there is a POSIX standard and they get their implementation ready to be released, the FIPS will have changed to reflect the final standard... maybe.

What it means to application developers is a little more obvious. Software that is in development today is probably too far along to consider making it POSIX conformant – or worse yet, ANSI C conformant. Software that is not yet in programming is going to take quite a while to get to market, so it can be made POSIX conformant without having to worry about the Interim FIPS.

In addition to this first FIPS, the NBS has stated that it is going to be releasing several more Interim FIPS based on some of the other POSIX work in progress, as well as the work of other groups (like AT&T and the SVID). During the POSIX meetings in Washington, Roger Martin from the NBS (and also chair of P1003.3 – Testing and Verification) made presentations to the various committees, explaining what the NBS intends to do in the next year with Interim FIPS.

In May or June an Interim FIPS for the Shell and Tools interface (POSIX P1003.2) will be proposed. It will be based on Draft 6 of the .2 document, and will contain (at least) the command set from that document. It may also contain text from that document, or in cases where the text is felt to be immature, will contain text from the SVID or some other source. This Interim FIPS will be based on Draft 6 until the final standard is completed sometime in later 1989.

In addition, the NBS will be releasing several other FIPS. These will be in the areas of Terminal Interface Extensions, System Administration, and Advanced Utilities. These are all terms from the SVID, and relate to just the things that you think they do. The Advanced Utilities FIPS may be rolled into the P1003.2 FIPS, since .2 encompasses most of those items that they wanted in there. The others will be based directly on the SVID (as far as I know). These are all to be in place by the end of 1988. This is an ambitious schedule, even for NBS. However if they meet it, it will mean that by the end of this year the government will have standards on most aspects of the UNIX operation system, and system implementors and application developers will have to conform.

IEEE P1003 Activities

As I mentioned above, the POSIX committees met in Washington D.C. in March. For the first time, all 7 of the committees met. As you can imagine, it was pretty difficult to catch all of what went on, but here are the highlights.

P1003.0 – POSIX Guide Project

This group met for the first time in Washington. Although they didn't get a lot of tangible work done, they did establish what their goals were, as well as starting to put together a timetable for production of their guide document. I don't have the details of this yet, but I will next quarter.

P1003.1 – System Services Interface

This group met to decide what we are going to be working on in the future. We have a few items that must be handled by the .1 group, and some that could be. Currently there are three projects being worked on by members of the committee:

Language Independent Description The ISO POSIX Working Group has requested that a language independent version of the .1 standard be produced as soon as possible after completion of the standard. Language bindings (like the current descriptions that are in the standard and the work being done by the .5 group) would be placed in supplements to the main standard, or in chapters within the standard itself.

Improved Archive Format Although the ISO community agrees that cpio and ustar are fine for the first cut of the standard, they have requested that .1 work on a more robust archive format that doesn't have the technical drawbacks of either, as well as one that takes into account the security features needed for trusted systems.

Terminal Interface Extensions Yes – we mean *curses/Terminfo*. Well, not really, but something very much like that. It will have to be something that resembles current practice (I imagine), but it could be improved in little ways. There was a lot of sentiment in the group for throwing out all of the *Terminfo* stuff and starting from scratch, but I don't think it will happen. We will probably get some proposals that are wildly different from existing practice, but it is outside the group's charter to totally supplant existing practice.

P1003.2 – Shell and Tools Interface

The .2 Group got a lot of work done in Washington. They went in with a 400 page draft 5, and by end of May a 450+ page draft 6 should be completed. This draft 6 will be used as the basis of the interim FIPS that the NBS will be using for their Interim FIPS on POSIX (see above).

The most significant developments in .2 were:

Source Code Control The committee felt that source code control was outside the scope of the standard, and it was removed (it had been added at the last meeting). A number of people still feel that some form of source code control should be in there, so the committee left a place in the document where it could be put back in later. The real danger here is that the rcs people and the sccs people will get into a religious war similar to the one that erupted between the tar and cpio factions in the .1 group.

