Australian UNIX systems User Group Newsletter

AUUGN

Volume 10, Number 4

August 1989

AUUG89 Conference Issue

Registered by Australia Post, Publication Number NBG6524

Callum Gibson

The Australian UNIX* systems User Group Newsletter

Volume 10 Number 4

August 1989

CONTENTS

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

AUUG General Information

Memberships and Subscriptions

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

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

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

General Correspondence

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

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

AUUG Executive

President

Greg Rose

greg@softway.sw.oz
Softway Pty. Ltd.
New South Wales

Secretary

Tim Roper

timr@labtam.oz
Labtam Information Systems Pty. Ltd.
Victoria

Treasurer

Michael Tuke

mjt@conjure@labtam.oz
Vision Control Australia
Victoria

Committee Members

Peter Barnes

pdb@uqcspe.cs.uq.oz
Computer Science
University of Queensland

John Carey

john@labtam.oz
Labtam Information Systems Pty. Ltd.
Victoria

Pat Duffy

pat@pta.oz
Pyramid Technology Australia
New South Wales

Chris Maltby

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

Next AUUG Meeting

Details of the AUUG90 Conference and Exhibition will be announced at AUUG89.

Further details will be provided in the next issue.

AUUG Newsletter

Editorial

Welcome to the special AUUG89 Connfernce issue of Newsletter. This issue contains the papers that will be presented at this Conference. I hope you will enjoy hearing our range of speakers which our Programme Chairman, Peter Barnes has assembled for this occasion.

This will be my last Newsletter I will be prepare for AUUG, after 3 years and 18 issues. It has been long haul and I look forward to a change of scene. I would like to thank Ken J. McDonell for getting me into this mess in the first place. Seriously, thank you kenj, for having the faith in me to do the job. Also thanks to Tim Roper, Robert Elz, Chris Maltby, and Kartharine Ching who have assisted me with the AUUGN production and distribution. I hope my contributions to the format and logistics of AUUG production will useful in building an even better Newsletter in the future. My only regret is that I have failed to keep local content as high as it should be, especially when AUUG conferences aren't providing an imputeus for papers.

I wish the new editor David Purdue all the best and every success.

Unfortuately you are not rid of me yet as I will be working on the General Committee this year and be looking to help carry the User Group forward in its endevours.

The Newsletter and the Annual Conference are the public face of AUUG and need your continuing and increased support. PLEASE CONTRIBUTE to the effort if you wish AUUG thrive and grow.

Goodbye for now - John Carey.

PS REMEMBER, if the mailing label that comes with this issues is highlighted, it is time to renew your AUUG Membership.

AUUGN Correspondence

All correspondence reguarding the AUUGN should be addressed to:-

David Purdue
AUUGN Editor
Labtam Information Systems Pty. Ltd.
43 Malcolm Road
Braeside Victoria 3195
AUSTRALIA

ACSnet: *davidp@labtam.oz*

Phone: +61 3 587 1444
Fax: +61 3 580 5581

Contributions

The Newsletter is published approximately every two months. The deadline for contributions for the next issue is Friday the 20th of October 1989.

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 30 mm left at the top and bottom so that the AUUGN footers can be pasted on to the page. Small page numbers printed in the footer area would help.

Advertising

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

Mailing Lists

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

Back Issues

Various back issues of the AUUGN are available, details are printed at the end of this issue.

Acknowledgement

This Newsletter was produced with the kind assistance and equipment provided by Labtam Information Systems Pty Ltd.

Disclaimer

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

Adelaide UNIX Users Group

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

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

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

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

Dennis Jarvis,
Secretary, AdUUG.

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

PHONE: +61 8 268 0156
UUCP: {decvax,pesnta,vax135}!mulga!aegir.dmt.oz!dhj
ARPA: dhj%aegir.dmt.oz!dhj@seismo.arpa
CSNET: dhj@aegir.dmt.oz

WAUG
Western Australian UNIX systems Group

PO Box 877, WEST PERTH 6005

Western Australian Unix systems Group

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

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

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

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

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

Glenn Huxtable,
Membership Secretary, **WAUG**

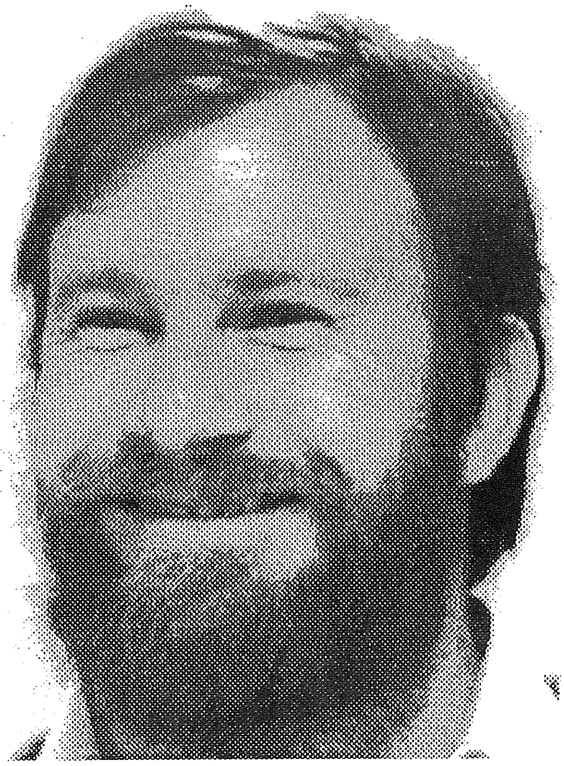
AUUG Institutional Members

ACUS / UNISYS
Adept Business Systems Pty Ltd
Aldetec Pty Ltd
Apple Computer Australia
Apscore International Pty Ltd
Australian Electoral Commision
Australian Nuclear Science & Technology Organisation
Australian Wool Corporation
Autodesk Australia P/L
BHP Melbourne Research Labs
Ballarat Base Hospital
Basser Department of Computer Science
CSIRO DIT
CSIRO Division of Manufacturing Technology
Centre for Information Tech & Comms
Civil Aviation Authority
Comperex (NSW) Pty Ltd
Computer Power IR+D, NSW Branch
Computer Software Packages
Computerscene International
Corinthian Engineering Pty Ltd
Cybergraphic Systems Pty Ltd
DBA Limited
Data General
Davey Products Pty Ltd
Dept of Agricultural + Rural Affairs
Dept of Industry, Technology and Resources, Victoria
Dept of Lands - Central Mapping Authority
Digital Equipment Corporation (Australia) Pty. Limited
Earth Resource Mapping Pty Ltd
Elxsi Australia Ltd
Flinders University Discipline of Computer Science
Fujitsu Australia Limited
Gould Electronics Pty Ltd
Great Barrier Reef Marine Park Authority
Harris & Sutherland Pty Ltd
Hewlett Packard Australia Limited
Hewlett-Packard Australian Software Operation
IPS Radio and Space Services
Ipec Transport Group
Labtam Information Systems Pty Ltd
Lands Department, Qld
Lionel Singer Corporation

AUUG Institutional Members

Macquarie Bank Limited
Macquarie University
Main Roads Department, Queensland
Monash University Computer Science
Motorola Communications Australia
NEC Information Systems Australia Pty Ltd
NSW Parliament
National Engineering Information Services P/L
Nixdorf Computer Pty Limited
Olivetti Australia Pty Ltd
Olympic Amusements P/L
Overseas Telecommunications Corporation
Performance Systems
Prentice Computer Centre
Prime Computer of Australia Ltd
Q. H. Tours Limited
Racecourse Totalizators Pty Ltd
Reark Resources
SEQEB
Sigma Data Corporation Pty Ltd
South Australian Institute of Technology
Sphere Systems Pty Ltd
State Bank Victoria
State Library of Tasmania
Sun Microsystems Australia
Swinburne Institute of Technology
Tattersall Sweep Consultation
Telecom Network Engineering - C.S.S
The Australian National University
The Department of Industry Technology and Commerce
The University of Adelaide
The University of Melbourne
The University of New South Wales
The University of Wollongong
University of New England
University of Technology Sydney Computing Services Division
Vicomp
Wang Australia Pty Ltd
Yartout Pty Ltd

President´s Letter



Welcome to the Proceedings of the AUUG'89 Conference.

Hopefully, as you read this, you will be surrounded by the glamour of the Sydney Hilton, with the excitement of the Exhibition of New Things just up the stairs, and an impressive list of speakers just waiting to address you.

More seriously, AUUG'89 is an important event. UNIX is entering the growth phase that some pundits predicted ten to fifteen years ago (and that others are still predicting to be ten years away, if ever). UNIX has suddenly been seen to be of great commercial importance, and this conference is the largest yet to be run by AUUG with a clearly commercial focus.

The theme of the conference is the proposition that *"Nobody ever got fired for buying UNIX"*. This is clearly untrue, as it is written in the past tense, and I know one person about five years ago... but that is a different story. The *intent* of the statement is that in the *future*, UNIX will be one unquestioned attribute of a correct purchasing decision. In many instances this is already the case, but I believe that it is in the best interests of the computer industry as a whole for us to work toward the universality implied in the statement.

That is, after all, what UNIX offers – universality.

The fact that the conference theme contains the word "*buying*" also demonstrates AUUG's commitment to servicing the entire spectrum of UNIX users, not only the academics and researchers who brought it to life and prominence, or the computer manufacturers whose very existence depends upon the availability of a portable, fully functional operating system. Many of the real users of UNIX today have never heard of it – but they should not be ignored, and AUUG is focussing more of its efforts toward helping these people, and the industry in general.

I hope you have been able to attend the AUUG'89 conference and Exhibition, and that it, and these proceedings, are of great use to you. If they are, it will be because of the great efforts of many people, and I would like to extend to them, "Many thanks".

Regards,

Greg Rose AUUG President.

(Note to the Editor: If you publish my picture with this, we'll have to find another editor...)

(Note from the Editor: Sorry Greg, you talked me into it, since this is my last issue couldn't resist)

Comments from the Programme Committee Chair

AUUG89

Peter Barnes

Key Centre for Software Technology
Department of Computer Science
University of Queensland

I accepted this position with some trepidation, after last year's successful conference. Could we possibly equal, or even better that performance? Having seen the papers for AUUG89, I believe we can claim to have done that. That is to say, this year's authors have done that, for the success of any conference derives, eventually, from the quality of its papers, and this conference is no exception. AUUG has achieved several milestones in that area this year.

Firstly, we have a greater breadth and depth of experience from our invited speakers than ever before. **Dennis Ritchie** is almost impossible to introduce without resorting to clichés. We are certainly honoured to have him at AUUG89. One of the parents of UNIX, he has continued in that role over the years, making further key contributions to its development and maturation. In his keynote address, he surveys his (now perhaps grown-up) child and its relations.

James Gosling, familiar to many for EMACS, brings to us considerable expertise in windowing systems, and is joined in that area by **Stephen Brown**, who will discuss user interfaces. Their insights into this rapidly evolving area, one of great importance to the commercial success of UNIX, should be of interest to all.

We are also pleased introduce **Sunil Das** from UKUUG, who brings us light-hearted news of their activities. Sunil will also be presenting a (serious) paper on aspects of an interactive spelling corrector. Lastly, we welcome back **Ross Bott**. Those who heard Ross' paper last year (on ABI and OSF) will be eagerly anticipating his paper on the commercial computing engine of the 1990's.

It is in the emphasis on commercial UNIX that AUUG achieves its second milestone, in hosting the largest UNIX conference in Australia, with a theme that is particularly relevant - UNIX in the commercial marketplace.

The third milestone is in the number, and quality of papers submitted for this conference. For the first time, we have had, reluctantly, to reject nearly as many papers as were accepted, such was the response to the Call for Papers. This is a very good omen for future conferences, and we hope that many papers which could not be included in this conference will be re-submitted for the proposed regional Summer Conferences next year.

Of course, having twice as many submissions as programme slots puts a considerable burden on the Programme Committee, and I would like to thank the members of the Committee for their timely and considerable efforts. Likewise, I should like to thank all authors for their patience with the inevitable slippages and re-organizations. Many thanks, as ever, go to John Carey, for bearing the brunt of all slippages, and working typesetting miracles to produce these proceedings. Finally I should like to thank my employer for the time and resources necessary for such (inter)national organization.

I hope that you will find much that it interesting and relevant in AUUG89, whether you be expert or neophyte, programmer or manager, academic or business person. I look forward to hearing *your* paper at the next conference.

AUUG 89

Australian UNIX* systems User Group
1989 Conference and Exhibition

Wednesday 9th to Friday 11th August, 1989
Hilton International, Sydney

PROGRAMME

DAY 1 - WEDNESDAY, AUGUST 9th

0800-0930	Registration
0930-1030	Keynote Address: *UNIX - a Dialectic* Dennis Ritchie, AT&T Bell Laboratories
1030-1100	Morning tea and Exhibition viewing
1100-1200	*UNIX User Interfaces* Stephen J. Brown, Hewlett Packard
1200-1230	*Adaptable Text Interfaces* Michael Paddon and David Spaziani, University of Melbourne
1230-1400	Lunch and Exhibition viewing
1400-1430	*Sun, Surf and X in California - a report on Xhibition 89* Andrew McRae, Megadata Pty. Ltd.
1430-1530	*The NeWS Window System: A look under the hood* James Gosling, Sun Microsystems
1530-1600	Afternoon tea and Exhibition viewing
1600-1700	*Storage and Retrieval Methods for an Interactive Spelling Corrector* Sunil K. Das and Philip M. Sleat, Computer Science Department, City University London
1700-1730	Break
1730-1900	Cocktail Reception, sponsored by Applied Computing Pty. Ltd.

DAY 2 - THURSDAY, AUGUST 10th

0800-0900	Registration
0900-0930	*Strategies for Writing Graphical UNIX Applications Productively and Portably* Stephen J. Brown, Hewlett Packard
0930-1000	*UNIX and Internationalisation* Andrew Tune, Technix Consulting Services, Pty. Ltd.
1000-1030	*ANSI C: So What?* Bruce Ellis, C-Side Software Services
1030-1100	Morning tea and Exhibition viewing
1100-1130	*The UNIX Software Operation and System V.4 status* Larry Crume, AT&T UNIX Pacific Co. Ltd. Japan
1130-1200	*Open Systems through an Open Process* Gary Oden, OSF - Open Software Foundation
1200-1230	Question Time
1230-1245	Sigma Data Award Presentation
1245-1400	Lunch and Exhibition viewing
1400-1430	*UNIX and Artifical Intelligence* Jason Catlett, Basser Department of Computer Science, Sydney University
1430-1500	*Open Electronic Messaging and its role in EDI* Ian Sharrock, ICL Australia
1500-1530	*Security in Standard Environments* Robert A. Michael, The Santa Cruz Operation, Inc.
1530-1600	Afternoon tea and Exhibition viewing
1600-1630	*Interprocess Communication in the Ninth Edition UNIX System* D. L. Presotto, Dennis Ritchie, AT&T Bell Laboratories
1630-1700	*UNIX User Groups: A Report from Europe* Sunil K. Das, Computer Science Department, City University London
1700-1800	AUUG Annual General Meeting
1800-1930	Break
1930-2300	Conference Dinner, sponsored by Pyramid Technology Australia

DAY 3 - FRIDAY, AUGUST 11th

0800-0900 Registration

0900-0930 *UNIX User Limits Revisited*
Ray Loyzaga, Basser Department of Computer Science, Sydney University

0930-1000 *Macintosh system emulation under A/UX, Apple UNIX*
Kent Sandvik and Philip Cookson, Apple Australia

1000-1030 *User Mode File Servers*
Bruce Janson, Basser Department of Computer Science, Sydney University

1030-1100 Morning tea and Exhibition viewing

1100-1130 *ACE: A syntax-driven C preprocessor*
James Gosling, Sun Microsystems

1130-1200 *Enhanced Error Processing for UNIX Parsers*
Arnold Pears and Rhys Francis, Department of Computer Science, La Trobe University

1200-1230 *No one ever got fired for delivering on time*
Stephen Young, NEC Informations Systems Australia

1230-1400 Lunch and Exhibition viewing

1400-1430 *Problems Facing Corporate Distributed Database Systems*
Russell Burstow, ICL Australia Ltd., and
Shaun Travers, Main Roads Department (Queensland)

1430-1530 *The Commercial Computing Engine of the 1990s*
Ross Bott, Pyramid Technology Corporation

1530-1600 Afternoon tea and Exhibition viewing

1600-1630 *OLTP Performance - What's Behind the Smoke and Mirrors*
Ken McDonell, Pyramid Technology Corporation

1630-1700 *What's Gold Lotto Online? - UNIX*
Mark Pickering and David Moles, DMP Software

AUUG 89 Keynote and Invited Speakers Biographical Notes

AUUG is honoured to have one of the fathers of Unix,

DR DENNIS M. RICHIE delivering the keynote address. Dennis was born in 1941, and received his Bachelor's and advanced degrees from Harvard University where, as an undergraduate, he concentrated in Physics and, as a graduate student, in Applied Mathematics.

Since joining the technical staff of AT&T Bell Laboratories in 1968, Dennis has worked on the design of computing languages and operating systems. After contributing to the Multics project, he joined Ken Thompson in the creation of the Unix operating system, and designed and implemented the C language in which Unix is written.

In 1982, Dennis shared the IEEE Emmanual Piore award with Thompson and, in 1983, he and Thompson won the ACM Turing Award. He was elected to the National Academy of Engineering in 1988. His current research is concerned with structure of operating systems.

Dennis' keynote speech will be *Unix: A Dialectic.*

He will examine some of the characteristics that contributed to the success of the Unix system: the simplicity and power of its model of computation, the culture of tool-making and tool-using, and portablility. Each of these have inherent negative aspects as well.

For example, the simplicity of the underlying model encourages the assumption that the user understands the model and is comfortable in using it. Similarly, the decomposition of tasks into simpler tools means that the user must be familiar with the tools, instead of being presented with a unified interface specialised to the task that the user needs to do. Finally, portability of the operating system has necessarily led to several variants of the system, which tends to defeat portability.

Dennis will also present a paper on *Interprocess Communication in the Ninth Edition Unix System.*

INVITED SPEAKERS

DR. JAMES GOSLING received a bachelor of science in Computer Science from the University of Calgary, Canada in 1977. He received his PhD in Computer Science from Carnegie-Mellon University in 1983. He has built satellite data acquisition systems, a multiprocessor version of Unix, several compilers, mail systems and window managers. He is very well known for his EMACS text editor for Unix systems and has also built a WYISWYG text editor and a constraint based drawing editor.

James will present two papers, *the NeWS Window System: A Look under the Hood*, and *ACE: A Syntax-Driven C Preprocessor.* The first will cover some of the issues inside NeWS, although a brief overview of NeWs will be included. Design decisions behind NeWS, particularly the use of the PostScript language will be discussed. It will cover the awkward problems in the language that led to alterations, but it will also cover some of the beautiful aspects of the language that lead to the elegant solution of many problems.

DR. ROSS BOTT, 37, is Vice-President Strategy and Planning, Pyramid Technology Corporation. Ross has a bachelor of science in mathematics from Stanford University and a PhD in artificial intelligence from the University of California, San Diego.

Prior to joining Pyramid, Ross spent four years at Xerox's Palo Alto Research Center on the Star workstation project, the precursor to the Apple Macintosh. He joined Pyramid in 1982 and has held various positions, including manager of the group that designed Pyramid's dualPort OSx operating system that combined AT&T System V and Berkeley Unix. He was also Manager of the Networking and Communications Group, Director of Software, Vice President of Advanced Architectures, and Vice President of International Marketing.

Ross will present a paper on *The Commercial Computing Engine of the 1990's.* He discusses how relational database management systems (RDBMS) and the client-server architecture they help to implement will have broad benefits for the next generation of commercial applications. They provide an elegant basis for distributed computing, freedom from slavery to a single instruction set or architecture, and much greater programmer productivity in an era where they are our most precious resource.

However, the revolution does not come for free. The performance and capacity requirements mandated by RDBMSs and client-server software are suprisingly (and remarkably) large. In this presentation Ross will parameterise some of the requirements and consider how the next generation of commercial computers must be designed to meet this challenge.

SUNIL K. DAS was awarded a bachelor's degree in Mathematics by the University of Surrey and a master's degree in Computing and Numerical Analysis by the Victoria University of Manchester. Sunil first encountered the Unix system in 1977 while employed as a research fellow in the Computer Networks Research Group at University College London where he was investigating distributed computing environments, local and wide area networking, and network computer systems.

In 1980 he joined the academic stuff of City University's Computer Science Department where his research includes concurrent and parallel programming, systems programming and software engineering.

Sunil is also chairman of the UK Unix Systems User Group (UKUUG). He has lectured on Unix and C related topics in North America, throughout Europe and in Asia.

At AUUG 89 Sunil will present two papers. The first will look at the role and functions of Unix User Groups in Europe, networking and standards. It will provide a non-technical overview of the news in the Unix world in Europe. The second paper describes the storage and retrieval methods for designing and building an interactive spelling corrector capable of running on a Unix system. This will be followed by an account of the design, coding and use of the spelling corrector.

STEPHEN J. BROWN is Product Line Manager, Unix User Interface Products for Hewlett-Packard. He has been at HP for eight years, joining as an R&D engineer. He spent four years developing several microcomputers which ran the Unix operating system. For the past four years he has worked in Product Marketing and has had project management responsibility for new product development for HP's Unix User Environments.

In addition to the work on HP's User Interface submission to Open Software Foundation (what is now OSF/Motif). Steve's group is currently working on future programs in the area of system user interface technology and bringing the HP NewWave technology to Unix.

Steve holds a Bachelor of Science degree in Mechanical Engineering from California Polytechnic State University and is a registered Professional Engineer in the State of California. Steve will present a paper on the advances in user interface technology. Starting with the emergence of key "foundation" software technology such as the X Window System, he will discuss emerging user interface standards such as Presentation Manager and OSF/Motif. Emerging object-oriented technology promises to provide the "foundation" layer which will enable the industry to make even greater strides in user interface technology.

The discussion will cover how far the industry has come with human/computer interface technology and how far we still have to go in order to provide users with the capability to access the full potential of their computing systems.

GARY ODEN is the director of American Operations for the Open Systems Software Foundation, and is responsible for all membership activities for North America, South America, Australia and New Zealand. Oden has bought 20 years of industry experience to OSF. He came to OSF from Digital Equipment Corporation where his most recent position was group manager for Ultrix product development. Gary holds a BS from the University of Connecticut and an MBA from Clark University.

His presentation is titled *Open Systems through an Open Process* and will cover the role of OSF in providing an Open Software Computing Environment.

AUUG 89 Conference Papers

In order of appearance

UNIX - a Dialectic
Keynote Address
Paper
Dennis Richie
AT&T Bell Laboratories

User Interfaces
Paper Unavailable
Stephen J. Brown
Hewlett Packard

Sun, Surf and X in California - a report on Xhibition 89
Preliminary Paper
Full Paper printed at end of this issue
Andrew McRae
MegaData Pty. Ltd.

The NeWS Window System - A look under the hood
Paper
James Gosling
Sun Microsystems

Adaptable Text Interfaces
Paper
Michael Paddon
David Spaziani
University of Melbourne

Strategies for Writing Graphical UNIX Applications Productively and Portably
Paper
Jannet Dobbs
presented by Stephen J. Brown
Hewlett Packard

Open Electronic Messaging and its role in EDI
Paper
Ian Sharrock
ICL Australia

AUUG 89 Conference Papers

continued

UNIX User Limits Revisited

Paper

Ray Loyzaga
Basser Department of Computer Science
Sydney University

The UNIX Software Operation and System V.4 Status

Paper Unavailable

Larry Crume
AT&T UNIX Pacific Company Limited, Japan

Open Systems through an Open Process

Paper Unavailable

Gary Oden
OSF - Open Systems Foundation

UNIX Internationalisation: recent developments

Paper

Andrew Tune
Technix Consulting Services Pty. Ltd.

UNIX and Artificial Intelligence in the '90s

Abstract

Jason Catlett
Basser Department of Computer Science
Sydney University

Storage and Retieval Methods for an Interactive Spelling Checker

Paper

Sunil K. Das
Philip M. Sleat
Computer Science Department
City University London

Interprocess Communication in the Ninth Edition UNIX System

Paper

Dennis Richie
AT&T Bell Laboratories

AUUG 89 Conference Papers

continued

Security in Standard Environments

Preliminary Paper

Robert A. Michael
The Santa Cruz Operation Incorporated

UNIX User Groups: A Report from Europe

Paper Unavailable

Sunil K. Das
Computer Science Department
City University London

Enhanced Error Processing for UNIX Parsers

Paper

Arnold Pears
Rhys Francis
Department of Computer Science
La Trobe University

Machintosh system emulation under A/UX, Apple UNIX

Paper

Kent Sandvik
Apple Australia

User Mode File Servers

Paper

Bruce Janson
Basser Department of Computer Science
Sydney University

ACE - A syntax-driven C preprocessor

Paper

James Gosling
Sun Microsystems

AUUG 89 Conference Papers

continued

ANSI C: So What?

Paper

Bruce Ellis
C-Side Software Services

No one ever got fired for delivering on time

Paper

Stephen Young
NEC Information Systems Australia

Problems Facing Corporate Distributed Database Systems

Paper

Shaun Travers
Main Roads Department (Queensland)

The Commercial Computing Engine of the 1990s

Paper Unavailable

Ross Bott
Pyramid Technology Corporation

OLTP Performance - What's Behind the Smoke and Mirrors

Paper

Ken McDonell
Pyramid Technology Corporation

What's Gold Lotto Online? - UNIX

Paper

David Moles
Mark Pickering
DMP Software Pty. Ltd.

Unix: A Dialectic

Dennis M. Ritchie

AT&T Bell Laboratories
Murray Hill,
NJ 07974 USA

This paper is about why the Unix® system has succeeded, and why it is good, and how its virtues lead to limitations. It is not about what it doesn't try to do, or even about what it tries to do but does badly, but instead is about problems that arise out its very nature and history. I will discuss its computational model, its use of tools, and portability.

The Model

The Unix kernel embodies a simple, coherent, and powerful model of computation. The operating system itself has only two underlying concepts: the file system, and the process. Once these are understood, the system as a whole is mastered. Moreover, it should be possible to understand them; they were designed to be comprehensible, in the sense that a 9-page paper, published in CACM more than a decade ago [1], still constitutes a description nearly complete enough to serve as an implementation plan.

The file system has two aspects: its static structure, and the operations that can be performed on it. The static structure is specified by the set of file names, and their organization, which is a tree of arbitrary depth. Some files are directories, and contain names of other files and subdirectories. Directories can be read, to enumerate the files they contain.

All ordinary files are simply sequences of bytes, without any system-imposed structure. Text files, in particular, are nothing but characters, with new-line characters to delimit lines. There are no ''access methods'' or ''file types.'' Of course, programs can write files with any structure they please, but the custom is to avoid complicated formats whenever possible.

The operations are few: create a file, delete a file, read bytes from a file, write some bytes. A newly created file replaces an old one of the same name, and files grow as necessary.

The file system operations generalize to objects other than ordinary disk files. For example, tapes and terminals and network connections behave, to the extent possible, in the same way as disk files. Each is, in its own way, complicated and unique, but a largely successful effort is devoted to making these things behave in the same way as files, so that a program seldom needs to know what kind of object it is talking to. In particular, devices have names in the file system, the same protection mechanism applies to them as to regular files, and the same I/O system calls are used.

The characteristic form of interprocess communication, likewise, is treated like file I/O. The pipe is a connection between two processes, whereby one process writes data that another reads, with buffering and synchronization handled transparently by the system. Of course, the ordinary read and write calls apply to pipes too.

The other fundamental object is the process: a program in execution. New processes are created when an existing process splits, or *forks.* Often the new process immediately performs an *execute* operation–that is, it calls in a new program from a file and causes this program to start running. Processes have only certain kinds of interaction with the rest of the universe. First, when a new program starts executing, it receives several character string arguments via the request that started it. Thereafter, it carries out I/O to files. Some of these it opens or creates by itself, often getting the file names from its arguments. There is a convention by which processes start off with a ''standard input'' and ''standard output,'' which are pre-opened files that often refer to the terminal but can be redirected elsewhere.

A standard command interpreter, called the shell, works this way: First, it reads a line from its standard input; this line specifies a file containing a command, and some arguments. A new process is created, and the new process calls in the named command, passing it the given arguments. If requested, the shell will adjust the standard input or output of the command to take input from, or place output in, a named file, or it will connect several processes to form a pipeline, in which the output of one command is connected to, and processed by another.

Tools

The characteristic style of user-level programs exploits the metaphor of the toolbox: the commands constitute a set of software tools, each performing a limited function, that can be combined in powerful and interesting ways. The books of Kernighan and Plauger (see, for example, reference 2) show how the approach can succeed in environments other than Unix. However, the tools work especially well in the world provided by the shell, with its notation for combining programs.

The tool metaphor has led to an explosion of interesting commands. Many of them are small: they are the nuts and bolts of the toolbox. They do things like concatenate files, count words, and search for text patterns. The approach has encouraged new ideas about how to design programs, and how to represent data. The crucial perception is that the output of any program is potentially the input of another. Elementary examples: one asks how many users there are on the machine by sending the output of the *who* command into the command that counts lines; one counts the distinct words in a document by splitting the document into one word per line, sorting, casting out duplicates, and tallying what results.

Larger programs, more akin to power tools, often embody ''little languages,'' and are often special-purpose processors of text. This is most evident in our approach to word-processing. The powerful if rebarbative programs *nroff* and *troff,* developed by Ossanna, provided a way to format documents consisting of ordinary text. The first significant preprocessor for it was *eqn,* by Kernighan and Cherry; it provided a language for describing mathematical equations. Others include *tbl* (Lesk) for setting tables; *pic* (Kernighan) and *ideal* (VanWyck) for incorporating line-graphic material; *refer* (Lesk) for consulting a database of references and incorporating them into the text*. *All of these programs are described in most volumes of the *Unix Programmer's Manual.*
Most recently, *grap* (Kernighan and Bentley [3]) has appeared; it is used for graphs.

The important observation is that it seems certain that these programs would never have been developed if they had to be incorporated into *troff.* The structure of the operating system, and the type of thinking it induced, encouraged modularity: each tool could be developed independently of the typesetter's internals.

Portability

A third contribution to the success of Unix is portability. The system is written in C, a reasonably expressive, medium-level language; programs written in it can be moved to a variety of kinds of hardware. The operating system proper makes relatively modest demands on the hardware, and itself is relatively easy to move. Moreover, the system is widely available in source form under remarkably liberal licenses. Consequently, many groups had a chance to contribute to the system, and it was tied neither to the machine on which it was developed, nor to the group that developed it.

People used to think of operating systems as givens, as lumps provided by the manufacturer that they must simply learn, and perhaps petition for changes in. Because of its history as a research effort not a product, this has been much less true for Unix: as it was being developed, people inside AT&T, and in universities, saw it simply as a program that could be understood and changed. Most notably, this apperception occurred within the Programmer's Workbench group at Bell Labs (they largely merged into the Unix Support Group or USG, and that group, much larger now, has since become part of AT&T Information Systems), and also at the University of California at Berkeley. There are those who think that the Seventh Edition distribution (or perhaps even the Sixth Edition) from Bell Labs research is the loveliest flowering of the system, but System V and BSD 4.2 or 4.3 are the versions that most people use. Certainly, the contributions from many separate sources were necessary to the system's success.

Portability to variegated hardware was also vital. Unix prospered early because it was first available for a popular machine, the PDP11, and was later moved to another popular machine, the VAX (after portability was demonstrated on a rare one, the Interdata 8/32). It has now propagated to nearly every important machine architecture, from the 8086 to the Cray 2. This development is important, if for no other reason than that availability of a common, understood environment for computing is a valuable good. This value is seen, for example, in the persistence of old languages, such as Fortran. Few, these days, defend the language itself. Numerical analysts, however, are realists: they would prefer to write in other languages, but understand that when an algorithm is expressed in Fortran, it can be communicated to their colleagues, that there are good Fortran compilers on most machines, and that by using it they are increasing their own effort but maximizing their contribution to society.

Thus, these three technical characteristics contributed to the success of Unix:

1) simplicity and coherence
2) the tool-using approach
3) portability

Each of these virtues, individually and in combination, has negative aspects. Although I think the negative aspects–the "dark sides," are considerably outweighed by the virtues, they are fundamental and have to be faced.

Simplicity and Coherence?

Consider "simplicity and coherence." The first fact is that the system is not so simple as it once was; time and reality have complicated it. For example: most operating systems have an ugly file system, with operations so low-level and complex as to be impossible for application programs to use, and an I/O library to paper over the complexities and ugliness. In part, the file system was designed to make this step unnecessary, so that one could actually understand the operating system interface. Nevertheless, the "standard I/O library" had to be invented for various reasons: buffering for efficiency, and for portability of programs to systems other than Unix. Unfortunately, the I/O library has itself become complicated–more complicated, indeed, than the underlying calls, yet it is the interface that most programs actually use. Why does this matter? For example, the underlying file system can make promises about atomicity of operations. However, these promises are lost when I/O is buffered by the standard library.

The intended simplicity, and concomitant intention that the workings of the system should be fully understood and predictable, leads both to a certain curtness in the documentation and to some blind spots in the services provided. For example, a utility to repair damaged file systems did not appear until relatively late in the development of the system; until *fsck* was written, administrators, at least, not only needed to know the structure of the file system, but how to fix it when it was broken.

More fundamental is this fact: Unix is simple and coherent with respect to a certain model of what you might want from a machine; it makes a strong, visible statement that what you want is an operating system. On these terms, it is excellent. Many people, though, do not want an operating system at all; they simply want the machine to do a particular job. They do not care about files or directories or processes or I/O redirection. These people form the majority of the computer-using population, and may just want to edit and enter text, and get it formatted nicely; or they just want to receive and send mail; or they just want to create and run their reactor-design or weather-modeling codes. If they are sophisticated, they may see, and begrudge, every machine cycle that goes into overhead.

Many users have no interest whatever in forming a correct model of what is really happening, and using it to predict the behavior of the machine when they try new things. Furthermore, a substantial fraction of them are entirely justified in not bothering even to learn what an operating system is, let alone how to use it creatively. In other words, the underlying simplicity of the concepts underlying the operating system is irrelevant to many of its users, because these concepts are unrelated in any direct way to their desires and needs.

Tool Using

This begins to touch on the second great virtue of Unix: composition of tools. I spoke above of our approach to text-processing. To write a paper with a complicated textual structure, one must deal with a great many programs: an editor, .he basic formatter, and some subset of the more specialized tools: one for equations, another for tables, another for graphs, another for references; and there is a whole host of ancillary services available, like spelling and grammar checkers, double-word finders, and so forth. The approach is certainly more forbidding and complex than a single, integrated word-processing program like the ones that can be bought for a microcomputer, because there is more freedom: you get to build things for yourself.

It is easy to argue that the several pipelined programs take more CPU time than they would if they were jammed together. Moreover, there are inherent limitations in pipelined text-processing programs, because the pipeline is one-directional, without feedback. For example, the programs setting tables and equations have no idea where their result will land on the page, which leads to imperfect placement; similarly, the equation-setter suffers from lack of knowledge of the properties of the font it is using.

Implications of Portability

Unix is portable in two senses. First, in the computer-specific way: it can be moved, without great difficulty, to a wide variety of machines. Second, it has been transported to, and through, a great many different groups of people; that is it has had many contributors.

That it had many contributors–the Bell Labs research group where it was invented, the development group that has become a division of AT&T Information Systems, and UC Berkeley–is historical fact. It was born as a research effort, not as a product, and was not described as a whole until recently, in even an informal specification: it grew over a period of years. It is regrettable, but was probably inevitable, that it split into somewhat differing branches, of which the most notable might be called the System V family and the BSD family.

When development of anything is undertaken by separate organizations, with differing purposes, the products of their development will diverge, as surely as evolution creates new species when similar populations become separated. In retrospect, I can see how certain of the most annoying differences–such as the terminal control specifications–might have been prevented by timely efforts. Other differences between System V and BSD, for example the approach to networking, could not have been prevented, because they developed independently and in different environments. And there is a whole host of small variations in command behavior, and bits of the software like include files, that differ for no important reason; to continue the evolution analogy, they illustrate random genetic drift.

Unfortunately, software portability itself is, for someone who is trying to sell hardware or whole systems, rather an annoyance. To succeed in selling something, a manufacturer needs product differentiation: people should to be able to tell the product from the competitor's. This is true even if the selling is noncommercial, as in the academic world. It is valuable to distinguish your ideas from those of others.

So to a manufacturer, there is a tension between providing something unique, and still remaining standard. Some, like Pyramid, have chosen an astonishing tactic: a ''dual universe'' machine that will run both System V and BSD programs. Others offer extensions with plausible explanations (or rationalizations) for the departure from the standard–and, of course, if there is no written standard, as there has not been until very recently, the departures are easier to rationalize.

The two-edged nature of software portability is most clearly evident in AT&T's efforts. Our company developed the Unix system, and owns it: it is a proprietary product. However, as history worked out, its market was created by distribution of low-cost software licenses, beginning well before we were selling hardware. Now that we are a hardware supplier, we find that we are, in effect, competing against our own software product.

Conclusion

To put matters rudely, what I have said is:

> Unix is simple and coherent, but it takes a genius (or at any rate, a programmer) to understand and appreciate the simplicity.
>
> The tool-using approach is powerful and intellectually economical, but it takes imagination to use. It may also be more costly to combine simpler, more general tools than to build a more specialized one.
>
> Software portability is socially valuable, in some ways is economically valuable, and Unix achieves it astonishingly well. But for big systems, perfect portability is practically impossible to realize, mostly for reasons that have nothing to do with differences between hardware.

All these contradictions exist, and should be faced. Rather than attempt a grand synthesis, or make strong suggestions, I will content myself with some small observations. Portability first: although it is unlikely that the differences between the various versions of the system will disappear, the trend does seem more towards convergence than differentiation. Standardization groups like the IEEE P1003 committee should at least expose the issues, and at best may create a politically neutral center towards which Summit, Berkeley and perhaps now Pittsburgh can gravitate. (At worst, they will create yet another variant version.) Moreover, even some of the economically active players in the market seem to be forming technical alliances intended to alleviate variation.

The degree to which Unix can continue to retain its original simplicity and coherence of design is less certain. There is a real struggle between those who would like to make it the operating system for the masses and those who like it the way it is. (A quick test on this issue: how do you feel about *''rm *''*? Is it too dangerous, or do you accept the logic that makes its meaning inevitable?) There is a related, but distinct struggle between those who find it necessary to add features and those who strive to generalize and extend it in ways consistent with its design.

Most other operating systems are bland; opinions on them run the gamut from ''ugh'' to ''it's OK.'' At best, they provide a neutral background in which to work, while Unix makes a statement about how to program; it has a style. Correspondingly, it attracts strong fans and vocal critics. It is the only system I know of about which people publish papers arguing that adding features is wrong. It has true believers: people who argue that the more options a command has, the less its author has thought about what it should really do.

The existence of such people is the final endearing thing about Unix, but one that can be annoying, and even dangerously stultifying. It is good, and it is genuinely interesting, to be involved in a system in which aesthetic judgments have such an important influence on the design. It is also unnerving to the populace to be assaulted by fanatics who assert that theirs is the unique road to salvation. One can be trapped by Unixism as much as by any other philosophy.

References

1. D. M. Ritchie and K. Thompson, *Comm. ACM* 17 7, (July 1974).

2. B. W. Kernighan and P. J. Plauger, *Software Tools,* Addison-Wesley, Reading Mass., 1976.

3. J. L. Bentley and B. W. Kernighan, ''GRAP–A Language for Typesetting Graphs,'' Comm. ACM 29 8 (August 1986).

Sun, surf and X in California.

Andrew McRae

Megadata Pty Ltd.
2/37 Waterloo Road
North Ryde

andrew@megadata.mega.oz

1. Introduction.

Xhibition 89 is scheduled for June 25th through to June 28th in San Jose, California. The conference technical program covers a broad range of topics, with keynote addresses from both Unix International and OSF executives. The program is backed up with a series of tutorial sessions aimed at a range of X users, from the novice X programmer to developers using X for complex graphic systems. An extensive vendor exhibition area provides attendees the opportunity to view the latest in display hardware and X applications.

In a related area, a major happening in the Unix world is the specification of a standard 'look and feel' graphical interface, which has been embodied as the OPEN LOOK user interface. XView is an standard toolkit being developed by Sun Microsystems conforming to OPEN LOOK, using X11 as the underlying graphics system.

I am intending to report on the conference, tutorials and exhibition. In addition I am visiting Sun Microsystems where I hope to obtain a first hand look at XView and associated development.

2. Xhibition 89

The X Window System stands out from the current crop of double standards (OSI vs TCP/IP, NFS vs RFS, OSF vs Unix International etc. ad nauseum) as one which is universally supported and accepted by virtually all vendors. The only serious challenger that appeared was Display Postscript, and similar genre such Sun's Network Window System (NeWS). The old adage 'that if you can't beat them, join them' comes to mind when we view the current directions of a merged X11/NeWS server, and a user interface based on X. In seems that Display Postscript is being strategically repositioned so that it can be perceived as an 'application engine', or 'imaging machine', which runs on top of or beside X, thereby resolving the direct conflict. It may be that such a compromise will bring out the best in both systems. Certainly the fact that Display Postscript being discussed at an X conference indicates that it is likely to survive in the X world.

The Xhibition itself is broken into three sections; a tutorial section, the technical program and the exhibition. The tutorials are centred around more basic areas of learning such as X Programming, Fundamentals of Graphics, Use of Colour etc. I'll summarize each of the tutorials briefly in the final paper.

The conference proper contains up to four different threads at any one time, thereby making it extremely difficult to cover all the areas that are of interest. The presentations naturally are more wide ranging and sophisticated than the tutorial program. Generally the presentations are related to the current strategic directions being taken by major players in the X game, as well as extensions to X, and Application programming issues. Porting the X server and X applications is also an area addressed by a number of papaers.

Richard Stallman from the Free Software Foundation will be presenting a paper on Look and Feel Issues, which will be interesting in the light of the current controversies in which he is embroiled. Jim Fulton from the X Consortium will be giving a paper on X Terminals (where do you draw the line between an X terminal and a discless workstation?). A colleague of mine is also attending the conference to cover other threads which I will then be able to report on.

The NeWS window system: A look under the hood

James Gosling

Introduction

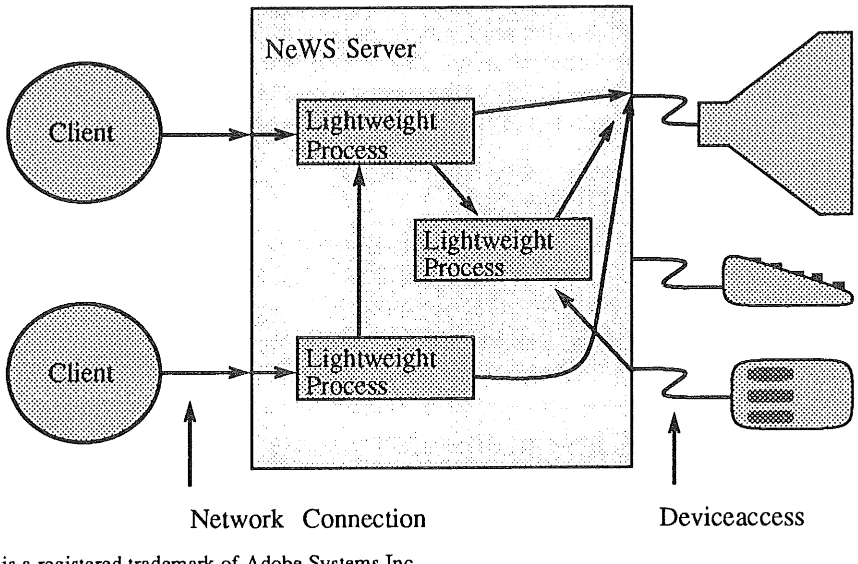
With the standards furor over the X11 window system, many people ask "why is Sun bothering with NeWS?". There are two answers: one is that PostScript† is an important standard too, although its importance is concentrated outside of the traditional workstation market. The other is that there are a number of important technical advantages to NeWS.