Basic Shell Changes There were many features of the Bourne shell that had been included in .2 for historic reasons. At this meeting the shell subcommittee agreed to remove some of those anachronisms. This will make way for (possibly) more enhancements to the basic shell mechanism in the future (e.g., substring manipulation).

Software Installation Two drafts past there was a very complex system in the standard that allowed software installation in a portable way. This was removed in the December meeting, and replaced at the March meeting by a very simple interface that should be acceptable to everyone. Although the details are not all clear, it looks like this will consist of an implementation defined command that will read the first file off of a POSIX conforming archive (tape) and execute it. Anyway, something about that difficult.

Electronic Mail Interface mailx was added in Draft 5 as a proposed way to portably transmit mail. Some committee members felt that the way in which it was described was too restrictive, while others felt that it was too liberal. In a compromise move, another interface was defined that allows very simple mail transmission in a portable manner. It also has a name that doesn't conflict with existing utilities.

P1003.3 – Testing and Verification

At the March meeting the chair announced that they were on target for completing the assertion lists for P1003.1, and that the .3 standard for .1 would be ready to ballot just as soon as the .1 standard was ratified. He also stated pretty clearly that P1003.3 didn't want to work as hard when generating verification standards for the other POSIX committees. He asked that in the future the standards be written in a way that makes it easier to develop assertion lists. The .3 committee will be working closely with the .2 effort (which is a little too far along to fix now), but the other committees will be changing their documents to reflect what assertion tests can be made about each function or command being defined. This should make it

easier to produce verification documents for those standards.

P1003.4 – Real Time

This committee made a lot of progress in the March meeting. However, they have a long road ahead of them, and I don't know that anything earth shattering happened – certainly nothing that I heard about. However, they have stated a target of 1990 for completion, and at this point it is a little early to draw any sort of conclusions.

P1003.5 – Ada Binding for the System Services Interface

The Ada™ group is still a very young committee, but they are moving right along. At the very least they are generating a lot of paper, but it has some excellent stuff on it. Although they haven't been a working group long, I expect to see a draft from them in the next six months, and a standard being balloted in a year. Although this may seem like a long time, it is really short work for a standards committee. Unfortunately, their work is very dependent on .1 getting a language independent description of the System Services Interface put together as quickly as possible. They have already looked into ways of describing POSIX independent of any language, and they will be helping .1 get this firmed up.

P1003.6 – Security

This was the first meeting of .6 as a real IEEE committee. They defined their scope and objectives, set a tentative production schedule, and defined the format of their document. As a /usr/group technical committee they produced a number of white papers, and I expect to see drafts coming out of the group based on those papers shortly. The only snag here is that the transition from a /usr/group technical committee to an IEEE working group wasn't as smooth as others have been. To help alleviate some of the tension this caused, the next .6 meeting will be held in conjunction with USENIX in San Francisco in June, instead of with the POSIX committees in July. After that they will follow the regular POSIX meeting schedule.

Local User Groups

The USENIX Association will support local user groups by doing an initial mailing to assist the formation of a new group and publishing information on local groups in *;login:*. At least one member of the group must be a current member of the Association.

CA - Fresno: the Central California UNIX Users Group consists of a *uucp*-based electronic mailing list to which members may post questions or information. For connection information:

Educational and governmental institutions:

Brent Auernheimer (209) 294-4373
CSNET: brent@CSUFresno.edu
uucp: csufres!brent

Commercial institutions or individuals:

Gordon Crumal (209) 435-6062
uucp: csufres!tower!gordon

CA - Los Angeles: the Los Angeles UNIX Group meets on the 3rd Thursday of each month in Redondo Beach.

Drew Bullard (213) 535-1980
{ucbvax,ihnp4}!trwrb!bullard

Marc Ries (213) 535-1980
{decvax,sdcrdcf}!trwrb!ries

CO - Boulder: meets monthly at different sites.