This paper starts with a brief overview of NeWS, so that those unfamiliar with it can understand the rest of the paper. This is followed by a section on living with PostScript that covers the use of object-oriented programming and rapid prototyping. Then there is a discussion of the performance implications of downloading programs, followed by a discussion of the advantages of having a close correspondence between the capabilities of a screen and printers. Finally, there is a discussion of one of the many ways that PostScript can be used to enhance X11: fonts.

1. The Structure of NeWS

The NeWS server resembles a self-contained operating system. It contains a set of concurrent processes that are independently and cooperatively executing. The server communicates with application programs through a variety of network connections, and with devices through the usual Unix device driver interface. The processes act as intermediaries between applications and devices, implementing the various tasks that go toward driving the user interface

The code that is executing in these processes is dynamically loaded into the server either from the file system or from a client process through a network connection. It is all written in an extended version of the PostScript language. The server is missing many of the features found in other window systems (windows, for example). These are implemented in PostScript, on top of the mechanisms

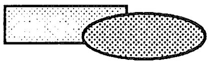


† PostScript is a registered trademark of Adobe Systems Inc.

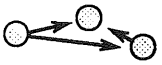
Sun Microsystems, Inc 2550 Garcia Avenue Mountain View, CA 94043 (415) 960-1300

provided by the server. This extensibility is not something that was added on, it is the central theme. The extension facilities are clean and protected, they provide mechanism, but not policy.

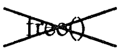
The PostScript language was originally designed for driving printers. For it to be useful in a window system, we needed to add a few extensions:



Multiple drawing surfaces (canvases.) In PostScript, there is only one drawing surface: the printed page. NeWS creates the illusion of multiple drawing surfaces. These can be composed in hierarchies, reshaped, and moved around. They are the basic building block out of which windows are built.

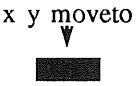


Lightweight processes. These are processes that are cheap to create, use relatively little storage, and share the same address space. The processes within NeWS handle requests from applications, deal with input translation, handle animations like menu highlighting, and perform an assortment of other tasks.



Garbage collection. NeWS has a true garbage collector, rather than the simple heap pointer reset in printer PostScript. PostScript code doesn't have to worry about freeing unused objects or placing gsave/grestore pairs appropriately: objects that are no longer referenced disappear automatically.

Input events. Printers only have to deal with output, while a window system has to deal with input as well. NeWS defines *events* which are messages that get sent from process to process. The keyboard an mouse appear as fictitious external processes that generate events when keys or buttons go up or down, and when the mouse moves.

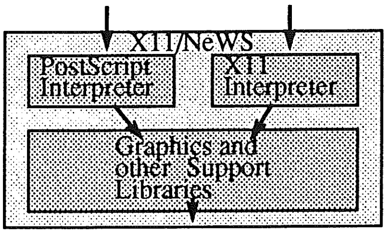


Encoded input stream. In the original PostScript specification, programs could only be written as strings of 7-bit ASCII text. This has significant negative performance implications. NeWS defines a compressed, pre-compiled, representation for PostScript code that is much more efficient to read.

One of the major attractions of PostScript is its approach to device independence. There is a lot of variability amongst displays: B/W bitmap, greyscale, color and an assortment of resolutions. We wanted to avoid a "lowest common denominator" approach. The PostScript imaging model doesn't say anything about pixels, it's a resolution independent description of the image. It is at a high enough level of abstraction to be able to make use of high performance displays.

There are two kinds of device independence: In the first, the server defines a high-level model that attempts to cover many different hardware characteristics. Under this kind of model, user programs are generally oblivious to the hardware characteristics, but they can inquire and adapt if they choose. In the second kind of device independence, the server defines many low-level models, providing different ones for different kinds of hardware. This is simple to implement, but it places a greater burden on application programs and it restricts the applicability of exotic hardware.

NeWS has been merged with X11, so that applications written for either window system can be run concurrently. This easiest way to understand how this works is to think of X11 as an interpreter for a simple language: the X11 protocol. The NeWS server was restructured to allow multiple interpreters to reside within it. They all sit on top of the same set of graphics and support libraries.



2. Classes and Object-Oriented Programming

PostScript is one of the world's worst programming languages. It was designed as a language in which programs are written by other programs, not by people. To a large extent, this problem has been dealt with in NeWS by adding facilities for object-oriented programming, which provide an easy way to define an manipulate objects in PostScript

The easiest way to understand this is to walk through a simple example:

```
/ConsoleWatchBag AbsoluteBag []
      classbegin
            /PaintCanvas { ... } def
      classend def
```

This defines a class called ConsoleWatchBag to be a subclass of a class called AbsoluteBag. This defines one method: /PaintCanvas (a method is a procedure that "belongs" to a class). ConsoleWatchBag behaves exactly like AbsoluteBag, except that it has this one different procedure. ConsoleWatchBag is a *subclass* of AbsoluteBag.

```
/win  [ConsoleWatchBag] framebuffer
      /newdefault ClassNoticeFrame
      send def
```

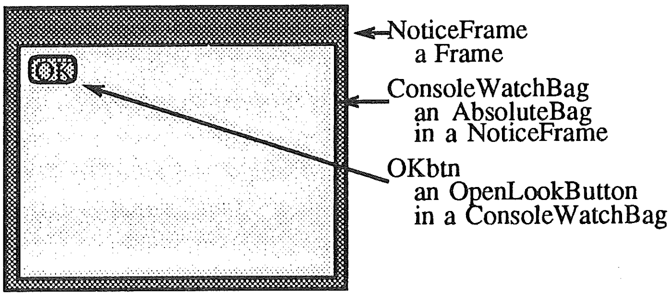
This rather opaque piece of code creates a new instance of ClassNoticeFrame (`/newdefault ClassNoticeFrame send`). This instance creation has a couple of parameters (`[ConsoleWatchBag] framebuffer`). A notice frame is a window that has a client pane within it: in this case, a ConsoleWatchBag.

```
/mybag /client win send def

/OKBtn [ 10 10
      (OK) { pop /unmap win send }
      OpenLookButton
] /addclient mybag send
```

Finally, we create an open-look button labeled OK that will unmap the window when it is pressed.

Here's what this looks like on the screen:



This example has defined a new kind of AbsoluteBag (a canvas that can be drawn on), that has a special method for painting its contents. Then we created a window that contained one of these, and we created and installed a button in it. This is essentially the complete implementation of a Macintosh-style confirmer.

The class paradigm followed in NeWS follows very closely that of Smalltalk. We have made a couple of extensions. Chief among them are multiple inheritance and method promotion. In multiple inheritance, a class can be a subclass of more than one class. This effectively lets you mix together the behaviors of several classes in one class. In method promotion, the definition of a method can be moved dynamically up and down the class hierarchy. For example, if we had a method that computed the area of instances of a class, for any specific instance of that class (an object) we could promote that procedure to be one that simply returned a constant, rather than performing a computation based on constant values.

PostScript turns out to be a very nice language in which to implement object oriented programming. All of the facilities present in NeWS can be implemented in pure PostScript. An instance is just a dictionary, which contains a reference to its class. A class is just a dictionary that contains an array of its superclasses. Entering a class instance (sending a method) just pushes the object, it's class, and its super classes onto the dictionary stack.

3. Classes and Rapid Prototyping

Since PostScript is an interpreted language code can be developed interactively and incrementally.

There is a utility called *psh* (the PostScript shell) that allows developers to interact directly with the PostScript interpreter, having code executed as it's typed, with the results visible immediately. When this is coupled with the class-based approach, for defining objects, new kinds of objects can be built easily based on others. This encourages an experimental approach that allows interfaces to be rapidly built up out of standard components.

There is a rich standard set of objects to start from. This includes blank drawing surfaces, menus, pop-ups, scrollbars, buttons, button stacks, dials, sliders, labels, and text items. There are many variations of each available.

Many facets of the application and its interface can be separated, allowing a range of changes to the interface which don't affect the application. By doing this outside of the application, applications written in any language can take advantage of it.

4. Printer Correspondence: making applications easier

For many applications, PostScript language compatibility isn't nearly as important as compatibility with the PostScript imaging model. They need to be able to draw a string on the screen and *know* that it will match the behavior of the printer. This is necessary for applications that want to implement WYSIWYG (what you see is what you get).

Applications written for window systems other than NeWS have to go through a rather laborious process of calculating what the printer would do, calculating what the window system will do, comparing the two, and compensating. NeWS removes the burden of ensuring that text on the screen matches text on the page. This lets application writers concentrate on the task at hand, rather than on how to reconcile the printer and the display.

Another source of problems is the power of the imaging model. Printers today, primarily PostScript printers, have a very sophisticated imaging model. They can scale and rotate arbitrary objects, including text and images. Many applications put a lot of effort into graphical rendering because the underlying system isn't powerful enough. By supporting the full PostScript model, NeWS applications avoid this.

Above and beyond the imaging model provided my most window systems, NeWS provides:

- Curves. In conventional systems, the boundaries of polygons are restricted to being straight lines. PostScript allows them to include arcs of ellipses and cubic Beziers.
- Image manipulation. PostScript can apply arbitrary transformations to images and can convert them from one form to another. For example, a full color image can be scaled by 75%, rotated 45 degrees, and dithered or halftoned to black and white.
- Arbitrary transformations. Transformations can be applied to *anything*. This includes simple graphics objects like vectors and circles, but also includes text and images.

As a side-effect of having PostScript in the window system it is easy to drive printers. A printed page is just a large monochrome image. It's a trivial matter to get NeWS to create such a large monochrome image, render a page to it, and write that page out. There are a number of printers on the market that can connect to a SCSI port and will print monochrome bitmap images sent to them. It is a "simple matter of programming" to write a program to transfer these images out to the SCSI port. This is made particularly easy by a new NeWS facility that allows an image to be in memory shared between an application and the server. Using this technique for printing guarantees WYSIWYG.

5. NeWS helping X11: fonts

NeWS has a very large standard font library:

AvantGarde-Book
AvantGarde-BookOblique
AvantGarde-Demi
AvantGarde-DemiOblique
Bembo-Bold
Bembo-BoldItalic
Bembo-Italic
Bembo
Bookman-Demi

Lucida-Bright
Lucida-BrightDemiBold
Lucida-BrightDemiBoldItalic
Lucida-BrightItalic
LucidaSans-Bold
LucidaSans-BoldItalic
LucidaSans-Italic
LucidaSans
LucidaSansTypewriter-Bold

Bookman-DemiItalic
Bookman-Light
Bookman-LightItalic
Courier-Bold
Courier-BoldOblique
Courier-Oblique
Courier
GillSans-Bold
GillSans-BoldItalic
GillSans-Italic
GillSans
Helvetica-Bold
Helvetica-BoldOblique
Helvetica-Narrow-Bold
Helvetica-Narrow-BoldOblique
Helvetica-Narrow-Oblique
Helvetica-Narrow
Helvetica-Oblique
Helvetica
LucidaSansTypewriter
NewCenturySchlbk-Bold
NewCenturySchlbk-BoldItalic
NewCenturySchlbk-Italic
NewCenturySchlbk-Roman
Palatino-Bold
Palatino-BoldItalic
Palatino-Italic
Palatino-Roman
Rockwell-Bold
Rockwell-BoldItalic
Rockwell-Italic
Rockwell
Symbol
Times-Bold
Times-BoldItalic
Times-Italic
Times-Roman
ZapfChancery-MediumItalic
ZapfDingbats

From PostScript, these fonts can all be scaled and rotated arbitrarily. In the X11/NeWS merge, support for these sophisticated fonts "leaks over" into X11. All NeWS fonts, including Folio and user-defined PostScript fonts are available to X applications as ordinary X fonts under X11/NeWS. We didn't have to extend X in any way, although we did have to take a few liberties with the definition of font names.

There is an "underground" X font name standard that we follow: "name*size*", where the size is a string of digits that trail the name. For example, "screen10" has a name of "screen" and a size of "10". When we execute the X open font request, we effectively execute:

```
/name findfont size scalefont
```

The font is looked up using the standard PostScript findfont operator and is scaled using the PostScript operator.

We also support the Logical Font Description (LFD) proposed X11 extension. According to this proposal, font names are long strings that can be decomposed into many fields. Here is an example of one:

```
/-adobe-helvetica-medium-o-normal--14-140-75-75-p-78-iso8859-1
```

We parse this name and calculate the font family, what encoding to use, and how it should be scaled. It effectively becomes the PostScript sequence:

```
/Helvetica-Oblique findfont
[14*75/72 0 0 14*75/72 0 0] makefont
/iso8850-1 encodefont
```

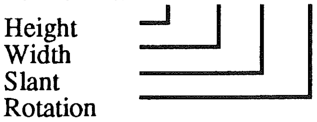
These fonts aggravate a number of generic X11 problems that applications run into.

- Ascent/Descent. X11 fonts have a pair of fields called their ascent and descent. According to the protocol specification, it is legal for characters to be taller than the ascent, or to extend lower than the decent. With many fonts, this actually happens: Applications want the ascent and descent fields to be tight, so that lines aren't too far apart. But the accents on upper case characters can extend rather high (as in the Icelandic spelling of Iceland: Ísland). The X ImageText primitive, which most terminal emulators use, depends on characters fitting within the ascent/descent of the font. This makes internationalization awkward.
- X11 requires that the width of a character be an integer number of pixels, and that it only has an *x* component. This makes it impossible to match the printer exactly, since when character widths are scaled, they are unlikely to be exact integers. We round character widths to integers for X, but not for NeWS, unless a NeWS application explicitly asks for it. This also makes rotated text impossible

- Another problem is with XListFonts. This X request is supposed to return to the application a list of all the fonts available. With scalable fonts, this list is infinitely long. We compromise by replying with a list of the fonts that have and-tuned bitmaps, and with LFD names with the size fields set to zero. This use of zero to indicate scalability is currently a “proposed extension to the proposed extension”.

There are a number of extensions to the X text model that we would like to propose and implement. Based on the mechanisms available through PostScript, these are trivial to implement. We could overload the *size* field of an LFD name to indicate both x and y scaling, skew, and rotation. With the exception of rotation, these fit into X11's existing text model. To implement rotation we'd either have to return useless width values, or define a new request that returned *x* and *y* values with subpixel precision.

/-adobe-helvetica-medium-o-normal--14x16s12r45-140-75-75-p-78-iso8859-1



It turns out that we can actually do all of these in X11/NeWS today by defining new fonts from the PostScript side. For example, if we wanted to create a version of Times-Roman that is rotated 90 degrees, we could execute the following PostScript code:

```
FontDirectory
  /Times-Roman findfont
  [ 0 1 -1 0 0 0 ] makefont
/Times-Rotated put
```

This looks up the Times-Roman font, transforms it by using the makefont operator, and then installs it into the font directory. This new font, Times-Rotated, now appears like any other X font. Text drawn with it will be rotated 90 degrees. If the X application inquires the widths of the characters, it will get 0 for all of them, since their *x* widths are 0. The *y* components are non-zero, but there's no way to report that to an X application.

Adaptable Text Interfaces *

Michael Paddon
mwp@munnari.oz.au
University of Melbourne

David Spaziani
davids@munnari.oz.au
University of Melbourne

July 9, 1989

1 Introduction

1.1 Motivations

In a modern computing environment, an application is required to deal with an ever widening range of terminals and workstations. This is not a new problem. In the past, terminal dependencies have been isolated in terminal description files of one type or another and libraries, such as *curses*, have allowed the programmer to use a virtual high-level terminal device.

With the advent of high resolution bitmap displays and the common availability of alternative input devices, the concepts that served so well in a terminal environment fall short. While it is still possible to emulate a terminal on today's hardware, users expect much more from interfaces running on a workstation. This leads to the development of new paradigms and systems; *X11*[1] and *NeWS*[2] dominate in this area in the *UNIX*[3] environment.

Despite the obvious advantages of workstations, however, it is costly to put one on everybody's desk. As a result, many applications are increasingly expected to provide the full featured functionality possible with bitmap displays and yet also support the relatively dumb terminals that will be common for many years to come.

From a software engineering viewpoint, maintaining multiple versions of an application targeted toward the different requirements of both ends of the interface hardware spectrum is not cost-effective. The ongoing *Titan*[4] database project at the University of Melbourne has had to address these issues in the design of its form and text-browsing interfaces.

*This research is supported by a *GIRD* grant and the University of Melbourne.

[1] X11 is a trademark of MIT.

[2] NeWS is a trademark of Sun Microsystems Inc.

[3] UNIX is a registered trademark of AT&T.

[4] Titan is a trademark of the University of Melbourne.

1.2 Addressing the Problem

Before attempting to develop a device independent solution to the problems outlined above, we must narrow our focus slightly. It is obvious that a bitmap display has a much greater range of capability than a dumb terminal; detailed line drawings are difficult or impossible to approximate on older hardware. However, users don't necessarily want to use these abilities.

The majority of applications (and this is particularly true under the UNIX operating system) deal exclusively with textual data, and, not surprisingly, all displays can efficiently handle information in this form. For many users, the advantage of a bitmap display is that they can run all their favourite programs simultaneously, side by side, and with facilities such as cut and paste between windows. Such an environment simplifies job control and allows rapid shifting of attention from task to task for the user.

Users being users, they soon notice that software written exclusively for their workstation has a really neat interface; menus, buttons, scrollbars. It is natural, then, to demand that all software becomes as easy to use when the facilities are available; but never, of course, at a cost to backward compatibility.

Our solution to this problem focuses on text applications, combined with the sort of high-level interface controls commonly found in workstation environments.

1.3 Adaptable Text Interfaces

We have designed a toolkit which lies above the layers of device specific software, such as *curses*, *NeWS*, *X11*, or any other low level interface to the underlying hardware. It presents a uniform interface model for applications. This software, known as *ATI* (Adaptable Text Interfaces), is totally device independent and is structured so as to always provide a consistent paradigm. See Figure 1 for the relationships between ATI, application and display library software.

ATI does not, in any way, specify the "look and feel" of the user interface. This decision is left to the implementation and, in most cases, to the lower layers of software utilised by the toolkit.

2 Device Independence Through Abstraction

The key to building an interface toolkit that is device independent across the whole spectrum of output devices, from terminal to bitmap, is ***abstraction***. More specifically, ATI hides hardware specific information by defining a group of generic types and objects which the application program manipulates.

Let us pause a moment to define some terms. We call any abstraction which stores but does not directly represent displayed information a *type*. An *object*, on the other hand, abstracts output data. Both types and objects are opaque

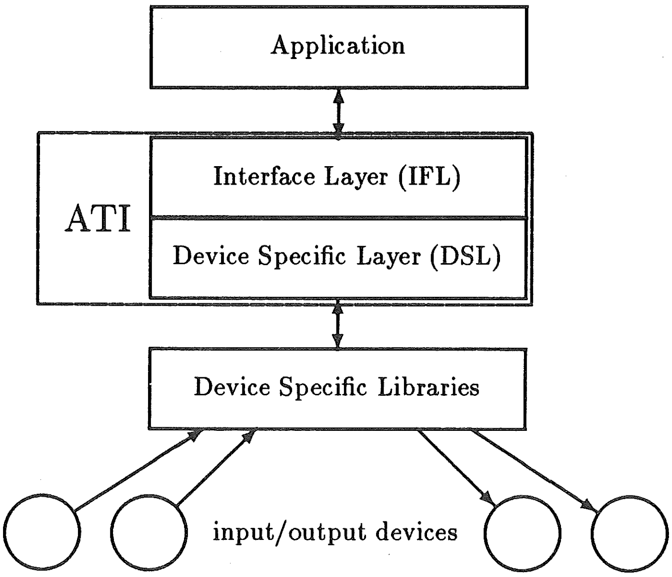


Figure 1: ATI layered structure overview

in that the application needs no knowledge of the implementation details, only their functionality.

This section describes all the important abstractions used by ATI. We focus on the services provided by the types and objects, and the interface they present to an application. We also examine the most important aspects of how these abstractions are actually represented on different hardware; usually concentrating on the extreme ends of the technology scale (dumb terminals and high-resolution colour bitmaps) on the assumption that all devices fit a niche somewhere in this range. Please note that such details are hints only — the exact way in which objects interact with the user are dependent on the particular implementation of ATI.

Implementation details will seldom be mentioned. The entire purpose of ATI is that such issues are of no importance to the application programmer.

2.1 Types

ATI provides three basic types to describe output oriented information:

- colours
- fonts
- strings

2.1.1 Colours

All *colours* are specified and stored to 24-bit resolution. Different hardware takes advantage of this information in diverse ways; typical examples follow:

full colour bitmap
This hardware will accurately reproduce any colour.

colour mapped bitmap
This hardware can only reproduce a limited number of colours at a time (most commonly 256, ie. 8-bit resolution). ATI will attempt to remove unused entries from the colour map to make room for the requested colour. If this is not possible for any reason, then the closest match will be selected from the table.

monochrome bitmap
ATI will use dithering to approximate the requested colour.

terminal
This hardware supports text-only output in either black or white. Normally, we will ignore colour information when using such a display and output all text regardless. As a general rule of thumb, ATI applications may assume that they normally output black text on a white background.

Therefore, if reverse video facilities are available, text closer to the white end of the spectrum is output in that style. Similarly, if the terminal supports half-intensity, some colours of text may output using that feature.

Defining an abstract representation for colour is straightforward. In fact, two methods already exist and are in wide use — *RGB* (red,green,blue) and *HSB* (hue,saturation,brightness). ATI uses the former scheme.

2.1.2 Fonts

A *font* describes the style of type used to output a character. Describing fonts in a generic way is a difficult task, due to the wide variety of methods in common use.

ATI requires three pieces of information to uniquely identify a font:

font family
The generic name of the font; for example, "Times", "Helvetica" or "Courier".

font face
The typeface in which the font is rendered; one of "normal", "italic", "bold", "underline", "outline" or "shadow".

font size
The size in which the font is rendered, measured in centi-points (see Section 2.2.1 for further explanation of this unit).

Choosing a local font family that is equivalent to the requested family (or at least the most similar available) is a difficult task. Our solution is to define a mapping of generic font family names to local equivalents in every implementation of ATI. Note that there is always an entry in the mapping table for a font family called "default"; used as a last resort if all other mappings fail. The algorithm in Figure 2 shows, in detail, the process of local font selection.

After a suitable local font family has been located, the toolkit selects the variation that most closely matches the requested face and size.

This scheme gives the maximum flexibility and portability. Most applications will only use generic font family names, but if special effects are required, then local facilities may be used without compromising generality.

Low end hardware has only a few fonts, perhaps as few as one. This does not affect the mapping algorithm; in the worst case all text is printed in the one font.

2.1.3 Strings

In general, text consists of an arbitrary number of characters in a particular sequence. Each character belongs to a particular font and has a specific colour. A *string* stores information of this nature.

```
found = false;
for (all generic font families in mapping table)
    if (generic_family == requested_family)
        local_family = map_generic_to_local (requested_family);
        found = true;
        break;

if (! found)
    for (all local font families)
        if (local_family == requested_family)
            found = true;
            break;

if (! found)
    local_family = map_generic_to_local (‘‘default’’);

use_family (local_family);
```

Figure 2: Selecting a Font Family

For implementation purposes, two factors aside from functionality must be taken into account. Firstly, it is desirable for the character information in strings to be easily and efficiently accessed from the C programming language. Secondly, storing attribute records for each character is extremely wasteful of memory resources for most text intensive applications.

We represent a string as shown in Figure 3; a sequence of bytes, delimited by a *null* (which is, in fact, what C expects of a string) and a list of *text styles*. A text style is a record containing:

offset
The offset of the first byte affected by this record.

length
The number of contiguous bytes affected by this record.

colour
The colour of the bytes affected by this record.

font
The font used to display the bytes affected by this record.

Thus, text which contains runs of characters with the same font and colour takes up minimal resources for attribute storage.

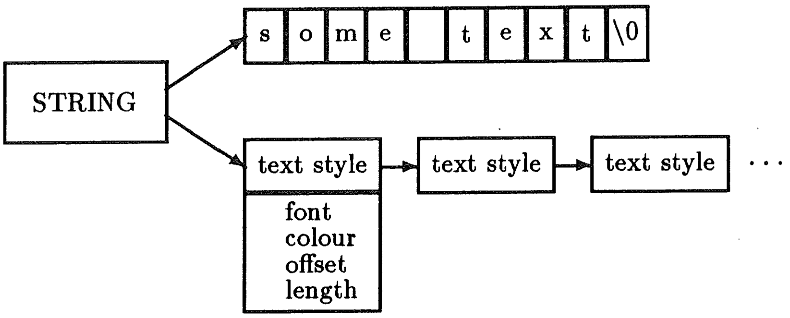


Figure 3: Representation of a string

ATI provides some string manipulation functions such as copying, concatenation and extraction of substrings. Simpler manipulation may be performed by standard C library routines.

For the remainder of this document, references to "string" refer to this abstract type, unless specifically stated otherwise.

2.2 Objects

The overriding design criterion for objects is that their facilities degenerate into usable forms on low-end devices. Their functionality must be easily and naturally expressible on dumb terminals and, nonetheless, also utilize the power of more capable hardware.

The three basic objects classes are:

- Windows
- Menus
- Items

The class *item* subsumes all objects which can (and, in general, must) exist inside a window. The currently defined items are:

- Buttons
- Selections
- Text boxes
- Fields

- Icons

All item objects share the same top level interface, with the subclass itself defining functionality. This makes it easy to add new item classes to ATI in a controlled fashion as they become necessary.

2.2.1 Windows

An ATI application performs all user interaction via windows.

Each window is a logically separate item space. The location of each item in a window is specified in terms of that window's coordinate system; origin at top left hand corner, with x values increasing to the right and y values increasing downwards.

The basic unit of measurement used within ATI is the centi-point, (ie. 1/100 of a point, or 1/7200 of an inch). This value was chosen for three reasons:

1. An absolute system of measurement aids portability and device independence.
2. The point is used as a fundamental unit due to the text based nature of the toolkit.
3. The positional accuracy of all measurements is far greater than the unaided eye can detect, and far beyond the resolution of most devices.

Figure 4 summarizes the available window attributes:

item list
The set of items that exist in the window space.

message strings
There are three areas for application defined messages.

scroll flags
The item space represented by a window may be larger than then window's physical area. If the appropriate scroll flags are set, mechanisms are activated which allow the user to pan a window either horizontally, vertically or both. Such mechanisms (generically known as scrollbars) work in a way analogous to button items (see Section 2.2.3). On bitmap displays, an actual scrollbar is displayed showing roughly where the current view is in the window space.

icon
A window may have an icon associated with it (see Section 2.2.7 for a definition of icons). If so, the window may be closed into a compact representation. Note that an icon is an item; therefore, it may exist in the space of another window. This permits a pseudo-hierarchy of windows.

menu

A window may have an application defined menu associated with it (see Section 2.2.2 for a definition of menus).

background colour

The background colour of the window.

A window maintains a list of items; each of which has a location relative to the top left corner of the bounding box of the item. An item's size is determined by itself and depends on the particular ATI implementation; the window has no knowledge of this information. This raises a problem. If a window layout is moved between devices, items may overlap visually if their sizes change radically. The easiest way to solve such problems is to ignore them. All items may be dynamically moved around their window by the user, allowing layouts to be adjusted as needed. ATI provides code for an application to portably store a window layout (which items, their location and any other item-specific data) in a file for later use.

As mentioned, each window has three message strings associated with it:

title

Description of window, usually displayed at top and centre of window. Information of a permanent nature should be placed here.

status

Temporary status messages, usually displayed at bottom left corner of window.

mode

Some indication of application's mode, usually displayed at bottom right corner of window.

These message strings are displayed outside the window's item space and are, therefore, always visible.

A window abstraction implies and requires the concept of a *cursor* to use as a selection mechanism and keyboard focus arbiter (where multiple windows are visible at one time). On a terminal, the screen cursor suffices; manipulated by arrow keys or control keys. A bitmap screen's cursor is usually controlled by a device such as a mouse or trackball, with one of more buttons; such differences are easily and transparently hidden.

On a terminal, only one window may ever be visible at a time. There is a global menu (by title) of windows that the user may call up at any time that allows a new one to be displayed. Bitmap displays allow more than one window to visible at any instant; these may overlap. The position and size of a window is fixed on a terminal; on a bitmap it is completely under control of the user. The application has no need of such information.

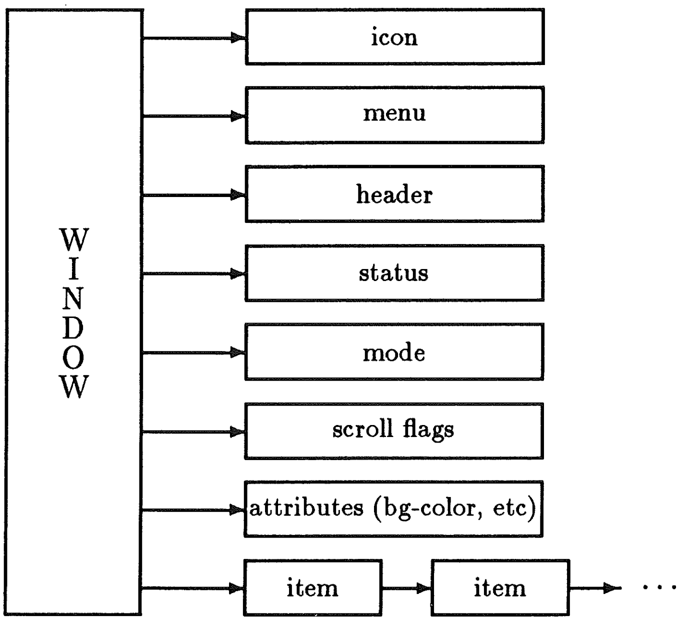


Figure 4: Structure of a window

2.2.2 Menus

A *menu* is simply a list of options that the user may select. Each menu has a name and a list of choices, each comprised of:

label
A string describing the choice's functionality.

keystroke sequence
An optional sequence of characters that may be typed at the keyboard to accelerate selection of a choice.

active flag
A choice may be deactivated, so that the user cannot select it.

marked flag
When this flag is set, the item is visibly marked.

handler
A pointer to the function to be executed when this choice is made. The handler function is passed a pointer to its calling object, allowing many objects to share the same action code.

Menus on a bitmap display may either pop up under the cursor or be pulled down from a menu bar, depending on the particular "look and feel" that the implementation espouses. In the former paradigm, the particular menu popped up depends on the context of the cursor at the time of invocation, ie. which window or item the cursor resides in. A choice may be made by pointing with the cursor. Accelerated choices may also be made (at any time) by using the optional keystroke sequence that may be associated with any menu option.

On a terminal, a menu screen is displayed after being invoked by a standard key. Again, cursor context determines the specific menu. Otherwise, operation is exactly the same as for the bitmap implementation.

2.2.3 Buttons

A *button* provides a means for the user to initiate a user defined action. Its attributes are:

label
A string describing the button's functionality.

keystroke sequence
A sequence of characters that may be typed at the keyboard to activate the button. Note that, unlike menu choices, this is not an optional attribute.

active flag
A button may be deactivated at request of the application.

handler

A pointer to the function to be executed when the button is "pressed". Semantics such as mutual exclusion between a number of buttons can be provided simply by writing an appropriate handler.

On a terminal, button activation is bound to a specified sequence of keystrokes. Nothing appears in the window at the button's coordinates, but ATI may display a line describing active button key sequences at the bottom of the screen. In any case, the user can request, at any time, a list of buttons and their trigger sequences.

On a bitmap display, a button consists of a region, containing its label, which may be activated by depressing a mouse button while the cursor is inside it. If desired, buttons may still be activated by using the keystroke sequences described above. This gives a user a consistent way of using buttons on all displays, as well as keyboard shortcuts for the experienced user even when bitmap technology is available.

2.2.4 Selections

Selections are a subclass of buttons with the following differences:

- They represent binary selection, ie. they take on a value of true or false. Their value may be set either by the user or by the application.
- They always have a visible representation. On a bitmap they may be displayed as labelled checkboxes; terminal representation is similar, using a special character (eg. '#') as a check.
- Additional semantics such as mutual exclusion can be enforced as for buttons.

2.2.5 Text Boxes

The *text box* is probably the most important item supported by ATI, providing facilities for input and output of textual data. In effect, a text box defines a text space, with certain attributes that affect the layout of lines of text in that space.

Figure 5 summarizes the available attributes:

text lines

The information content of a text box is stored as a sequence of lines. Each line of text is represented as a string and may be of arbitrary length.

text attributes

Text may be left or right adjusted, or centered in the box. Line wrapping may be requested, either at word of character boundaries. The text box itself may be defined as read-only or read-write; this is useful for display only purposes.

maximum number of lines
An application can fix the maximum number of lines allowed within a text box. This is useful for limiting the amount of information a user can input. If this is not specified, then the text box can hold an arbitrary number of lines.

background colour
A text box's background colour may be specified — normally it is white.

menu
A text box may have a menu associated with it.

The text box item implies the existence of a *caret*, showing the current text insertion point. A caret may be either active or inactive, the state of which is set by the application. There can be only one active caret (and, thus, one active text box) in each window. Which caret is active, and where it is positioned in its text box are controlled by user manipulation of the cursor. The exact mechanics depends of the device — clicking the mouse button for a bitmap, or simply moving the cursor to the desired place on a terminal.

An application can read the contents of a text box at any time. Usually, it will wait until the user presses a button or makes a menu selection indicating that processing may occur.

In general, a text box is a rectangular region in a window; the size of which is dynamically adjustable by the user without affecting the application. On both bitmap displays and terminals, text boxes may always be scrolled (using the same mechanisms that windows use). Most device dependencies are already handled by the definition of strings; text is simply represented as faithfully as possible.

ATI implementations on bitmap displays are expected to provide a cut and paste facility between text boxes. It is possible to provide such services on a terminal, but their operation would probably be awkward.

2.2.6 Fields

Fields are a subclass of text boxes. They provide a prompt string as an extra object attribute. The prompt is displayed at the immediate top left of the text box proper.

The field abstraction is useful for form-based applications.

2.2.7 Icons

An *icon* is a compact representation of a window, used to preserve screen real estate whenever the contents of that window are uninteresting to the user.

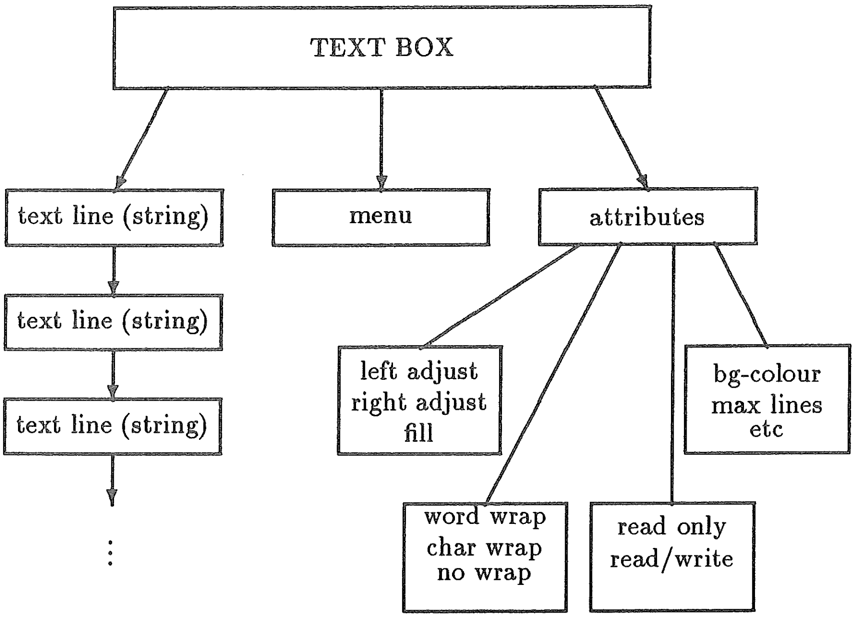


Figure 5: Structure of a text box

An icon has the following attributes:

label
A short string describing the window.

glyph
A short string, usually in a special "glyph" font that provides pictures. If that font is unavailable, then this attribute is ignored.

keystroke sequence
A sequence of keys that may be typed to open the window.

On bitmap displays, icons may be opened by using the cursor and a mouse button. Alternatively the accelerator keyboard sequence may be used. Both these methods are adaptable to terminals.

On a terminal, icons are represented by their label, specially marked with some characters. On bitmap displays, they are square or rectangular regions with room for both label and glyph.

Icons are special in that they are the one item subclass that may exist outside of a window (ie. no associated window specified).

3 Implementation

The prototype ATI implementation is being developed over the NeWS windowing system. At the time this paper was written, many parts of the toolkit had been fully or partially coded and tested, including windows, menus, most items and all types.

It was our experience that a large proportion of ATI could be implemented in a device independent fashion. Therefore, the toolkit has been split into two parts; the *interface layer (IFL)* which provides the application interface to ATI's abstractions, and the *device specific layer (DSL)* which actually communicates with the lower level display software.

The standard NeWS *lite* library did not provide the sort of facilities we required, so we implemented a low level PostScript[5] library, somewhat based on the *OpenLook*[6] "look and feel" to support ATI. This was a non-trivial, but by no means overwhelming task, and gave us some idea of what sort of porting effort would be required given only primitive facilities to build on.

Table 1 shows the current and expected size of the prototype ATI code. Given that there is is good level of existing software support for a device (for example the X toolkit), the effort of porting ATI is reduced to re-writing the device specific layer. As can be seen, this effort involves only a few hundred lines of code; most of which simply massages and passes data between IFL and the underlying layers.

[5] PostScript is a registered trademark of Adobe Systems Inc.

[6] OpenLook is a trademark of AT&T.

Module	*Current Size*	*Expected Size*
	(lines of code)	
IFL	1842	2500
DSL	281	400
PostScript library	1276	1500

Table 1: Current and Expected Implementation Sizes

3.1 Shared Libraries

The ability of having a single application drive a number of devices is limited in use if the user must choose an appropriate binary for his/her display. Ideally, we want the correct DSL code to be loaded dynamically at execution time. This has the dual advantage of allowing a single binary, while minimizing the disk space that the application consumes.

This is made possible by dynamic loading of shared libraries. At execution time, appropriate directives are given to the dynamic loader to choose the desired library. At worst, this may involve using a front-end shell script to set up the correct environment to execute the application. Usually, however, it is sufficient to set an environment variable.

Alternatively, incremental loading is being investigated as an option. In this scenario, the application locates and loads the correct library itself. This is difficult to implement portably.

At worst, separate binaries must be maintained, with a front end script deciding which version is suitable.

3.2 Future Work

Our highest priority task, following the completion of of the prototype implementation, will be to be to write a DSL library for a dumb terminal version of ATI. We expect this to layer over the standard *curses* library, with a moderate amount of effort.

New and modified items will be added to ATI to support abstractions such as sliders (for range setting), numeric fields and pins for menus. A more compact version of selections needs to be developed for cycling through a number of mutually exclusive settings.

We intend to support an image object which understands data in a number of standard bitmap formats. Naturally, this feature will only work on high resolution displays.

4 Conclusion

User interface programming can be a very expensive process in terms of time if appropriate high level tools are not available. Many such libraries exist for specific target displays. ATI is an effort at bridging the gap between all these partial solutions for a specific, narrow domain of application.

With ATI running under NeWS, X11 and curses environments, any text based application that uses our toolkit will be able to move between a wide variety of UNIX machines. This allows interface code to be much more cost-effective simply by removing the problems of maintaining multiple versions of a product.

Strategies for Writing Graphical UNIX† Applications Productively and Portably.

Janet Dobbs

Hewlett-Packard
1000 NE Circle Blvd. Corvallis, Oregon 97330

ABSTRACT

Writing applications with friendly, graphical user interfaces is discussed. A reference model for user interface is given, and productivity/portability issues are reviewed at each reference model level. The X Window System[1] and associated user interface toolkits are covered as the likely basis for creating these applications. Two technologies that are likely to assist in expanding the number of applications with graphical user interfaces for UNIX[2] are outlined.

The ease with which portable software has been developed for the UNIX operating system has long been one of its major benefits to software developers. This process consisted of writing disciplined C code; using libraries such as stdio and libc to insulate programs from the hardware on which they ran. The user interfaces for these applications were based on command line switches and tty's. Programmers were able to concentrate on the *meat* of an application, because little was expected when it came to user interface. This kept programmers productive, and generated a great deal of portable code, but also left UNIX open to its most common criticism: *UNIX is hard to learn and use.*

Today, as the market for UNIX expands, many users are demanding applications that are easier to use. In response to this demand, many vendors selling UNIX systems are providing new technologies to develop applications with graphical user interfaces. Witness the number of vendors providing the X Window System (X) with their platforms. Unfortunately, technologies (such as X) have not yet turned into solutions for end users. One of the reasons for this is that writing applications for environments such as X can be an order of magnitude more difficult than the stdio/libc environment described earlier. Taking the steps necessary to be more productive when writing applications based on X can endanger the portability and vendor independence so cherished in the stdio/libc environment.

This paper will concentrate on strategies for programming in the increasingly complex environments todays UNIX user is demanding. First, a reference model for user interface is presented. Next each level of the model will then be reviewed with respect to portability and productivity. Then, two potential solutions for creating more applications with graphical user interfaces will be explored.

1. A Reference Model for User Interface

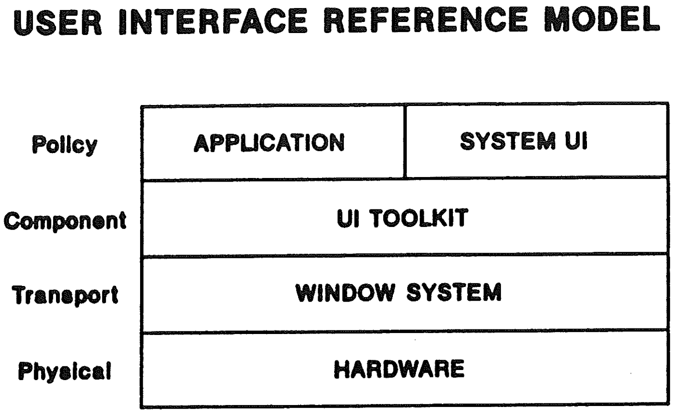
People working in the networking discipline have enjoyed an advantage for years if only because of their ability to reference the ISO OSI model. Granted, the model isn't a perfect one, but it at least gives network researchers a common basis for discussion.

The time has come for a similar model to emerge in the area of window systems and user interfaces. A proposal is depicted in Figure 1.

As with the OSI model, this user interface (UI) reference model contains several layers of functionality, each distinguished by unique services and interfaces. The UI model departs from the OSI

[1] X Window System is a trademark of MIT.

[2] UNIX is a trademark of AT&T



USER INTERFACE REFERENCE MODEL

Policy	APPLICATION	SYSTEM UI
Component	UI TOOLKIT	
Transport	WINDOW SYSTEM	
Physical	HARDWARE	

Figure 1. User Interface Reference Model

networking model, however, in that each of its levels does not necessarily represent a unique layer of abstraction. Rather, the higher one goes in the model, the more strictly enforced the user interface policies are.

1.1 Hardware Layer

The **hardware** (physical) layer of the UI model provides for such input and output devices as displays, keyboards, and mice. Flexibility is a key attribute at this level, inasmuch as voice and gesture recognition hardware may eventually become common options for data input, while live video and sophisticated imaging may become output norms. Without the ability to accommodate these capabilities, we could easily find ourselves cemented in outmoded technology. Consider, for instance, what's resulted from the coupling of the UNIX interface to teletype technology: in a computing era symbolized by icons and pop-up menus, an interface that's both terse and line-oriented stands out as an anachronism.