Front Range UNIX Users Group
USENIX Association Exhibit Office
5398 Manhattan Circle
Boulder, CO 80303

John L. Donnelly (303) 499-2600
{boulder,usenix}!johnd

FL - Coral Springs:

S. Shaw McQuinn (305) 344-8686
8557 W. Sample Road
Coral Springs, FL 33065

FL - Melbourne: the Space Coast UNIX Users Group meets at 8pm on the 3rd Wednesday of each month at the Florida Institute of Technology.

Alex Stover (305) 724-3962
codas!lola!als

Bill Davis (305) 242-4449
bill@ccd.harris.com

FL - Orlando: the Central Florida UNIX Users Group meets the 3rd Thursday of each month.

Mike Geldner (305) 862-0949
codas!sunfla!mike

Ben Goldfarb (305) 275-2790
goldfarb@hcx9.ucf.edu

Mikel Manitius (305) 869-2462
{codas,attmail}!mikel

GA - Atlanta: meets on the 1st Monday of each month in White Hall, Emory University.

Atlanta UNIX Users Group
P.O. Box 12241
Atlanta, GA 30355-2241

Marc Merlin (404) 442-4772
Mark Landry (404) 365-8108

MI - Detroit/Ann Arbor: meets the 2nd Thursday of each month in Ann Arbor.

William Bulley (313) 995-6211
web@applga.uucp

Rich McGill (313) 971-5950
rich@oxtrap.uucp

Steve Simmons (313) 426-8981
scs@lokkur.uucp

MI - Detroit/Ann Arbor: dinner meetings the 1st Wednesday of each month.

Linda Mason (313) 855-4220
michigan!/usr/group
P.O. Box 189602
Farmington Hills, MI 48018-9602

MN - Minnetonka: meets the 1st Wednesday of each month.

UNIX Users of Minnesota
4732 Spring Circle
Minnetonka, MN 55343

Mark Colburn (612) 935-2688
mark@ems.mn.org
ihnp4!meccts!ems!mark

MO – St. Louis:

St. Louis UNIX Users Group
Plus Five Computer Services
765 Westwood, 10A
Clayton, MO 63105

Eric Kiebler (314) 725-9492
ihnp4!plus5!sluug

NE – Omaha: meets on the 2nd Thursday of each month.

/usr/group nebraska
P.O. Box 44112
Omaha, NE 68144

Sukan Makmuri (402) 422-8367
ihnp4!ugn!root

New England – Northern: meets monthly at different sites.

Emily Bryant (603) 646-2999
Kiewit Computation Center
Dartmouth College
Hanover, NH 03755

David Marston (603) 883-3556
Daniel Webster College
University Drive
Nashua, NH 03063

decvax!dartvax!nncuug-contact

NJ – Princeton: the Princeton UNIX Users Group meets monthly.

Pat Parseghian (609) 452-6261
Dept. of Computer Science
Princeton University
Princeton, NJ 08544

pep@Princeton.EDU

NY – New York City:

Unigroup of New York
G.P.O. Box 1931
New York, NY 10116

Ed Taylor (212) 513-7777
{attunix,philabs}!pencom!taylor

New Zealand:

New Zealand UNIX Systems User Group
P.O. Box 13056
University of Waikato
Hamilton, New Zealand

OK – Tulsa:

Pete Rourke
$USR
7340 East 25th Place
Tulsa, OK 74129

PA – Philadelphia: the UNIX SIG of the Philadelphia Area Computer Society (PACS) meets the morning of the 3rd Saturday of each month at the Holroyd Science Building, LaSalle University.