1.2 The Window System Layer

Up a level from the hardware level of the UI model, in the *window/graphics* (transport) layer, we find the mechanisms (windows) that makes it possible for multiple applications to share the same hardware. This layer also provides the primitives needed for drawing in windows and for receiving window input events. In a nutshell, the window system's role is as a resource manager concerned with a machine's display, color maps, and keyboard. It also is at the window/graphics layer that portability across different hardware implementations is achieved. Three basic rules apply: first, all of a window system's mechanisms should be low-level so as to ensure flexibility; second, all of a window system's mechanisms should operate across machine boundaries (network transparency); and third, no user interface policy decisions should be made at the window/graphics layer.

1.3 The Toolkit Layer

One level higher, in the *toolkit* (component) layer, we find the interactive graphical objects--including menus, scroll bars, and push buttons--that can be used to create a pleasing user interface. Generally speaking, few user interface policy decisions should be *made* at this level, but when the time comes to *implement* policy, the toolkit layer inevitably emerges in an instrumental role. As an example, a toolkit might provide both pull-down and pop-up menus and let a higher layer decide which is appropriate. Notice that several valid toolkit implementations can peacefully coexist so long as side-by-side toolkits rely only on the *window/graphics* layer, they can share a system's display--as well as other resources--without problem.

1.4 System User Interface Layer

Finally, we come to the *system user interface* (policy) layer of the reference model, where we find the mechanisms and utilities for connecting computer users with the applications and system resources they desire. It is here that the policy that determines how a system "looks and feels" is shaped. Iconic desktops, clipboards for moving data between applications, and graphics system configuration tools are among the end user facilities found on this level. It is also on the system user interface level that software developers can find tools and protocols for creating windowed applications that abide by the given user interface policies and can thereby be neatly integrated together. The result is a whole interface structure that's truly greater than the sum of all its assorted parts.

Indeed, for any user interface environment to realize its full promise, it must support and provide functionality at each level of the UI reference model. Programming in the environment should encourage productivity and portability, and the rest of this paper will examine these two issues as they relate to user interface environments for UNIX.

Figure 2 shows how current user interface environments fit onto the reference model.

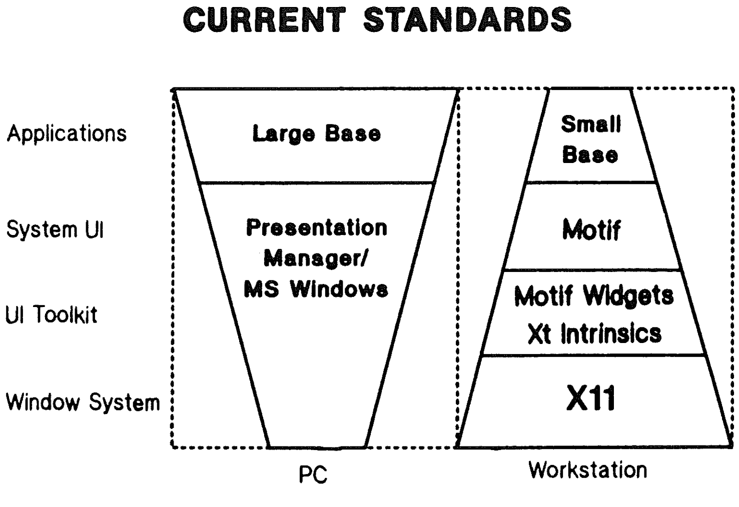


Figure 2. Current Environments

The dotted lines outline areas representing the breadth of functionality software developers require at each level of the user interface reference model. Compare this with the areas enclosed in solid lines, which indicate the fraction of the necessary functionality actually available today in the OS2/MS-DOS[3] based PC domain and the UNIX-based workstation domain.

Notice that on the PC side of the illustration, functionality is shown to be best at the highest level; the lower levels, by contrast, are not nearly so robust, nor are they well separated. The window system primitives at the lowest level don't allow for easy creation of applications with distributed user interfaces, and portability across hardware and operating systems is not supported. The domain of potential applications may be unacceptably restricted as a result, but the PC domain does appear to have reasonable functionality at all levels, with the greatest concentration of offerings being at the level closest to the user.

On the UNIX side of the illustration, meanwhile, X version 11 and the X toolkit provide reasonable functionality at the window and toolkit levels but until recently left a gap at the system user interface level. The separation between UNIX levels is good, and a wide array of applications is provided for, but these advantages did not make up for the lack of suitable range at the highest level. Because of this need for a better user interface for workstations, the first request for technology from Open Software Foundation[4] was for graphical user interface. After consideration, OSF chose Motif--a combination of technologies as shown in the following table.

TABLE 1. OSF/Motif

OSF/Motif	
Component	Accepted Technology
Style Guide	HP/Microsoft Style Guide
Appearance	HP 3D (Bevelled) Appearance
Window Manager	HP Window Manager
Toolkit (Widgets)	DEC XUI/HP CXI *
UI Language	DEC UIL

2. X and the UI Reference Model

The lowest software level in the UI reference model is the one responsible for supplying window and graphic primitives. In the case of X, this is the role of the window system itself, as shown in figure 2. By means of the X protocol and the X binding for C (Xlib) programmers are able to gain access to a robust set of capabilities for managing windows, drawing in them, and receiving those events which occur while an application is running. A detailed description of these features appears in "Xlib--C Language X Interface" [2] and "X Window System Protocol" [3], two documents bundled into the X Version 11, Release 2 distribution. To date, **Xlib** is the only language binding to have been adopted by the X Consortium.

The task at hand is to look at programming the X window system from a portability and productivity standpoint, and determine if it makes sense to program at that level.

2.1 X and Portability

Portability is largely a question of availability. To write applications that are portable, libraries being used must exist on many, if not all, platforms of interest to the programmer. This issue, in the case X,

[3] MS-DOS is a trademark of Microsoft.

[4] OSF, OSF/Motif and Motif are tradmarks of the Open Software Foundation, Inc.

* DEC is a registered trademark of Digital Equipment Corporation.

would appear to have been answered by the large number of companies who have joined the consortium at MIT that supports X, and by those that have announced X products. As of August 1988, the following companies were Members and affiliate members of the X consortium:

X Consortium Members

- Apollo Computer, Inc.
- Apple Computer, Inc.
- AT&T
- Bull
- Calcomp
- Control Data Corp.
- Digital Equipment Corp.
- Fujitsu
- Hewlett-Packard
- IBM
- NCR Corp.
- Sequent Computer
- Sony Corp.
- Sun Microsystems, Inc.
- Tektronix, Inc.
- Wang Labs
- Xerox Corp.

X Consortium Affiliate Members

- Adobe Systems
- Ardent Computer
- Carnegie Mellon Univ.
- Evans & Sutherland
- Integrated Solutions
- Interactive Systems Corp.
- Interactive Development Environments
- Integrated Computer Solutions
- Locus Computing
- Stellar Computer

This impressive list means that most, if not all, vendors in the list will provide Xlib as part of their products. Add X/Open, the Open Software Foundation, ANSI, and other standards bodies endorsing X and portability at the window system level would seem assured.

Several of X's design goals are aimed directly at portability issues. Reference [1] gives these as a few of X's design goals:

- *It should be possible to implement X on nearly any and all bitmap displays, as well as a variety of input devices.* A walk through almost any UNIX-oriented trade show will attest that this objective has already been achieved. Notice, though, that the emphasis is on bitmap displays. X doesn't run on character-cell displays. Higher levels do provide terminal emulators however, to leverage existing terminal based software.
- *X Applications must be device-independent.* It's essential that developers not be required to rewrite, compile, or even relink applications for each new hardware display. No device dependent code is included with X applications, and therefore applications are not tied to a particular piece of hardware. Independence of this sort is as important to software vendors as it is to users. After all, nobody wins when customers are forced to relink in order to use new display hardware.

2.2 X and Productivity

In reference [4] David Rosenthal writes: *Ideally, a window system should provide facilities at three levels. At the lowest of these, clients need the ability to create and lay out windows, to draw in them, and to obtain input events from them. This is the "system call" level of window systems.*

Programming every client at this level would be tedious, so the system should also provide a higher "standard I/O" level consisting of a toolkit of user interface components, such as menus and text panels.

Several of the design goals for X also impact the productivity of those writing programs using only Xlib. They are:

- *X should be capable of supporting different application and window management interfaces.* No single user interface is "best", as evidenced by the fact that different technical communities seem to have adopted radically different ideas about what constitutes an ideal user interface. By not dictating policy, X has a better chance to move along with technology, even as new user interface paradigms grow in popularity.
- *X should provide high-performance, high-quality support for text, 2D graphics, and imaging.* Notice that no user interface tools such as menus or buttons are described in this goal.

These design goals show that at the window system level, X and Xlib were optimized for flexibility rather than ease of programming. Xlib contains hundreds of function calls and parameters and the documentation for it runs hundreds pages. There are no tools that deal with user interface directly, these are left as an exercise for the reader.

2.3 Conclusions at the Window System Level

It would seem obvious from the material presented that writing programs at the Window System level, or Xlib in the case of X is very portable. However, even the designers of X grant that little user interface programming should be done at this level directly. This type of programming should be done at higher levels for productivity reasons.

3. Toolkits For X

While most X experts agree on the strengths and weaknesses that characterize Xlib and the X protocol, there is less agreement at the Toolkit level for X.

The significance of this problem is evidenced by the fact that one of the most persistent problems with X has to do with the difficulty of writing X applications. As described above, many of the problems stem from the design decision to leave Xlib flexible and devoid of policy. For all the freedom this offers programmers, it also implies a great deal of knowledge and work be done to write even a simple application properly. Fundamentally, the choice to retain flexibility at the **Xlib** layer was predicated on the assumption that policy issues and ease of programming would be resolved through the use of user interface toolkits.

The toolkit for X covered in this paper is called the X Toolkit or **Xt**. It is very closely tied to Xlib, making heavy use of windows. A client created with the X Toolkit, for instance, might contain hundreds of windows. Xt also uses X's resource data base heavily, allowing users to change things like colors, button strings, and more at the time an application runs. There are two layers within the X Toolkit: one consisting of *intrinsics* and the other being made up of *widgets*. Widgets are the graphical objects (menus, scroll bars, and such) that are key to the toolkit portion of the reference model. Intrinsics, meanwhile, are what glue the widgets and **Xlib** together. Applications communicate with Xt through the use of *Callback Functions* implying the toolkit extolls a *don't call us, we'll call you* communication model.

3.1 Portability at the Toolkit Level

As with the window system level, portability at the toolkit level is largely a question of availability. To write truly portable code, the toolkit Application Programmers Interface (API) must be available on a wide variety of systems.

The Xt Intrinsics are gaining widespread support in the industry. They are in the public domain and have the support of the following companies and standards bodies:

- Hewlett-Packard
- DEC
- AT&T
- MIT X Consortium
- X/Open

If all programmers needed to write applications were the Xt Intrinsics we'd be home free. Unfortunately, widgets are also required. A single standard would not seem to be forthcoming. As there may be two UNIX standards, at least two sets of widgets are likely. Motif from the Open Software Foundation, and OpenLook from AT&T.

At the time of this writing, Hewlett-Packard, Sony, an MIT have developed widget sets and placed them and their source code in the public domain. Up until now, if a programmer wished to write a portable program with the Xt intrinsics and widgets, using one of these public sets was an attractive option.

3.2 Productivity at the Toolkit Level

The main reason to use a toolkit is productivity. The best reference showing the productivity benefits of using a toolkit are shown in [5]. This article [5] compares the code required to write a "proper" program that prints *Hello World* into a window following reasonable X programming conventions. In the summary of the article the following data is given: *In the case of "Hello, World", a program that took 40 executable statements to program using the basic X11 library took 5 statements to program using the X11 toolkit. In addition, the toolkit version had more functionality ad better repaint performance than a library version with 60 statements.*

This demonstration shows a significant decrease not only in the amount of code written, but also in the knowledge required to write code for X. Since less code is written, fewer bugs are expected and we move back down the complexity curve. This moves us back towards the simple environment we once had when programming with libc and stdio. Plus, our users live in a more hospitable environment.

3.3 Conclusions at the Toolkit level

Much programming productivity is to be at the toolkit level. James Foley of George Washington University claims a 50%-80% gain in productivity when using a toolkit. The numbers given in [5] would seem to bear that out. Unfortunately, some portability would seem to be lost at the toolkit/widget level at the time of this writing. Using public domain widgets is an option, but hopefully one or two sets of widgets will emerge in the near future to give programmers a stable base for writing programs productively.

4. System User Interface.

For end users, SUI provides tools to control their computing environment, and the model for thinking about their environment in general. For programmers, SUI provides conventions and utilities for writing applications that work together. Having established that an X-based application can be developed more productively using a toolkit, we must look at other aspects of the programming environment that may affect productivity and portability. Unfortunately, the SUI level is the least understood part of the reference model.

Of the many components that make up a system user interface, the following have the greatest effect on the programmer:

- Inter-Client (Application) Communication [4]
- Look and Feel (Style Guide)

The following sections give examples of how programmers deal with programming at the system user interface level.

4.1 Inter-Client Communication

With several applications able to run concurrently on the same display, the usefulness of a mechanism for moving data between applications should be obvious. For instance, situations often come up where it would be helpful to be able to copy the value of a spreadsheet cell (or the value of a range of cells) to a text editor. To assist in such instances, X provides a facility called *Selections*, to assist in inter-client communication. In the following spreadsheet/text editor example, the selection mechanism is used to share data.

Step #1. If an end user were to move the mouse cursor to the spreadsheet client and select a range of spreadsheet cells, the spreadsheet client would highlight the cells as feedback to the user. Since the user may which to copy these values to another client, the spreadsheet would request, and receive, ownership of the *current selection* from the X server.

Step #2. The user could then move the mouse cursor to the text editor and perform some action--perhaps picking a "paste" item in a menu as an indication that the current selection, (the highlighted range of spreadsheet cells) should be pasted into the text editor. This would cause the text editor client to request the contents of the current selection from the spreadsheet client--thus indicating not only what type of data the text editor would like (probably a string), but also including its own window id. This would cause the spreadsheet to store the selected data as an X property on the text editing window and send the text editor an event indicating a successful transfer. The text editor would then be free to get the spreadsheet data and show it in its window. Thereafter, the primary selection would stay with the spreadsheet client until requested by some other client.

This example shows two responsibilities on the part of X clients: First, they must use the selection mechanisms to share data easily with other clients. Second, they need to give some feedback to the end user to show how to share the data.

The example given above shows one of many responsibilities given the programmer when writing an X application, some help in following the environment programming conventions would be welcomed by most programmers.

4.2 Look and Feel/Style Guide

The look and feel of a system is a conglomerate of all aforementioned system user interface components, and applications. Having a consistent system look and feel provides large benefits to end users. Positive transfers of learning occur when applications used have similar interfaces. Reduced training costs and frustration (for end users) are but a few of the results of a well organized and consistent system look and feel.

Some users also need to use a variety of operating systems and hardware. A user familiar with the Presentation Manager* on OS/2, or Microsoft Windows[6] on DOS is quite happy to see that look and feel moved to UNIX applications with Motif. Other users may prefer yet another interface.

For this reason, the designers of X did not choose a standard look and feel. X's resource database, providing run-time modification of Colors, Fonts, and other resources, together with X's toolkits provide the mechanisms by which system look and feel will be implemented.

4.3 Portability at the System User Interface level

There is some significant work in progress at the system user interface level. Witness the announcement of Open Look[7] from AT&T and Sun Microsystems, a look and feel/widget set whose first implementation will be on X and the X toolkit. The Presentation Manager is another logical look and feel candidate. Hewlett-Packard has announced its intention to move the NewWave environment from the personal computer world to UNIX and X. NewWave provides an iconic office desktop metaphor, and is an example of a system user interface that could be used by a computer novice. NewWave also provides an object system, that keeps track of a users data relationships, and an automated *record & playback* mechanism to generate shell scripts for users without shell programming. As a result of New Wave, HP will be providing the same user interface on its personal computers and UNIX systems. Digital Equipment Corporation is also providing significant technology at the System User Interface level.

Fortunately, for portability sake, all these vendors are committed to follow the *Inter-Client Communication Conventions* referenced in [4]. Writing programs that communicate with each other is a matter of following the conventions. Writing portable code at the application communication level should be possible.

It isn't as good a story as far as style guides are concerned. Each vendor is expected to "add value" to their implementation, and this is bound to cause programmers headaches. The best solution may be the adoption of a style guide from the PC world, such as the Presentation Manager so that at least one standard interface exists in the computing world. Following style guides for various computing platforms is going to cause problems as far as portability is concerned.

4.4 Productivity at the System User Interface Level

Inter-client communication and user interface style guides are one place where real trouble begins for programmer productivity. Thick documents explaining the rules for programming conventions and style guide conformance will proliferate. Unfortunately, the training required to learn rules such as these do not add functional value to an application. Therefore, a software vendor must make a choice as to whether new features should be added to an application, or programming conventions and style guides should be followed. Apple has already experienced trouble moving MacIntosh applications to

[6] Presentation Manager and Microsoft Windows are trademarks of Microsoft.

[7] Open Look is a trademark of AT&T

AUX because programmers either refused to follow, or did not understand Mac programming guidelines.

Expect similar problems down the line for UNIX environments unless some type of help for programmers is forthcoming. The rest of this paper will discuss two potential solutions to this problem.

5. Interactive interface builders

Much of the software development in the near term for X will be widgets. Expect to see spreadsheet and WYSIWYG text widgets in the future. The development of libraries of widgets will make programming interfaces much like laying out a board using a TTL data book.

The programming productivity gains realized by using a toolbox are large, but a quantum leap can be made with the development of tools that support the use of toolkits. It can take hours to lay out widgets in an applications window, for instance. Likewise, any radical changes to a finished layout can consume hours. To reduce the hours to minutes, we have an opportunity to develop *interactive interface builders* that can be used to edit windows. These tools will allow an interface to an application to be developed as easily as a text file is created using vi, or emacs. Using such tools, programmers will be more likely to try several user interfaces, rather than using the first one developed, and thereby produce superior products.

Interactive tools should be used to assist the programmer in several areas:

- Window Layout
- Environment programming conventions (i.e. Inter-Client Communication)
- Style Guide Conformance
- Application Internationalization

Humans should write less and less code, not more as is the case with the environments being pushed today. An interactive builder could help a programmer follow environment programming conventions and style guides without reading huge documents. Writing code for the increasingly global community in which we live should also become easier.

Eventually it should become possible even for non-programmers to construct the user interface for a new application, or to add a graphics interface to awk, grep, or mail. We will all be much better off when human factors specialists create user interfaces as opposed to software engineers. Interface builders are not yet a requirement at the toolkit level, but they probably will become a requirement soon enough.

UIMX from Visual Edge Software Ltd. provides the best combination of productivity gains in conjunction with portability concerns in the authors opinion. It (UIMX) is an interactive tool that helps create application interfaces in accord with the goals stated above, and is implemented on the Xt Intrinsics, and the HP X Widget Set. Expect to see other such tools to be available in the near future.

6. API's From Other Environments

Another solution that could help programmers be more productive is to have common programming interfaces between operating systems. Moving the Presentation Manager API to UNIX would help UNIX programmers if only because of the expected proliferation of training and books for the PM environment.

The difficulty here, will be to move the API to X and UNIX without sacrificing performance, and compatibility. This will be an expensive and time consuming effort but the dividends paid, if successful, would be great.

7. Summary

The world has become a complex place in which to program. The tradeoffs between writing portable code to standards versus productivity concerns are difficult to understand. Our best hope is to minimize the number of standard programming environments, and the amount of code written by humans. Perhaps by pursuing these two goals, we may avoid constantly rewriting existing code and be able to concentrate on more productive tasks.

8. References

[1] Robert W. Scheifler and Jim Gettys, “The X Window System”, *ACM Transactions on Graphics*, Vol. 5, No. 2 (April 1986).

[2] Jim Gettys, Ron Newman, and Robert W. Scheifler, “Xlib--C Language X Interface”, X Version 11, Release 2 (September 1987).

[3] Robert W. Scheifler, “X Window System Protocol”, X Version 11, Release 2 (September 1987).

[4] David Rosenthal, “Inter-Client Communication Conventions Manual”, draft (February 1988).

[5] David Rosenthal "Going For Baroque", UNIX REVIEW, June 1988 pp. 71-79

[6] Ed Lee "Window of Opportunity", UNIX REVIEW pp. 46-61

Open Electronic Messaging

and its role in EDI

AUGUST 1989

Ian Sharrock
Open Platforms Consultant
ICL Australia
ian@icl.oz

ABSTRACT

The evolution and acceptance of the X400 international standard for electronic messaging has progressed more rapidly than might have been expected.

X.400 initially defined a set of standards for the structure and a message medium for the transmission of text and data of any length as documents, document images and letters in the form of electronic messages. More recent international efforts to explore the convergence of X.400 Message Handling Systems (MHS) and Electronic Data Interchange (EDI) have begun. Resulting EDI traffic will considerably increase the size of the messaging market, and the availability of a communications backbone such as MHS will speed the growth of EDI.

This paper represents research on understanding,

- X.400 and its relationship with EDI,
- the current status of EDI and X400 as a standard and the relevant influences on the standards,
- the role of Unix,
- the responsibility and challenges to Unix vendors in introducing X.400 messaging architecture to their users.
- the relevant strategies and offerings from newly established VAS (Value Added Service) suppliers in Australia,

Whilst technically speaking a "global public messaging system" is achievable and desirable in today's climate, there are a number of commercial constraints preventing this from happening. The paper endeavours to explore a number of these constraints.

1. Electronic Data Interchange

Over a period of 20 years EDI (Electronic Data Interchange) has evolved to the state where it can be described as "paperless trading", in the same sense that EFT and EFT at Point of Sales (EFTPOS) can be described as "paperless payment". The basic principle of EDI is that computer- generated trading documents, such as purchase orders and invoices, are transmitted directly to a company's trading partner's computer via an appropriate medium. In today's ruthlessly competitive commercial environment, EDI represents an efficient, reliable, more effective manner to conduct business.

EDI involves the transmission from one electronic processing node to another of any common business document e.g. purchase order, invoice, payment instructions, manifests, schedules, etc. in accordance to a set of established dialogue rules. These documents can be generated automatically as the result of a trigger to an application or manually according to a defined procedure.