G. Baun, UNIX SIG
c/o PACS
Box 312
La Salle University
Philadelphia, PA 19141

{inhp4,cbosgd,rutgers}!{bpa,cbmvax}!
temvax!pacsbb!{gbaun,whutchi}

TX – Dallas/Fort Worth:

Dallas/Fort Worth UNIX Users Group
Seny Systems, Inc.
5327 N. Central, #320
Dallas, TX 75205

Jim Hummel (214) 522-2324

TX – San Antonio: the San Antonio UNIX Users (SATUU) meets the 3rd Wednesday of each month.

William T. Blessum, M.D. (512) 692-0977
7950 Floyd Curl Dr. #102
San Antonio, TX 78229-3955
{gatech,ihnp4}!petro!bles!wtb

The local *uucp* network "postmaster" is:
Bruce Andreen (512) 656-3053
{gatech,ihnp4}!petro!bruce

WA – Seattle: meets monthly.

Bill Campbell (206) 232-4164
Seattle UNIX Group Membership Information
6641 East Mercer Way
Mercer Island, WA 98040

uw-beaver!tikal!camco!bill

Washington, D.C.: meets the 1st Tuesday of each month.

Washington Area UNIX Users Group
2070 Chain Bridge Road, Suite 333
Vienna, VA 22180

Samuel Samalin (703) 448-1908

Future Events

USENIX 1988 Summer Conference and Exhibition San Francisco, June 20-24, 1988

UNIX Security Workshop
Portland, OR, Aug. 29-30, 1988

The Program Chair is Matt Bishop of Dartmouth College. See page 6.

AUUG Winter Conference
Melbourne, Sept. 13-15, 1988

For information write Tim Roper: uunet!munnari!labtam.oz!timr

UNIX and Supercomputers Workshop
Pittsburgh, PA, Sept. 26-27, 1988

Program Chairs are Melinda Shore of the Pittsburgh Supercomputer Center and Lori Grob of New York University. See page 7.

EUUG Autumn Conference
Portugal, Oct. 3-7, 1988

See page 10.

C++ Miniconference
Denver, CO, Oct. 17-21, 1988

The Program Chair is Andy Koenig of AT&T. See page 8.

Large Installation
System Administration II
Monterey, CA, Nov. 17-18, 1988

The Program Chair is Alix Vasilatos of MIT's Project Athena. See page 9.

EUUG Spring Conference
Brussels, Apr. 10-14, 1989

Long-term USENIX Conference Schedule

Jan 31-Feb 3	'89 Town & Country Inn, San Diego
Jun 12-16	'89 Hyatt Regency, Baltimore
Jan 22-26	'90 Washington, DC, Omni Shoreham
Jun 11-15	'90 Marriott Hotel, Anaheim
Jan 22-25	'91 Dallas
Jun 10-14	'91 Opryland, Nashville

Publications Available

The following publications are available from the Association Office. Prices and overseas postage charges are per copy. California residents please add applicable sales tax. Payment **must** be enclosed with the order and **must** be in US dollars payable on a US bank.

The EUUG Newsletter, which is published four times a year, is available for $4 per copy or $16 for a full-year subscription.

The July 1983 edition of the EUUG Micros Catalog is available for $8 per copy.

Conference and Workshop Proceedings

Meeting	Location	Date	Price	Overseas Mail Air	Overseas Mail Surface
USENIX	Dallas	Winter '88	$20	$25	$5
C++ Workshop	Santa Fe	November '87	20	25	5
Graphics Workshop IV	Cambridge	October '87	10	15	5
USENIX	Phoenix	Summer '87	10	25	5
USENIX	Wash. DC	Winter '87	10	25	5
Graphics Workshop III	Monterey	December '86	10	15	5
USENIX	Atlanta	Summer '86	10	25	5
Graphics Workshop I	Monterey	December '84	3	7	5