EDI has already been taken up and is being used extensively by complete industries, such as banks, airline booking, automotive manufacturer etc. or by powerful retailers wishing to trade with the suppliers or sales outlets (eg Coles/Myer). In fact most of today's EDI usage comes from such sources. A number of International Organizations have also been established specifically aimed to market EDI services in the countries with large established commercial bases. (eg INS in UK focus on the manufacturing, pharmacy and insurance markets.)

> *A classic and basic example of a business cycle which does have standards in place is what is referred to as the business supply cycle. This cycle is characterised by a number of defined steps and procedures;*
>
> *Request for quotation.*
> *Issue Order.*
> *Despatch Acknowledgement.*
> *Invoice.*
> *Statement.*
> *Payment.*

With EDI, this business cycle compacts from weeks to possibly hours, with benefits for both the supplier and customer.

2. EDI Standards

The structure and format of any business documents and the manner in which they flow within a specific business transaction amongst trading partners, relates to the nature of the transaction and is the result of long-established business rules, procedures and protocols. In today's modern "paper oriented" trading, the protocol and structure of documents or forms that are exchanged during the business cycle has evolved with time. The forces at play on the evolutionary process are many and varied and are very much affected by the manner in which that specific business operates. What can be said, however, is that the evolutionary process that continues to reshape a specific business cycle, has always continued to embrace technology, is "forms oriented", is aimed at efficiency and expediency but does not impact underlying legal requirements.

EDI is a generic term which loosely covers the structure and electronic interchange of business documents.

EDI standards for specific industries or business transaction types have evolved from various special interest groups throughout Europe and North America. (eg, Grocery Manufacturers of America, American Iron and Steel Institute, British Department of Trade and Industry). Rationalisation and consensus amongst North American special interest groups saw the formation of the ANSI X.12 committee in 1978, whose charter is to focus on the development of generic transaction types of message and data exchange.

The number of American document standards that have been developed since the initial codification of X12 has grown and these standards have gone on to influence most EDI implementations in use today.

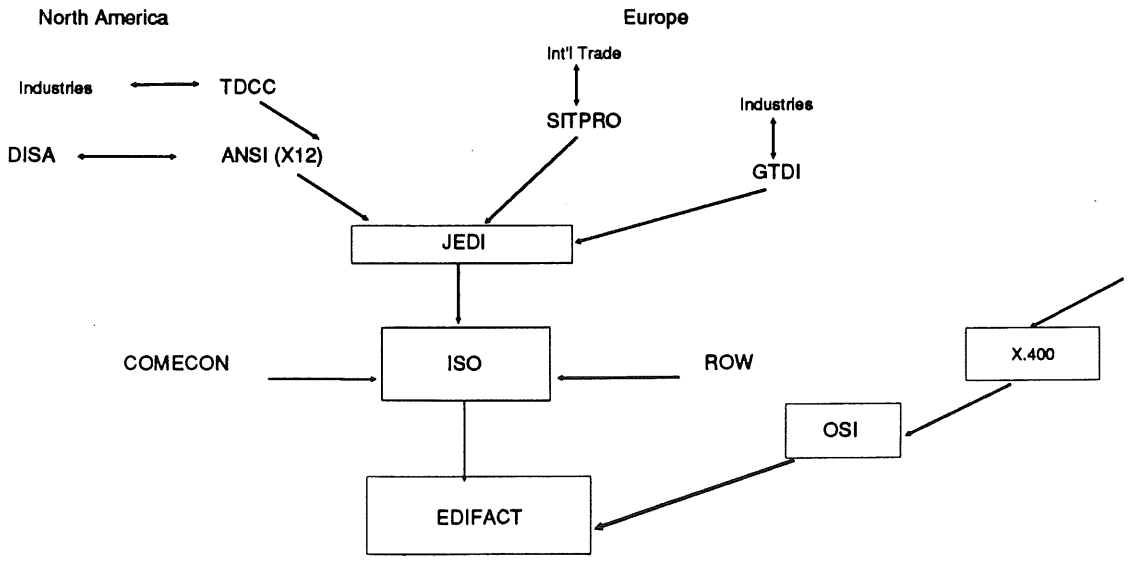
Under the aegis of the United Nations, ISO (International Standards Organization), has a key activity to liaise with the ANSI X12 committee and other similar bodies world-wide to specify EDIFACT (Electronic Interchange For Administration, Commerce and Transportation). The role of EDIFACT is to specify terminology, syntax, trade data directories and message types. By 1990, ISO anticipates that in excess of 1000 message types will be standardized, covering over 80% of the world trade.

For example;

Set No.	*Description*
110	*Freight Details (Air)*
314	*Shipment Details (Rail)*
843	*Response to RFQ*
875	*Purchase Order*
905	*Remittance Advice*
943	*Warehouse*

The requirement for a reliable EDI message tranfer and handling system was identified early by the various driving forces on EDI standards. With the growing popularity of EDI there is considerable interest in using X.400 as the message handling system for EDI data. In theory, because X.400 was set up to handle files of any type, it should be a strong marriage.

Figure 1. The Evolution of Standards



3. X400

Ever since computers moved from being batch driven to user interactive systems, a suitable environment has existed for developers to introduce schemes whereby messages can be passed from one interactive user to another. Various mail regimes have been introduced by different vendors to accommodate interactive messaging systems.

Unix mail has evolved from a similar requirement and is characteristic of these proprietary architectures. There have been many successful attempts to extend the standard Unix mail and to overcome some of the shortcomings in traditional mail systems. Products like MH-Mail have been aimed at improving the user interface. Similarly other products such as MHSnet and UUCP have had mail successfully integrated to provide the mechanisms for distribution over a wider network of Unix systems. Unix Office Automation (OA) product developers, not satisfied with standard Unix mail, have introduced their own proprietary electronic mail regimes to make mail more commercial and allow for integration into relevant OA modules.

Basic e-mail messages are text-based and for a Unix user, the choice of mechanisms to present and manage standard text messages are limitless. To achieve a global electronic messaging system with true compatibility between different e-mail systems, there needs to be consensus on the manner in which messages are routed, transferred and controlled within a mail network. That is, there needs to be consensus on the network wide message handling system (MHS).

In 1984, the CCITT approved the results of collaborative work undertaken by a number of companies interested in developing a comprehensive set of technical specifications for the interconnection of e-mail systems. The eight documents that resulted from their efforts are referred to as the X.400 Series recommendation.

Since 1984, the initial X.400 recommendations have steadily achieved acceptance throughout the wider e-mail industry. Most office equipment vendors, computer manufacturers, and Value-added Service providers are implementing or offering X.400 interconnect capability.

In 1985 CCITT began work on several key extension to X.400 that enable, for example, secure messaging, user friendly naming and distribution lists. In Melbourne, CCITT's Plenary Assembly ratified X.400 (1988) second edition.

The Message Handling System.

The MHS (Message Handling System) model serves as a tool to aid in the explanation of the standard. The model uses the techniques of the OSI reference model to formally define the layered communications structure.

In the case of interpersonal messaging (IPM), a user prepares messages with the assistance of an IPM User Agent (UA). A UA is an application process that interacts with the Message Transfer System (MTS) to submit or take delivery of messages.

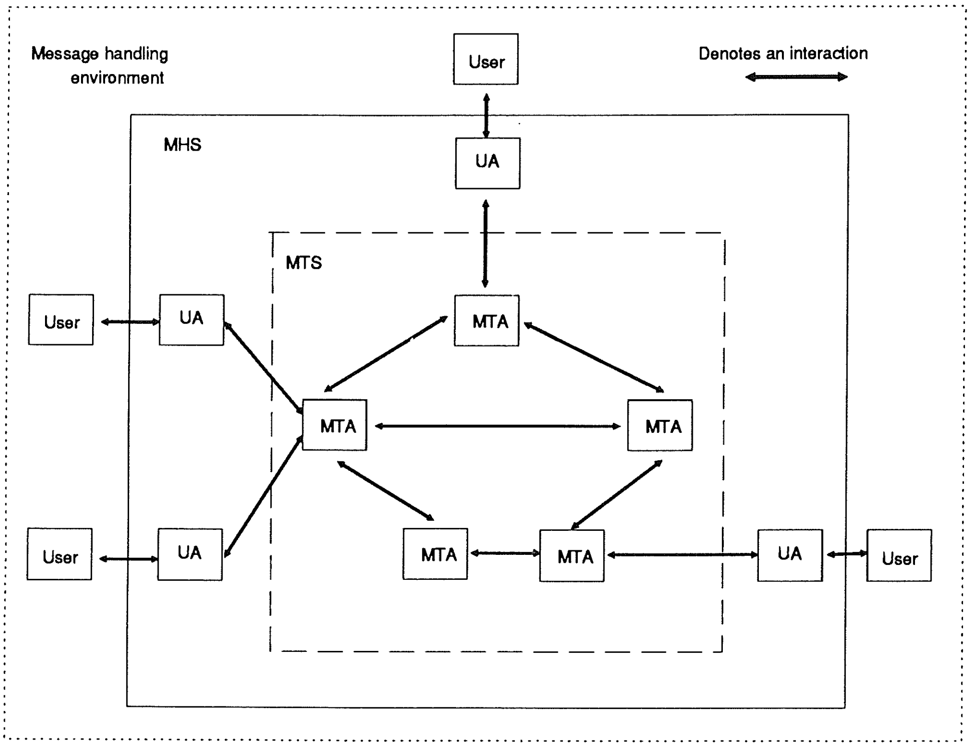


Fig 2. The MHS Model

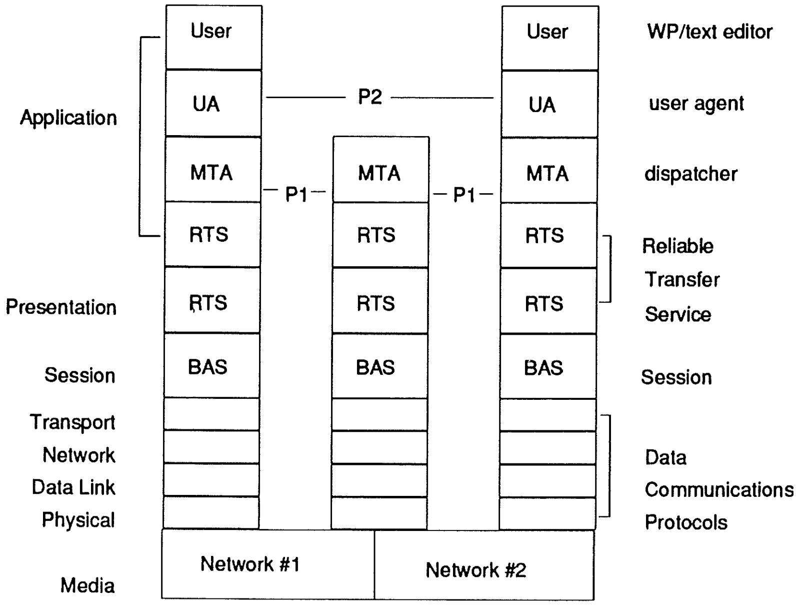


Fig 3. The Relationship between OSI and X.400

The MTS consists of a number of MTAs (Message Transfer Agents - sometimes referred to as post rooms) operating together. The MTAs relay messages and deliver them to the intended recipient UAs, which in turn make the message available to the intended recipients.

A collection of UA's and MTA's is referred to as a message handling system (MHS).

The CCITT has defined a common message transfer protocol referred to as "P1", that is used to transfer messages between MTAs. For interpersonal messaging (IPM), another protocol "P2" is used. "P1" defines the electronic message envelope while "P2" specifies the envelope contents in terms of an interpersonal message.

For reliable transfer of messages between MTAs, OSI implementations of X.400 provide a Reliable Transfer Service (RTS). Most RTSs utilizes the OSI Transport service subset and guarantee delivery of complete messages between MTAs by providing checkpointing, and restart of data transfer after transport connection failure.

Naming and Addressing Conventions

Within an IPMS, each MHS user is assigned a unique hierarchical O/R name. The prefix "O/R" recognises that a user can either be a recipient or an originator of a message . An O/R name distiguishes one user from another user address, and it enables the MHS to locate a user to deliver a message to a users UA.

An O/R name is a collection of one or more name attributes whose syntax is defined. Some of the attributes are used to unambiguously identify the "management domain (MD)" to which the user is attached. These "base attributes" are important to the MTA so it knows how much of the O/R name to consider when transferring a message to the next MTA.

The base attributes are:

Country	*AU Australia*
Administrative Mail Domain	*in Telecom "Telememo"*
Private Mail Domain	*ICL*

Other attributes can include

Organization
Unit
Personal Name

Example: I.Sharrock/AUS07A/ICL/Telememo/AU

4. The Vendors

Tactically, X.400 can be seen as a gateway specification, as a technology base for connecting one proprietary e-mail system (or component) to another, that is a common ground. This position is characteristic of the first generation of X.400 offerings from manufacturers of end-user commercial products (both hardware and software). As a result of market requirements and user pressures, one of the first areas to move into X.400 messaging is Office Automation (OA). Manufacturers of products such as Officepower, Uniplex, All in One, Quadratron, CEO and Wang Office offer or plan to offer X.400 gateways, while continuing to interwork internally with their proprietary messaging system.

The shortcoming of such products is that they can only interwork with other products built by another vendor to a limited degree. Users can not exercise all the features of both. Such imperfections, intrinsic to the gateways, result from architectural differences between the product and the gateway specification to which it can only approximately conform.

From a strategic point of view, X.400 can be considered as the chosen architecture for a messaging system. Products characterised by a native X400 MHS are now starting to appear on the market. Products, such as OfficePower from ICL and Retix Mail from Retix Corp California USA, do not have features that depart from those described by X.400; thus, being architecturally aligned to X.400, have a high level of interworking.

Computer product vendors or software solution houses in the EDI market have, up till now, been offering some form of proprietary EDI. From the vendors point of view the use of X.400 messaging as the basis of EDI is a relatively new phenomen and it this stage no clear direction appears to be apparent.

However, because EDI, when applied as an "open" solution requires participation from a number of hardware vendors, software solution houses and network providers, and because X.400 will have to fit into established defacto standards such as TCP/IP, MSDOS and VMS as well as established standards such as OSI and UNIX, the trend towards collaborative EDI solutioning has evolved. Examples of collaborate efforts between what have been, up till now, traditional network providers and software houses whose services cover complete industries, have started to appear under the generic term of Value Added Services (VAS) Providers. Possibly the most visible examples of VASs are those that have been established by OTC and Telecom. Telecom and OTC are fiercely competitive in this area and in conjunction with their respective partners can offer today a number of X.400 based EDI solutions.

5. The VAS Provider position

Australia Telecom

Telecom, through their Value-added Services (VAS) operation known as Telecom Plus, offer a range of messaging and EDI services. Telecom's messaging system is an extension to the familiar "Keylink" service and is based on a commercial messaging product from the USA called Telemail. Telecom claim this messaging system is X400 compliant.

Telemail distribute their products and services through a number PTT's throughout the world; of Telecom Australia is one. The Administrative Management Domain naming attribute for Telecom's Telemail electronic message service is "telememo".

The two current messaging services being offered by Telecom Plus are:

1 Connection of Private Management Domain Messages Transfer Agents to Telememo via X.25.

Users wishing to connect to the "Telememo" must have X.400 compliant MTA's.

Proposed
Tariff: *85 cents per k/char international.*
40 cents per k/char local.

2 Keylink is a public, user interactive message processing system based on telemail X.400 messaging service. Users connect to the Keylink User Agent via character modeinteractive terminals and an X.25 PAD.

Proposed
Tariff: *46 cents per k/char local.*
86 cents per k/char international.
plus
20 cents per minute connect time.

Telecom have commercial arrangements with other Telemail based message service vendors throughout the world. Telecom claim to offer not only an integrated service between Keylink and their X.400 ADMD public service but also an almost global integration between both services and a LIMITED NUMBER OF SIMILAR SERVICES that are predominantly Keylink-based throughout the world.

Telecom Plus have two EDI service offerings based around X.400. The two facilities, one in Sydney and the other in Melbourne, represent collaborations with partners and offer services that focus on two specific market requirements.

1 TRADELINK is a collaboration between Telecom and ACI computers. TRADELINK offers solutions to and is focused at the commercial supply cycle of particular industries. It consists of consulting services that offer solutions to particular business needs based on Telecom's X.400 messaging service.

So far, only one TRADELINK-based solution is up and running. The solution provides the ability for Automotive manufactures to order electronically on their suppliers.

2 T-NET is a collaboration between Telecom and P&O shipping. T-Net offers a range of services designed specifically for Transport and Shipping and allied industries. T-Net is a spin- off from the successful collaboration with ACI with a different focus. Once again the EDI solutions being offered by T-Net are X.400 messaging based.

So far, the only T-Net solution being offered is WoolCom, an industry-specific product aimed at businesses specialising in the preparation, storage, export and shipping of wool.

OTC Australia

OTC Australia offer a range of EDI services through joint arrangements with a number of complementary companies, including NEIS and Computer Power. In addition, OTC in conjunction with British Telecom offer X.400 and X.400-based EDI services through a joint venture company known as Network Innovations.

Network Innovations based in Sydney offer competitive services to those offered by TelecomPlus.

1 Users with X.400-compliant MTAs can connect their private mail domains to OTC's mps400 service. The mps400 service consists of two distinct elements or systems. Whilst each element has a different origin, OTC claim that a seamless interface of services exists between both elements and market mps400 accordingly. Users of mps400 can be identified as having as having the ADMD name of "otc".

The Elements are;

X.400 MHS software developed by the University of British Columbia referred to as Messager 400 and running on a VAX provides the backbone to OTC's EDI oriented services. In having a source license to Messager 400 OTC are able and willing to tailor their X.400 offerings to suite user EDI requirements.

A recent example of this is the service developed in conjunction with Colonial mutual. This service provides a means for Field Representatives to forward and receive insurance proposals and documentation on portable personal computers.

For users that have a more "generic" IPMS requirement, OTC will hook into OTC's Dialcom service. Dialcom 400 is a worldwide competitive product to Telemail and is distributed and marketed under similar arrangements. Dialcom is US based but owned by British Telecom.

2 OTC's public user interactive message system is also known as Dialcom and competes with Telecom's Keylink. Once again, Dialcom products are utilized to provide this service. Users of the service have access to the "closed" worldwide base of Dialcom's interactive users. OTC will open up Dialcom and plan to integrate the "Dialcom" public message system into mps400 in the foreseeable future.

Tarrifs are not available from OTC on either of these products.

Possibly the most well known EDI implementation in Australia is EXIT. EXIT is the result of OTC's successful bid for Australia's Customs Service's requirement for a fast and accurate tracking of exports. EXIT is designed to be taken up by those parties in the import-export cycle and is specifically intended to improve Australian Custom's efficiency in this area. It is anticipated that EXIT will ultimately track 360,000 consignments shipped by more than 3000 shipping agents.

The EXIT EDI requirents have been met by tailoring OTC's standard X.400 product to adapt to the EDI application. OTC consider that in having source and hence control of their own X.400 product, they can offer more flexible EDI solutions and can respond more quickly to market requirements. Whilst OTC are not prepared to disclose their future EDI strategies they do see a distinct advantage over their competition by being in such a position.

6. Global Messaging Systems

Being able to transmit text and data of any length as a document, letter or message almost instantly anywhere in the world is a powerful facility. But to work, it is essential that all message carriers conform to the X.400 international standard.

The X.400 standard has come a long way in its four short years. To achieve a global messaging system, connectivity between relevant ADMDs for public traffic needs to be established. Given the commitment of telecommunications providers, network value added service providers (VAS) and computer companies to the standard, you would not be mistaken in thinking that this connectivity is achievable today.

Whilst technically it is possible, a number of commercial influence are stalling the process.

The competitive climate that exists between OTC and Telecom is not unique to Australia. In such a climate, where two or more rivals are competing for dominance in the EDI/Messaging market, vested interest is first before consensus on dialogue towards connectivity.

Further, for a message to pass from one side of the world to the other, it may be required to transit a number of message carriers. Calculating the tariff levied on the originator would involve all the carriers responsible for transitting the message. Todate, agreement on charging has not been reached.

7. X.400 and EDI Convergence.

Whilst X.400 is oriented towards Inter Personal Messaging most of the message handling features of the X.400 Message Handling System are applicable to an EDI message handling system.

A new international effort to explore the convergence of X.400 MHS and EDI has recently begun. MHS and EDI seems likely to be a powerful combination. EDI traffic will considerably increase the size of the messaging market, and the availability of a communications backbone such as MHS will speed the growth of EDI.

Although X.400's architecture is set up so that it can transfer EDI documents, such as purchase orders and invoices, the specific service elements for an EDI application do not exist yet.

example: *While the X.400 IPM service element identifies sender and recipient(s), they do not recognize that either of the communicating parties have a multiple trading arrangement together. Instead, X.400 assumes that a sender and recipient have a single relationship (mailbox) that receives all messages. To be suitable for a EDI application level service, X.400 would have to recognize that one company could have multiple trading relationships involving separate trading agents.*

8. Summary and Conclusion

X.400 as an international accepted standard has achieved rapid success. Major administration and service providers have launched their X.400 service - while major suppliers have produced their end-user systems or X.400 gateways from their proprietary product offerings.

In addition, it must be remembered that X.400 is not restricted to electronic mail, ie. interpersonal messaging. It also offers a common Message Transfer Service, that can carry all sorts of electronic data. One of the most important candidates is the electronic business trading data in the form of EDI.

However, for X.400 to succeed as a carrier of EDI messaging, X400 will have to be extended to overcome the philosophical division between IPMS and EDI. X.400 is designed for passing messages between parties that have a messaging relationship while EDI has a requirement for a MHS to pass messages between parties that have a trading relationship.

Before X.400 can achieve the status of "global electronic messaging system" a worldwide agreement on the connectivity of Value Added Service (VAS) providers and Administrative Mail Domains (ADMD) must be reached. Apart from the agreement on tariffs, the biggest single influence working against this connectivity is the commercial competitiveness of parties offering such services in their respective regions.

For organisations strategically adopting X.400 as their native messaging architecture, numerous products are and will soon be available. Most products available are tactically based, for example offer gateways to X.400 but others are strategically based, and X.400 as the native messaging architecture. Hence organizations adopting X.400 will have to choose products carefully to fit in to their overall strategy. For such organisations the benefits of uniform messaging and a fixed interface between system components will be similar to those gained by organiations which have chosen to adopt Unix as the basis of it being an "open" operating system.