Minutes of the AUUG Management Committee Meeting
February 29, 1988

1. The meeting opened at 10:11. Present were Chris Campbell (CC), Piers Dick-Lauder (PL), Robert Elz (KRE), John Lions (JL) in the chair, Chris Maltby (CM) (arrived late, as noted below), and Tim Roper (TR). Also present was John Carey (JC), the AUUGN editor. Wael Foda (WF), Steve Jenkin (SJ) and Greg Webb (GW) each attended for a short while (as indicated).
2. The minutes of the previous meeting (December 1987), which had been circulated earlier, were tabled.
3. Moved (TR, seconded CC) **That the minutes be accepted.** Motion postponed.
4. Business arising from the minutes

Items carried forward from previous meetings

— **Usenix 4.3BSD Manuals:** These have mostly been distributed to those who placed orders, though a few copies still remain available. No action has yet been taken towards re-advertising their availability and determining if another order should be placed with USENIX.

— **Database location:** The secretary had been unable to get the database sent to him.

— **Redirection of the AUUG P.O. Box:** Has not been done yet.

— **Secretarial Assistance:** Discussion deferred.

Item 9 **Treasurer's Report:** JL is still chasing NSWIT. JC remarked that he had the impression that none of the members who had joined at the NSWIT conference had been entered into the database yet.

Item 12 **ACSnet SIG legal assistance:** Nothing notable done yet.

Item 15 **Name registration problem:** The solicitor had been contacted, and, after consulting with the Corporate Affairs Department, confirmed that a name change to **AUUG Inc** would solve the problem. AT&T have not been contacted.

Item 30 **President's letter on meeting policy change:** Done.

Item 32 **Replacement Committee Member:** In accordance with the short list drawn up at the previous meeting, the secretary contacted Peter Wishart, who agreed to join the committee.

Item 33 **Meeting Guidelines Documents:** PL reported that they were half done. He is still working on them.

Item 38 **Usenix Journal:** We may have missed our opportunity here, the secretary wasn't able to obtain the number of journals needed when

requested by Usenix. The number was sent eventually, no reply has been received.

Item 41 **Newsletter printing and distribution from Melbourne:** Done, first issue printed that way should be in the mail now.

Item 44 **Badge competition:** Nothing done yet, PL volunteered to handle this.

Item 45 **Next Meeting:** It was noted that the year indicated was incorrect, it should be 1988, not 1987.

5. Moved (JL, seconded PL) **That the minutes be amended to correct the typographical error, and substitute "1988" for "1987" in item 45.** Carried (5/0).

6. Original motion (item 3) Carried (5/0).

7. The president gave a brief report. He indicated that Ken Thompson has accepted an invitation to attend the September AUUG meeting, though he might not be able to attend all 3 days.

8. The meeting adjourned at 10:41 and resumed at 10:43.

9. The newsletter editor (JC) presented a report. He indicated that his report of the proceedings of the last committee meeting appear in the issue of AUUGN currently being printed. Minutes of committee meetings will not be published until they have been confirmed at the next meeting.

   Newsletter printing has been moved to Melbourne. The editor investigated using the Monash print shop, but determined that service would not be good enough. He had decided to continue using Pink Panther, who had indicated that he cost would not be any greater than it had been in Sydney.

   The December newsletter should be out this week, it cost $2950, plus %20 tax (the reason for which is unclear), and not including postage. Pink Panther will handle printing, envelope stuffing, and mailing. 320 copies were printed, 288 are to be mailed, the issue contains 124 pages.

   The Post Office registered publication number has been moved from Strawberry Hills, NSW, to South Melbourne, Vic, without any additional charge, as it happened to be due for renewal.

   The current schedule for coming newsletters is for the next issue, Volume 9 number 1, to appear in March, about 1 month late, and then for volume 9 number 2 to appear in April, which will be back on schedule.

10. JL asked for publication dates to be printed on the cover. It was pointed out that the dates do appear on the first page. JC indicated that he could arrange for the publication date to appear on the cover if desired.