X.400 is a working commercial example of an OSI Layer 7 application. After nearly a decade of standardization, global e-mail is now clearly on the horizon. Unix is a product that is competing successfully to win business as a Departmental System from its acceptance as a major industry standard. To cement X.400's position as a major industry standard, UNIX vendors and UNIX users should be considering X.400 as the first major application of Open Systems Interconnection within its repertoire of communications capabilities.

9. References

CCITT Red Book Volume V111 - Fascicle
Data Communication Networks Message Handling Systems

OSN Newsletter Volume 2 Issue 7
CCITT Takes First Steps To X.400 and EDI Convergence

The A to Z of EDI Paul Kimberley

Article - X.400 Standardized As An Interconnector Of E-Mail
James White

UUL - UNIX User Limits

Ray Loyzaga yar@cs.su.oz

Rex di Bona rex@cs.su.oz

Basser Department of Computer Science
The University of Sydney

1. INTRODUCTION

The Basser Department of Computer Science has recently moved its undergraduate teaching workload from a single Vax 11/780 running AUSAM to four Mips M120's running UNIX† System V.3. The improvement in response time has been tremendous, but the loss of a secure, manageable environment has had quite an impact on the department's programming resources.

1.1 Problems with Unlimited UNIX

Students within our environment have an unfortunate tendency to write incorrect programs. Some of these students make demands on the system that have the potential to bring any system to its knees. Runaway programs can completely devour disk space, provoke massive amounts of swapping or just soak up so much CPU time that other students suffer. Obvious areas that require attention are:

- Disk usage limits
- CPU usage limits
- Process limits
- Memory limits

1.2 Anecdotal Student Incidents

It is not just run-away programs that cause problems. Some students are amazed by the speed of the machine, and correspondingly their finesse at writing efficient programs has decreased. One example of this is a program written by a student as an alarm clock. It was a shell script that continually calculated the time and compared the result with the requested time (Figure 1).

```
#!/bin/sh
while true
do
        if [ `date | cut -c12-20` = $1 ]
        then
                echo $2
                exit
        fi
done
```

Figure 1. Alarm Program

† UNIX is a trademark of Bell Laboratories.

2. *RELATED WORK*

Several other systems have been developed that encompass parts of the UUL system. Each of these have had certain advantages and disadvantages.

2.1 AUSAM

AUSAM was developed at the universities of Sydney and New South Wales, with input from elsewhere. It is a comprehensive package designed to provide a secure run time environment for student teaching based upon UNIX machines. It provided control over virtually all aspects of a users ability to use the system. Amongst it features were;

- Disk limits
- Memory limits, and Memory working set limits
- Process limits
- Printer limits
- Terminal Access control
- CPU fair sharing
- Flags for controlling resource access
- Connect time limits
- Automatic Account expiry
- A umask mask to restrict file permissions

The AUSAM system required major changes to both the kernel and to user programs. It worked by adding a per user structure to the kernel. This structure, called an *lnode*, contained all the information that the kernel required to perform limit checking, and resource accounting. When not active user *lnode*s were stored in the password file, which was extensively modified to allow for the additional data.

The password file was changed from a simple random indexed ASCII file to a hashed binary data base. The entry for each user contained not only the information required by the kernel, but also information that was used by user programs. The printer limit was an example of this; the number of printer pages allowed, and the number used, were both stored in the password file but only updated by the printer daemon software. This change to the password file was a significant advantage for a system which had a large number of users, as it provided fast accessing due to its hashing scheme, but was a major disadvantage as programs that expected a simple linear ASCII file would no longer work[1].

AUSAM has several major disadvantages. It performed all charging based on the *lnode* of the invoking program. The *lnode* was only changed by an explicit system call. Setuid programs (e.g. *mail*, *hack*, *lpr*, *sendfile*) did not change the *lnode*, but did change permissions and ownership of created files. Use of setuid programs could

[1] This disadvantage was removed in a version of AUSAM which used a mounted process to simulate an ASCII file for /etc/passwd.

create a situation where the invoker of a program is charged for files that are owned by the owner of the setuid program. These files, in most cases, are removed by a daemon, such as the printer daemon, that is started as boot time. This results in the invoker having a higher than real disk charge, and the daemon owner having a lower than real disk charge. The solution to this was to run, each night, a daemon that corrected the real disk charge of each user.

2.2 Quotas

The popular BSD variant of UNIX includes an optional piece of code that performs quotas for disk usage limiting [Lef89, Sun86]. A user is given a per-filesystem limit on the amount of disk available. An attempt to use more than this results in an error, and the operation is not performed.

The quotas system does not attempt to limit anything other than disk usage. It has the drawback of increased administration, as a limit has to be allocated for each user for each filesystem that is to be quotaed. The quota system also slows down a system reboot by doing a complete scan of each filesystem which have quotas to calculate the real disk usage for each user before the system allows logins.

2.3 Ulimit

System V provides for a very simple limiting of file length on a per-file basis. The *ulimit()* system call provides a simple user changeable limit for maximum file size [Mips88]. This allows a maximum file size to be set per user by having the *login* program set the limit to the maximum possible value. A user other than the super user can only decrease this limit.

2.4 Share

Developed at The University of Sydney in conjunction with the AUSAM package was a fair share scheduler [Lau80, Kay82, Kay88]. This system allows for a per-class user share of the machine. Share only tries to share out the CPU, it provides no other limit services. The standard UNIX scheduler is based upon a round robin system [Bach86] which does not perform fair scheduling as a single user with many processes ready to run receives proportionally more CPU time than a user with only one process ready to run.

2.5 Summary

Each of the present systems aim at providing UNIX user limits, but all fail at providing a general and comprehensive limits system. The closest current system is the AUSAM system which has shown major flaws during its lifetime here at Basser. The UUL system is based upon AUSAM, but corrects the discovered flaws. UUL also being extended to provide a limits system for multiple kernel systems.

3. DESIGN OVERVIEW

The system described is a descendant of AUSAM, many ideas have been extracted from the AUSAM implementation running at Basser.

A difference between AUSAM and UUL, and an influencing factor on the design of UUL, is that the department must ensure that a student has a certain minimum level of resources available, mainly in the form of CPU, connect time and an amount of disk space. On a totally uncontrolled system this requirement cannot be guaranteed to be met.

3.1 Philosophy

The philosophy of UUL is to provide an efficient method of controlling the amount of system resources available to individual students. One of the major restrictions that we imposed on this development was that the system changes should not require the modification of any existing user sources, including /etc/init and /etc/login.

The major design decision of the UUL system is the strict enforcement of a "User requested" policy. A user process is defined by its real, not its effective, user id. It is this user id that references the *lnode* that is charged for all resources the process consumes. One major exception to this is disk accounting. All disk storage charges are accrued to the *lnode* of the owner of a file. This way setuid programs and publically writable files behave as a-priori expectation would suggest. We believe this to be a more natural model. It is thought that the owner of a setuid program is willing, by the chmodding to setuid, to be accountable for all disk charges that the program accrues, but the CPU charge is a more difficult problem. It is the invoker who should be charged for this as it is the invoker who requested the program to be executed.

3.2 Lnodes

As with AUSAM, the major data structure associated with the limits of each user is an *lnode*. A unique *lnode* structure is associated with each user id. This structure contains values that define the various user limits and any usage information. Each process entry also contains a pointer to the associated *lnode* structure. Each *inode* also contains a pointer to the *lnode* for the owner of the file if the file is open for writing. This is changed only when a successful *setuid()* system call is made to change the real user id of the invoking user.

A limits file exists to store this information. The name of the limits file is provided via the system call which enables limits accounting. This system call is analogous to the SysV *acct()* system call. The use of a limits file differs from the AUSAM method of a merged binary password/limits file. This conforms with the design goal of minimum system impact, and with no changes to user programs other than those specifically dealing with limits.

3.3 Changing Lnodes

An *lnode* is accessed via the real user id rather than the effective user id of a process, this means that the *setuid()* system call provides the means for transferring to a new *lnode* structure. An advantage of this system is that the AUSAM style *limits()* system call is no longer required, this call was used by *login* to set *lnode* structures, and by *init* to reap and adjust *lnode*s for non-active users. An *lnode* is not transferred when a setuid program is *exec*'ed.

A *lnode* can now become active through two different methods. The first, and most common will be through the *setuid()* system call when the real user id of the owner of a process changes. The second method is through system calls such as write, close, unlink, truncate; any system call which either allocates or deallocates disk space. The *lnode* which is used is the *lnode* referenced by the user id of the owner of the file. This preserves the semantics of the "Disk Usage" field of the *lnode* structure, and removes most of the need for the *dlscan* program required in AUSAM for disk resynchronisation.

A *lnode* becomes retired when two conditions are met. Both the last process that referred to the *lnode* has died, and been *wait*'ed upon; and there are no files open belonging to the user id that the *lnode* is referenced by.

3.4 Limit Accounting

System areas that required changing are:

- Process Creation
- Process Reaping
- File Opening
- File Deletion
- File Expansion or Contraction
- Other Operations that affect charges which have limits

There would also have to be provided means whereby the current limits of a particular user may be modified by an authorised user, mainly the super-user, because of a change in a particular limit. Provision also has to be made to both enable and disable limit accounting.

4. IMPLEMENTATION

Changes are required in several parts of the kernel. The most extensive are required in process creation and deletion, and file extension and contraction. The interface to the limits operations is now a side effect of the requested system calls. This provides an automatic checking of limits whenever a system function that might modify the charge is performed.

Other limits functions; starting, stopping, and on the fly modification, are all performed through two added systems calls, the *limits* system call, and the *limctl* system call. These system calls, and their functions will be described below.

4.1 The Limits File Structure

The limits file contains an entry for each user on the system detailing current limits and usages. This file is a sparse file, users who do not exist are represented as "holes" in the file. This file is indexed by the associated real user id.

To ensure easy access for the kernel to an *lnode* entry the limits file is organised so that an *lnode* entry is guaranteed not to cross a block boundary. Since an *lnode* entry is now always contained within a single block only one block fetch is required by the kernel to retrieve the entry. This decreases the access time for *lnode* retrieval, speeding up access and also reduces the number of blocks in the buffer cache used by the UUL system. If the disk has sufficiently large blocks multiple lnode entries are stored on one disk block.

The actual *lnode* entry in the limits file contains two parts. These are the kernel data and the user data. The kernel is not interested, and does not track charges that are of interest to user level programs. The printer system is an example of this. A printer daemon only modifies that section of the limits file structure that contains the printer charge. The kernel only modifies that section containing kernel charges and so no mutual exclusions are required to control update accesses.

4.2 Booting Limits

The limits system could start automatically upon kernel boot, or it could be started explicitly (cf. *acct)* by a process in one of the boot files. Having limits start at boot is advantageous in that correct information is kept even for root. Having limits started by a system call has the advantage that it is a minor change to allow limits to be disable by the same system call.

A combination of these two is the best solution. The kernel keeps track of accounted *charges* from boot time, but only starts enforcing limits when limits are explicitly enabled by the limits system call. Before the limits system call is executed all charges are accrued as normal, but since no actual limits are known it becomes impossible to decide when a charge has exceeded its limit, so it is assumed that all users have no limits. When limit enforcing is enabled by the limits system call any change to a charge will result in a comparison with the associated limit being done, and having an excessing charge will result in the appropriate action; either a warning, or failure of the change.

If limits enforcing is kept off then it is possible that the number of active *lnode* structures exceeds the kernel data structure to hold them. If this occurs then data loss will occur, as *lnode*s will be discarded using an LRU scheme [Knu75]. It is possible to actively control whether enforcement is to be carried out on a per-system basis. It is unclear currently whether a finer grained control is needed.

4.3 Kernel Changes

4.3.1 Additional Routines Four main routines are required to be added to a kernel to implement the limits system, these are the creation or reading of an *lnode*, the closing or writing of an *lnode*, and the external interface handlers. To interface with the external world two additional system calls are added, these are the *limits* system call, and *limctl* system call. The *limit* system call is used to enable and disable limits enforcement. The *limctl* system call is used to query or modify the incore *lnode* structures. Two internal routines need to be written; these are *lget* and *lput. Lget* retrieves an *lnode* if necessary and increases the reference count for that *lnode. Lput* reduces the reference count and writes the changes back to the *lnode* file when the reference count reaches zero.

4.3.1.1 Limits system call The limits system call is used to enable or disable limit enforcement. The semantics are similar to those for *acct()* in System V;

```
int limits(path)
char *path;
```

If path is non-zero, and points to the name of a valid limits file, and limits enforcement is not already turned on, and the caller is the super user then limits enforcement is enabled using the stated limits file for limits information. If the name is a zero pointer, and limits enforcement is enabled, and the user is the super user then limits enforcement is disabled.

4.3.1.2 Limctl system call When a change has to be made to an *lnode* the avenue taken is through the *limctl* system call. The semantics for *limctl* are;

```
#include <limits.h>

int limctl(func, limptr)
int func;
limit *limptr;
```

The parameter *func* is used to select whether the call is requesting the .current data on a particular *lnode*, or whether the *limptr* field points to data which will be used to modify either the charge or the limit of a particular user. This is used after a system crash to correct disk usage and by the query program to obtain current kernel charge information. Only kernel specific information is returned by this system call. The query program must also read from the limits file to obtain the charges for user level programs.

4.3.1.3 Lget routine The *lget* routine is used by other kernel routines to ensure that an *lnode* is available for updating. If the requested *lnode* is not available the requesting process waits upon a disk read for the *lnode* and then is woken up. When the requested *lnode* is located its reference count is incremented, and a pointer to the *lnode* is returned.

If limit enforcement is not activated *lget* will return the *lnode* if available, otherwise an empty *lnode* will be returned with all limits disabled. If enforcement is subsequently turned on a call to *lget* will detect this, read in the limits file *lnode* and update it with the changes that have occurred.

4.3.1.4 Lput routine For every *lget* that is performed a corresponding *lput* must be executed when the *lnode* is no longer needed. A call to *lput* will decrement the reference count. When the reference count reaches zero the *lnode* is written out to the limits file, and then placed onto the retired free list. A subsequent call to *lget* will return the *lnode* from the retired list if it is still available. This caching of recently used *lnode*s reduces the number of disk accesses required by the UUL system.

When limits enforcement is disabled an *lnode* will not be written to the limits file when its reference count reaches zero. Instead it is placed on the end of a history queue, and will be retrieved by *lget* if limits enforcement is enabled.

4.3.2 Modification of Routines Several kernel routines need changing. These routines are divisible into two broad groups. The first group of routines are those associated with *lnode* activation and retirement; those which affect the existence of a particular *lnode*. The second major group are those routines which affect the data within the *lnode* itself.

4.3.2.1 Lnode activation and retirement There are two main ways in which an *lnode* can become activated. The first, that associated with the establishment of a new *lnode* for a process, is within *setuid()* (and *setreuid()*) which transfer the process from the active *lnode* to the *lnode* of the new real user id. The *setuid()* (*setreuid()*) system call uses *lget* to obtain a pointer for the new *lnode*, and also calls *lput* after reducing the number of active processes that refer to the old *lnode*. This call to *lput* might retire the old *lnode* if the calling process was the only process referring to the *lnode*, and no files owned by the *lnode* owner are open for writing.

The second method by which an *lnode* can become active is through a system call which modifies the value of some kernel parameter in the *lnode* structure. For example, a call to one of *chown()*, *close()*, *open()*, *rmdir()*, *unlink()*, *write()*, (and on BSD systems *truncate()*), could change the disk charge of a user *other* than the current *lnode*'s user. If the current *lnode* is different to the real user id of the executing process then one of *bind()*, *creat()*, *link()*, *mkdir()*, *mknod()*, *open()*, or *symlink()* could also cause an *lnode* to become active.

It is questionable whether the *lnode* should be modified at the low level block allocation and block freeing routines, or at a higher level. It appears that some actions, such as *write()*, benefit most from doing limit checking at a low level, but some functions, such as *chown()*, operate at a higher level. In the final decision each operation has to be examined and a decision made whether to alter at a low or high level based upon the semantics of the actual operation.

An *lnode* can become retired when either the last process closes, or the last charge change has occurred. Processes die completely when a *wait()* system call is executed which reaps the zombie of the process. Files open for writing stop affecting the referenced *lnode* upon a close of the last file descriptor open for writing on that file.

System calls such as *unlink()*, (*truncate()* on BSD), *mknod()*, *mkdir()*, and *rmdir()*; system calls which affect files based upon their PATHS must do an explicit *lget* at the beginning of the operation, and an *lput* at the end of their operation to ensure that the *lnode* is available to be adjusted for the change in charge that may occur.

4.3.2.2 Lnode modification *Lnodes* are modified by many routines, the disk modifying routines are outlined above. For changes of other charges similar code would be wrapped around the required system functions. For system functions that can guarantee the required *lnode* is available no additional call to *lget* and *lput* are required. For other routines (as above, routines that referred to files by name, not file descriptors) a call to *lget* must be made on routine entry, and a call to *lput* at routine exit.

4.4 User Level Changes

One of the design goals was to remove all user level changes, other than those directly associated with limits and the like. The only user level program directly affected is the super user command, *su*, which used to change both the real and effective user id to the super user id. This caused problems with the share scheduler as the scheduler gives priority to super user processes. This is corrected by running with only an effective user id of the super user, not a real user id of the super user. All permissions are still checked as super user, but scheduling is now done on the *lnode* of the invoker. As a side benefit setuid programs can now no longer trap unwary persons running as the super user by calling *setuid()* (*setreuid()*) to change their effective user id to their real user id.

The current version of *su* was moved to a new name and a new program written as outlined above. The old version had to be kept, as there are situations in which having a real user id of the super user is required. During catastrophic system behaviour is one example, another is for programs that check the real user id.

5. PROBLEM AREAS

The scheme outlined above would have been a great improvement over the AUSAM scheme used on our VAX. Unfortunately we have replaced the undergraduate VAX with four machines all running NFS, Sun's Network File System [Sun89], with students encouraged to load balance by switching machines. Each machine presents an identical file system view to the students. This similarity raised problems that are not encountered on a single kernel machine, the problem of trying to coordinate the kernels.

5.1 Multiple Machine Limits

Each of the four M120's run an independent kernel and this leads to problems of keeping the limits files consistent. How do we set limits in such a way that no user can exceed them by just moving to another machine? A single limits server could achieve this, but would necessarily require network I/O, and would have a grave performance impact on process and file creation times. A call to *lget* could conceivably require a network transaction to fetch the required *lnode*. Having separate servers on each machines is a better system, but has synchronisation problems.

5.2 NFS Disk I/O.

The use of NFS on multiple hosts leads to the question of where (which machine?) is the right place to modify the *lnode*? Does the use of different machines allow the user to exceed his disk limits? The approach that we have taken will only account for local disk blocks. NFS disk usage will be accounted on the *lnode* on the remote machine. This means that a users' limits should be different on the machine that contains the users home directory. All other machines should have limits that would allow the creation of files on the local filesystems such as */tmp* so that the users can function normally on hosts that are remote to their home directory.

6. CONCLUSION

User limits are both a practical and possible extension to the UNIX operating system. It is possible to create a system which is robust enough that users cannot, either maliciously, or accidently, destroy charging information. This system provides a flexible method to allow system administrators to allocate resources based upon local criteria. Its one current drawback is its inability to easily provide shared limits between machines. This inability is the subject of ongoing work at The University of Sydney.

7. REFERENCES

Bach86. Bach, M.J. *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs.

Kay82. Kay, J., Lauder, P., Maltby, C., and Tollasepp, S. ''The Share charging and scheduling system''. Technical Report 174. Basser Department of Computer Science, The University of Sydney, Australia,

Kay88. Kay, J., Lauder, P. ''A Fair Share Scheduler'' *Communications of the ACM*, **31**(1), pp. 44-55 (January 1988).

Lau80. Lauder, P. ''Share Scheduling Works!'', Internal Documentation, Basser Department of Computer Science.

Lef89. Leffler, S.J, et al., *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley.

Knu75. Knuth, D.E. *The Art of Computer Programming, Volume 3, Sorting and Searching*, Addison-Wesley, pp. 478-479.

Mips88. MIPS Computer Systems, Inc. *Programmers Reference Manual*, Part Number 84-00136, MIPS Computer Systems, Inc. Sunnyvale, CA.

Sun86. Sun Microsystems, Inc. *UNIX Administrators Guide*, Revision B 17th Feb 1986, Sun Microsystems, Inc. Mountain View, CA.

Sun89. Sun Microsystems, Inc. *NFS: Network File System Protocol Specification*, SRI Network Information Center, Menlo Park, CA (March, 1989).

UNIX and Internationalisation

Andrew Tune
adt@technix.oz

TechNIX Consulting Services Pty. Ltd.

ABSTRACT

This paper is a summary of recent developments in the area of the internationalisation of UNIX™. There's some very interesting work being done in the area at the moment, and some of it has some far-reaching technical implications. Internationalisation is viewed as one of the "hot topics" in the UNIX industry at present.

The interest in this area in the UNIX world is only a relatively recent phenomenon, but despite this, it is one of the areas in which UNIX is far ahead of most proprietary operating systems. The drive towards this area has come from three sectors: from Europe, which needs eight bit character support for non-ASCII characters; from the East, where multibyte character support is needed and is being pushed by people such as SIGMA; and from the vendors, some of whom have smelt a market which is currently under-serviced, and growing. (Actually, the response of vendors to the need for internationalisation has varied from strong and pro-active, on the part of DEC and HP, amongst others, to some who have been dragged along, kicking and screaming by their clients).

For Australian companies, especially systems integrators, vendors, and software houses, the international market should be of particular interest. Few companies in the Australian computer industry produce solely for the local market, and our proximity to the multibyte market makes the issue even more important.

The recent efforts of some of the major players in the area, including the vendors and the various standards committees, are covered, along with a little of the work of the SIGMA group, and the software industries in Asia, particularly China, Japan, Singapore, Korea, and Taiwan.