11. Problems: It is unclear what the 20% tax is about, it is probably sales tax, but it is unclear that we are buying anything. It was remarked that we should not pay without justification. JC indicated that he didn't want to distribute extra copies of the newsletter. He said that he has 12 boxes of old newsletters, and asked what we wanted done with them (except for one box, which IPEC had apparently lost). It was agreed that we should attempt to sell back issues at the next meeting, and then discard superfluous copies.

12. JC also noted that the December issue (the issue being printed) had 288 copies, whereas the January issue will have only 230 copies. He indicated that some better renewal scheme was needed, irregularities in the newsletter delivery schedule meant that people tended not to notice that AUUGN was no longer arriving, and were not renewing. he suggested a switch to a fixed renewal date. This was not considered possible.

13. PL considered that the problem could be alleviated if the database was moved to Melbourne, and moved that CM be asked to expedite shipping it to Melbourne, in whatever form appropriate. The motion was not seconded, and was postponed.

14. Moved (PL, seconded TR) **That the editor's report be accepted**, Carried (5/0).

15. CC indicated that there was no ACSnet SIG report to give, as there had not been a meeting since the last committee meeting. The next meeting should happen sometime soon. PL asked about legal advice for the company, it was pointed out that this had been mentioned in business arising from the minutes, and that no action had yet been taken.

16. The meeting adjourned at 11:08 and resumed at 11:14, with CM in attendance.

17. CM indicated, when asked, that we have always paid the 20% tax, which is sales tax. He was also not sure what it is exactly that we were being taxed for.

18. CM also indicated that he had not yet forwarded the database, or the computer on loan to AUUG from Sigma Data, as he was waiting for the secretarial assistance issue to be resolved.

19. The secretary gave his report, in which he indicated that he had no idea of the current state of the association, as he didn't have the membership database. CM indicated that he didn't have it either, it is at UNSW.

20. The secretary also tabled copies of correspondence received and sent. The solicitors are currently in hold mode, waiting for action from us. Mike Lesk and Mike Karels have indicated their willingness to attend the winter meeting, John Mashey has not yet made a firm commitment.

21. SJ arrived.

22. The treasurer presented a current balance sheet for the organisation, and indicated that that did not include $12000 currently on term deposit. No progress has yet been made towards moving more funds from the cheque account to a higher interest bearing account. He indicated that AIDC currently have 12% at call rates available, CC questioned the wiseness of such a move. It was generally agreed that a bank account, similar to that already in use, was the right approach.

23. Moved (PL, seconded TR) **That the treasurer's report be accepted** Carried (6/0).

24. The treasurer then indicated that he would not be seeking re-election at the end of his current term, and sought authority to spend money to get extra advice, and for the accounts to be audited. JL asked whether we could not seek the services of our original auditor.

25. JL also remarked on the state of the membership database, CM indicated that it might be possible to gain some assistance from Elaine (from Softway).

26. It was noted that Ken McDonell had been unable to locate anyone willing to undertake the type of secretarial work we are seeking to have done. TR undertook to continue this search, including writing a job specification, etc.

27. The database is to be moved to Melbourne, the Post Office box is to be redirected, and the labels (physical things with sticky backs) are to also be sent to the secretary in Melbourne.

28. The meeting adjourned at 12:19 and resumed at 14:05, without SJ, but with WF and GW.

29. WF presented plans of the meeting rooms at the Southern Cross, and some alternatives for layouts, including his recommendations. The conference facilities are to be available at no cost, provided that a minimum of 100 people attend each catered function, viz: morning and afternoon tea breaks at $3 per person twice each day, lunches at $20 per person, each day, cocktails at $14.50 per person, once, and dinner, at $45 per person, once.

30. The dinner is to be held on the Wednesday night.

31. There was some discussion of the tentative budget that had been presented and agreed to at the previous meeting. WF indicated that he didn't believe that we could do other than lose with the attendance fees set at the levels suggested.

32. After some discussion, the committee agreed on a new rate structure, being $200 for members (including lunches, and cocktails), $250 for non members, with an additional $50 late fee. The dinner cost would be $30. WF suggested single day attendance rates be provided, which was agreed to, and set at $100 per day, with no member's discount. It was agreed that $80 (early) and $130 (late) were

reasonable rates for student registrations (excluding lunches, etc).

33. Meeting timing: It was agreed that June 30 be the early registration deadline. A brochure would need to be prepared by mid May.

34. Conference advertising was discussed next. It was noted that a mass mailing would cost approximately $1000, and advertisements in 3 Australia wide publications (The Age, the Australian, and Computing Australia), which should be placed in mid May would cost about $3000.

35. PL suggested that perhaps Ken Thompson might be persuaded to give one or two newspaper interviews, which might assist with publicity.

36. A brochure is likely to cost $2000, plus about $1500 for assistance with its preparation. It was agreed that the brochure should be 3 A4 pages, folded.

37. WF departed at 15:25.

38. GW indicated that he expected to have final information from the September 87 NSWIT conference withing 3 weeks. He anticipated a net profit of approximately $8500, meaning a net return to AUUG of approximately $12000 including the initial float.

39. A Call for papers for the coming meeting should appear in the coming AUUGN, its deadline is March 11.

40. CC is to arrange badges for attendees at the conference, and is suggested he order approximately 250 at anything up to $3.50 each.

41. Moved (PL, seconded JL) **That CC order 450 badges, marked "AUUG 88" at a cost of less than $5 each.** Carried (6/0).

42. The programme committee should handle the best student paper competition.

43. The issue for a student travel subsidy was canvassed, no real decision was reached, general consensus was to continue with the status quo, which allows such a subsidy to be granted upon application from a student whose paper has been accepted.

44. It was noted that the conference still needs a conference organiser, the job needs to be specified.

45. The secretary is to write to ACMS, and obtain details of exactly what they will be doing, so the committee can determine what is left to be done.

46. The meeting adjourned at 16:22 and resumed at 16:36.

47. GW agreed to write a job description for AUUG conference organiser, and transmit it to JL.

48. It was noted that during the adjournment, Greg Rose had **volunteered** himself as conference chairman! It was suggested that he should be asked to prepare a draft brochure.

49. GW departed at 16:44.

50. Incorporation has stalled waiting for action on the name issue.

51. Moved (PL, seconded KRE) **That a ballot be held in conjunction with the May election to change the name to "AUUG Inc".** Carried (6/0).

52. Because of this, no letter to AT&T is required.

53. The secretary is to talk to the solicitor again, with regard to trade marks, etc.

54. Moved (TR, seconded CM) **That the ballot be held as soon as possible, in the next AUUGN (11th March)** Carried (6/0).

55. CM and KRE are to prepare a budget for FY 1988/1989.

56. Moved (CM, seconded PL), **That the membership and subscription rates for 88/89 be set at $65 for Members and Newsletter Subscriptions, $40 for students, and $300 for institutions.** Carried (6/0). It was noted that the Member rate was still too low, but that increasing it to a sensible value would be too large a jump in one year.

57. Discussion of benefits for institutional members was deferred.

58. Discussion of potential constitutional changes was deferred until after incorporation.

59. The next meeting is to be in Sydney, Thursday May 5, in the Bevington Room in Softway's building if that can be arranged.

60. The treasurer is to get an accountant to prepare the books before the next meeting.

61. The smaller, state wide, February meetings are to go on as planned at the last meeting in February 89.

62. The next AUUGN should hold a call for nominations for the next AUUG elections. We want to obtain the best possible committee for the coming year.

63. The meeting closed at 17:28.

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.

Robert Elz

AUUG Secretary.

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 $55.00
- ☐ Renewal/New* Student Membership $30.00 (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 ☐ Mastercard ☐ Visa.

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 $250.00

☐ 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 ☐ Bankcard ☐ Mastercard ☐ Visa.

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 $ 55.00
- ☐ 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 ☐ Mastercard ☐ Visa.

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#* __________