Finally, some of the technical issues are covered in some depth, including the questions of

- C support for multibyte characters
- Collating problems (and why half a job can be worse than none)
- Multi-byte character set standardisation
- Regular expressions and multibyte characters
- Hardware Support (how to go about typing a multibyte character)

™ UNIX is a Trademark of AT&T in the U.S. and other countries.

Introduction

Internationalisation is a relatively recent trend (there was virtually no activity before 1985), and is still quite restricted. The standard of localisation available in off-the-shelf software for the UNIX environment is still poor, but getting better. In recent years many of the standards bodies have put effort into this area, and after a few false starts, real, working implementations of useful software are emerging. The trend will continue as the market gets larger, and as more vendors become aware of the possibilities which internationalisation presents.

In order to resolve the inevitable ambiguities which will arise, the following definitions are offered:

Locale A combination of native language, local customs, and conventions for representation of data.

Internationalisation The capability within a computer system component (operating system, application, peripheral, etc) to support the language dependent requirements of many countries.

Localisation The process of tailoring a system component to fit the needs of a particular *locale.*

Locales

What constitutes a locale? Locales can be specified as a combination of the following:

- Character typing
- Collation
- Conventions for the representation of
  - Dates and Times
  - Numbers and Currency Amounts
  - Yes/No strings

Character Typing

Character typing used be one of those areas which was regarded as simple – no-one who has looked at `_ctype[]` could be left with any other impression. In the light of internationalisation, this can no longer be considered true. Consider the case of characters. Upper case is not difficult to distinguish from lower case in English[1]. However, many of the Asian languages are caseless, other languages provide characters which are the same in upper and lower case, and Arabic and Greek provide several lower-case versions of some letters – the choice amongst them to depend on the position within the word.

In terms of binary representation, the näive user will see one language's punctuation character as another languages's control character, and the issues of character typing of non-alphabetic languages (such as Chinese) are complex.

Collation

Collation (or sorting) is harder now, as well. The equivalence relationship between character weight and native character code which works so nicely for ASCII[2] is inadequate for almost any other language, although the problems are different.

[1] Nor is it difficult to convert from one case to the other. Compare this with French vs. Canadian French! Canadian French upshifts "é" to "É", whereas in French, "é" becomes simply "E".

[2] Well, almost. ASCII never was in real dictionary order. The discrepancy is in the area of case folding.

European Languages

For a start there are accents to worry about. Europeans are used to seeing accented characters collating with their non-accented versions.[3] Thus the sorting

épice
épicé
épicéa
épicène
épicentre

is correct in French, but not what some European codesets give you.

To further complicate things, there are three other special types of characters to worry about: two-to-one characters, one-to-two characters, and don't-care characters. An example of a two-to-one character is the Spanish "ch", which collates as a single character immediately after the "c". The Spanish "ll" is similar. The German "ß" is a one-to-two character, since it is supposed to collate as a "ss". Don't-care characters are characters such as hyphens, where "dis-able" and "disable" are meant to collate identically.

Additionally, some languages require a two-pass collating algorithm in order to deal with accenting.

Asian Languages

Collation in Asian languages ranges from relatively simple (e.g. Indonesian) to several orders of magnitude more complicated than for 8-bit languages. First of all, there are multibyte collation units to cope with. Secondly, there are several collation methods in use, for example:

- Binary
- Phonetic
- Radical content
- Stroke count
- Local custom
- User defined

Binary collation is, of course, almost meaningless[4]. The other methods are difficult, and, depending on the character size, can chew up quite a bit of memory and disk space[5]. But the real problem is determining the correct collation sequences. The collation sequence used, for example, in Chinese telephone books is, in descending order, Phonetic, Radical content, Stroke Priorities[6], and finally Binary.

There is, however, a different sequence used in dictionaries, another method used in older dictionaries, and a fourth used in lists of people's names[7].

Other Conventions

The differences in conventions used for the representation of numeric and monetary values are well known, and the various alternatives are pretty simple. They revolve around such things as what the currency symbol is, whether it precedes or follows the amount, how the amount is represented (the radix character and the thousands separator), and so on. Examples include DM1,000.00 (German), 1 000,00FF (French), and 1.000$00 (Portuguese).

Other conventions include the representations of date and time, as well as the names of the days of the week and the months of the year. In some countries, date and time are represented relative to local historical or astronomical events. In others, time may not be easy to relate to timezone information. Much of the Middle East and Africa, for instance, uses Solar Time[8]. In the east, times may well be expressed in terms

[3] This is generally, but not always true. In Finnish, the "å" character collates *after* "z"!

[4] There are actually government-provided standards, so it's not quite meaningless.

[5] The new **ISO 10646** pending standard is a 32 bit character set. So collation tables could be up to 16 Gbytes *each!*

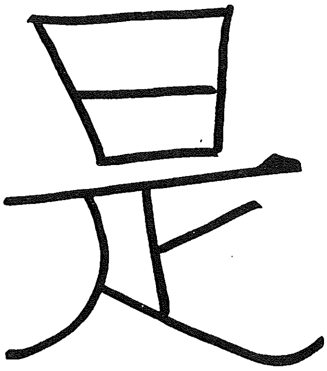
[6] In descending order: horizontal strokes, vertical strokes, left, then right slant strokes, and then others.

[7] In Japanese society, names are often sorted in descending order by importance!

[8] Also know as WYSIWYG time!

of Emperor/Era name and year.[9]

Even the issue of answering "Yes" or "No" to questions is not simple. There are, of course the obvious problems of having to realise that "o/n" means more to a French user than "y/n", but in Asian languages it's more difficult again. In Chinese there are a number of ways, but the most common ways for each are represented in any case by two-byte quantities! See Figure 1.

Character Number	Hex Value	Entry Keystrokes
2508	BBE1	AMYO
		

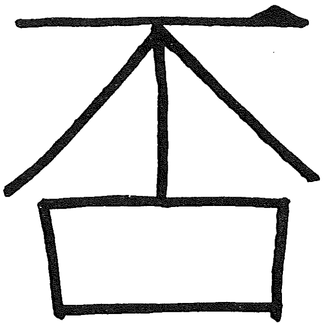
Character Number	Hex Value	Entry Keystrokes
0694	A8C5	MFR
		

Figure 1. "Yes" and "No" in Chinese

Technical Issues

The technical issues involved in the use of non-ASCII character sets are centered around the following areas:

- Code and Character sets
- Bytes per character
- Directionality
- Regular expressions

Code and Character Sets

A code set and a character set are not the same thing. A character set is just that – a set of characters which are used to make up the words of a language. A code set is a set of unambiguous rules which specify the representation of characters and a one-to-one mapping of binary codes to characters.

There are a number of standard codesets for eight bit languages, mostly being attempts to cover the European market in one go. The sixteen-bit codesets (and their market) are a little more complicated, and a number of standards exist *for each language* in the area. Beyond that, there are a number of larger codesets, and a number of compliant implementations available from, and supported by, various vendors.

[9] Hewlett-Packard is presently pressing X/OPEN to add Emperor/Era support in the X/OPEN Portability Guide, Issue 4 (XPG4).

There are basically two means of representing characters from different codesets in the same stream of bytes: single-shift and locking shift. Single shift is the use of a trigger to say "the next *n* bytes are from the other character set". An example of this is the Hewlett-Packard HP-15 character set, which represents a two-byte character as an initial byte with the top bit set, and a following byte with the top bit **not** set. Thus characters from HP-15 can be intermixed with ASCII characters, and the immediate context will always distinguish between the two. Hence the byte stream

```
<start> 00101011 01011101 10011101 00101101 01011101 <end>
```

can be unambiguously interpreted as two ASCII characters, followed by one two-byte character, followed by a final ASCII character. The byte before the current byte must always be accessible in order to distinguish between an ASCII character and the second half of a sixteen-bit character. This is, of course, no longer possible when eight-bit characters are mixed with sixteen-bit, and the second method, locking shift, is often used here. The meaning is obvious:

```
... <startseq> chars from new code set <endseq> ...
```

The obvious problem is the context sensitivity of this system – an arbitrary amount of context is required in order to determine codeset.

The main players at the moment are:[10]

- **ISO 8859/1** – All characters and symbols needed for Western European languages – the full name is ISO 8-bit Single Byte Coded Graphic Set Part 1: Latin Alphabet No 1. Also known as **DIS 8859/1**.
- **EUC** – Extended UNIX Codeset from AT&T – this is a codeset, or more accurately, a codeset template, consisting of a Primary and three Supplementary Codesets which is based on ISO 2022. EUC can support JIS-C6226.
- **JIS-C6226** – Another ISO standard for supporting Japanese – a sixteen-bit codeset which contains both the *hiragana* and *katakana* alphabets and about 7000 kanji ideograms.
- **Shift JIS** – Defacto standard for PCs in Japan.
- **ISO 10646** – Single multibyte code set which simultaneously supports all the characters of all the languages in the world.

AT&T's JAE codeset, (EUC defined for AT&T's JAE product), is currently in use by just over half of the UNIX vendors in Japan. It's based on ISO 2022 and JIS-C6226. However, despite its wide acceptance, there are problems with EUC-based codesets.[11] In particular, they cannot support:

- The **Shift-JIS** codeset (see above)
- The packed Hangul codeset used in Korea
- The **Big 5** codeset used in ROC and Taiwan
- The Chinese Code for Data Communications used in ROC

Directionality

Most languages, fortunately, work in what we regard as the "conventional" manner, i.e. from left to right, from the top of the page to the bottom. This is not the case for all languages, in particular the MEA (Middle East and African) languages. Obviously the difficulty of supporting languages which work bottom to top, right to left, is increased by the dearth of hardware support (how many terminals will wrap from the left margin to the right margin of the previous line?), but the issue appears in many areas. An example from the user-interface area is the use of pull-right menus: pull-left menus are apparently more intuitive for those who write right-to-left.

[10] Note that ISO 10646 is still under development (i.e. not yet an ISO standard), and it remains to be seen how many vendors will be willing to pay the costs associated with its increased functionality.

[11] Despite these and other problems, AT&T has been lobbying X/OPEN for acceptance of EUC as the recommended codeset for transmission and internal use in conforming systems. At the time of writing, they have been unsuccessful, and X/OPEN appears to be leaning more strongly toward a position of codeset independence, even to the point of removing current references to ASCII and ISO 8859/1.

Regular Expressions

Writing regular expression code for multibyte languages is not easy, but for better (or for worse) the X/OPEN specification now includes some useful extensions to regular expressions[12], including the concept of *collation elements*, *equivalence classes*, and *character classes*[13].

Concept	Representation	Meaning
Collation Element	`[.ch.]`	A multi-character collating element, such as the German "ss" and the Spanish "ch".
Equivalence Class	`[=a=]`	A set of collating elements whose primary collation weight is the same as the specified collation element. For instance, `[=a=]` will match all forms of the character "a", such as "a", "A", à, á, ä, and so on.
Character Class	`[:class:]`	The set of characters belonging to the ctype(3) class *class* of characters. Valid alternatives are **alpha, upper, lower, digit, alnum, space, print, punct, graph, cntrl,** and **xdigit**.

Thus, for example, if "a", "á", and "A" all belong to the same equivalence class, the regexp:

```
"^[=a=][:alpha:]+"
```

will match any one of these characters, followed by an alphabetic character, all at the start of a line.

The Solution

The problems associated with producing software which is internationalised, i.e. easily localisable, can be broken down into the following:

- Working out where you are
- Communicating information *to* the user
- Getting information *from* the user
- Internal processing
- Producing externally comprehensible output
- Dealing with multiple languages

Finding The Current Locale

Before anything else can be done, the current effective locale has to be established. This may be nothing to do with the physical location, and is determined, according to X/OPEN and POSIX[14], by the contents of your **LANG** environment variable; nothing new there. But it's not as simple as that. Where appropriate, **LANG** can take the representation:

```
LANG=language[_territory][.codeset]
```

[12] Regular expressions which provide these facilities are known as *Internationalised Regular Expressions*, to distinguish them from *Simple* and *Extended Regular Expressions*

[13] The usefulness of character classes is obviously limited in those languages in which some of the classes make no sense, such as caseless languages.

[14] Published by, and available from the IEEE.

Values of LANG, somewhat sadly, are often given in English, although the X/OPEN standard specifically leaves this to the implementors. It seems contrary to the spirit of internationalisation to have German-speaking users have to specify LANG=german, when LANG=deutsch is so little extra work[15], although this does of course bring up the question of whether the French user should be able to say LANG=allemande! So much for a perfect world!

On top of this, Hewlett-Packard is currently implementing support for a LANGOPTS environment variable, for specifying latin versus non-latin mode, and keyboard versus screen data order. Whether this will be adopted by anyone else remains to be seen.

For specifying locale details to a finer resolution, aspects of the locale can be specified individually. The following environment variables:

- LC_COLLATE – collation sequence
- LC_TYPE – character typing (à la ctype(3))
- LC_MONETARY – monetary conventions
- LC_NUMERIC – numeric representation conventions
- LC_TIME – time and date representations

are all specified as follows:

```
LC_...=language[_territory][.codeset][@modifier]
```

For example, to interact with a system in Dutch, but sort German files, using unfolded collation, the user could specify:

```
LANG=dutch
LC_COLLATE=german@nofold
```

At runtime, these values are bound to a process's locale by the setlocale(3) function, which can be found in XPG3[16], POSIX.1, FIPS 151-1[17], and the ANSI C standard[18]:

```
#include <locale.h>

char *setlocale(category, locale)
const int category;
const char *locale;
```

The category is one of the LC_... alternatives, or LC_ALL for the whole of the locale information. There are three alternatives for the locale name. The NULL string allows defaulting to the settings of the environment variables, and will be the most common. The argument "C"[19] enforces use of the default values[20], and legitimate locale names do as expected[21]. For almost all purposes the use of

```
setlocale(LC_ALL, "");
```

will suffice. The application must remember its locale – there seems to be something of a hole in the standards in that there is no specified way to *get* the current locale.

[15] Ideally, it should be SPRACHE=deutsch, but that would leave the application program in a somewhat difficult position!

[16] X/OPEN Portability Guide Issue 3

[17] FIPS stands for Federal Information Processing Standard, as defined by the National Institute of Standards and Technology (NIST). FIPS 151 has been approved by all the appropriate U.S. government bodies, and FIPS 151-1 is coming out sometime late this year.

[18] X3J11, X3.159

[19] The proponents of Ada are apparently complaining about the choice of "C", so this may change. Watch this space...

[20] The default locale cultural data contains some surprises, in particular the absence of a currency string.

[21] Timezone information can be more finely specified again, to cope with local rules.

User Communication

The issue of user communication is of primary importance. Having things such as:

```
        printf("Customer number is invalid.  Please re-enter\n");
```

in source not only makes localisation very difficult, but in the international environment, it's somewhat näive to expect that the eventual user will understand you. Asking questions and expecting "Y" or "N" as the answer falls into the same category. The thought of extraction, translation, and replacement is rendered unattractive by the difficulties of extraction (it's not easy to do so programmatically), and the problems with maintenance. And yet it's universally recognised that a solution to the localisation problem which requires recompilation of the source of the program concerned is no solution at all.

The solution is almost obvious – keep the messages in files known as *Message Catalogues*[22]. This was a solution developed by Hewlett-Packard, and subsequently adopted by X/OPEN. **catopen()** opens a message catalogue and returns a catalog descriptor (of type **nl_catd**). Subsequently, **catgetmsg()** and **catgets()** are used to extract the messages appropriate to the current locale, and **catclose()** to close the catalogue.

Not all user communication is in the form of set messages. POSIX or X/OPEN conforming systems nowadays produce, among other things, locale-appropriate messages from **perror()**.

Support has been added to **printf()** and family, allowing conversions to be applied to the *n*th argument, rather than to the next, as was traditional[23]. This is done by using **"%*n*$d"** (for instance) instead of **%d**. This enables support of such things as date and time represntations, whose ordering changes according to locale.

A new routine, **strftime(3)**, is specified by XPG3 and ANSI C. It converts the contents of a **tm** structure (as returned by **ctime(3)**) to a formatted string. This is done according to the program's locale, passed parameters, and the contents of the **TZ** environment variable. Thus with the use of the same format string, a date and time of the format:

```
Sunday, July 3rd, 10:02
```

would be produced for American usage, whereas for German usage, the output would be:

```
Sonntag, 3 Juli, 10:02
```

This removes from the programmer many of the headaches associated with this type of low level and probably foreign date/time formatting.

For conversion of ASCII strings to doubles, **strtod(3)** has been enhanced to recognise the appropriate radix character[24] for the locale, and the locale's appropriate definition of "white space".

Internal Processing

In terms of the normal day to day processing of strings, multibyte characters throw something of a spanner in the works. In processing (for instance), strings built from a sixteen-bit character set, a string seven bytes long makes little sense. The last byte is presumably half a character. The problem is further complicated by the presence of character sets which are four and three-byte. Enhancements to the old, faithful **string(3)** library have been specified by XPG2, XPG3, POSIX.1, FIPS 151-1, and ANSI C, to provide support in this area. There is a slight disappointment here though – not all of these support multibyte character sets. This is helpful in a sense, one expects by now that **strlen()** will return the number of **bytes** in a string, not half, a third, or a quarter of that number for a multibyte character set.

Day to day processing for most applications will remain the same – fortunately no-one seriously expects character sets which include NULL bytes to be used successfully in the UNIX and C environment! Much of the difficulty is, as mentioned, in communicating with the user – reading input and producing output. Strings of bytes, no matter what code set they are, can still be treated as strings of bytes provided a little thought is given to them.

[22] A misnomer, really. These can be used to store many things other than just messages.

[23] This is supported by SVID, XPG2, XPG3, POSIX.1, FIPS 151-1, and ANSI C.

[24] But sadly, not the appropriate thousands separator.

Wide Characters

The concept of a wide character as discussed here is that first introduced in ANSI C. A multibyte character consists of one or more bytes that represent a "whole" character. A wide character is composed of a fixed number of bytes (type of `wchar_t`) whose value can represent any value in a character encoding. It's much more convenient from the programmer's point of view, in dealing with characters internally, if they can be viewed as being of a fixed width.
The relevant routines are as follows; note that their behaviour depends on the `LC_CTYPE` category of the current locale.

```
#include <stdlib.h>
int mblen(const char *s, size_t n);

int mbtowc(wchar_t *pwc, const char *s, size_t n);

int wctomb(char *s, wchar_t wchar);

size_t mbstowcs(wchar_t *pwcs, const char *s, size_t n);

size_t wcstombs(char *s, const wchar_t *pwcs, size_t n);
```

The `mblen()` function returns the number of bytes in the multibyte character pointed to by `s` if the next `n` or fewer bytes form a valid multibyte character. The `mbtowc()` function converts a multibyte character pointed to by `s`, and up to `n` characters long, into a wide character, pointed to by `pwc`. `wctomb()` performs the reverse operation, and `mbstowcs()` and `wbstombs()` perform similar operations on strings.

The relevant header files to be aware of are `<stddef.h>`, which defines `wchar_t`, which is an integral type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales; `<limits.h>`, which defines `MB_LEN_MAX`, which is the maximum number of bytes in a multibyte character for any supported locale; and `<stdlib.h>`, which again defines the type `wchar_t`, as well as `MB_CUR_MAX`, which is the maximum number of bytes in a multibyte character for the extended character set specified by the current locale. This value will never be greater than `MB_LEN_MAX`.

Producing Externally Comprehensible Output

The remaining problem in using non-ASCII code and character sets is the difficulty in communicating externally. How do you write tapes? What will other people be able to understand?

There is an X/OPEN-specified utility called `iconv(1)`, which exists precisely for the purpose of fixing this problem.

```
iconv -f fromcode -t tocode [ file ... ]
```

converts input (from `file` if specified, otherwise `stdin`), from the `fromcode` codeset to the `tocode` codeset.

The other problem in this area is the problem of what to use as a portable character set. The POSIX.1 Portable Filename character set[25] seems widely agreed upon as suitable for this purpose, although there are some obvious further restrictions that would need to be put in place for portability to other common environments.

Dealing with Multiple Languages

The question of dealing with multiple languages and their attendant problems is really part of the problem, more than part of the solution. On the first level, there is the difficulty of dealing with German data if you speak Chinese. The interaction with the system itself must be in Chinese, but the treatment of the data and its representation must be all in German. The concept of *User Languages* and *Data Languages* has been proposed, and seems to solve the problem reasonably well.

[25] Which is basically "A" – "Z", "a" – "z", "0" – "9", ".", "-", and "_".

But there is a whole class of problems more difficult than this – the question of multiple language data access. Take the situation of two users of the same machine, accessing an ISAM file keyed on a string value. One user is using German and wishes to insert a record. The other is using French and wished to retrieve a range of records. The potential for surprise is obvious, but *all* solutions so far seem to involve either unexpected behaviour, or significant overhead.

Why Bother?

The final question is, of course, "Is it worth all the effort?" The answer has got to be "Yes!" For too long now, the computer industry as a whole has continued along the straight and traditional but nonetheless very narrow path of ASCII and English. The majority of the world does not speak English, and ASCII is woefully inadequate for most languages other than English. Many of the technical difficulties have already been addressed – others are being dealt with at present. It's a new area, and not without its technical challenges. But the amount of investment required to internationalise a product, and, more importantly, to lay the groundwork for the incorporation of internationalisation techniques in future development is small compared to the potential gains. A few things about the Asian market are worth pointing out here.

Japan

The Japanese market has been recognised from the start as the obvious Eastern market with the most potential, at least in the short term. Of particular interest is the SIGMA project. SIGMA stands for Software Industrialised Generator and Maintenance Aids. The aim of the SIGMA project is to produce "high quality software" in "great quantities" by making software production more efficient and rational. To this end, the official SIGMA workstation and the SIGMA OS have been specified[26].

Some points of interest:

- Funding: 25 billion Yen ($A 235 Million at the time of writing).
- Several hundred corporate members, including foreign and local computer vendors, software vendors and other manufacturers.
- Current schedules call for commercial operation in April of 1990.
- SIGMA has specified a code set, called "UJIS", which uses AT&T's EUC code set encoding scheme, but does **not** define use of the third plane.
- SIGMA specifications translated to English should be available "soon".
- SIGMA has defined the following extended internationalization functions:
  - `getwc()`, `getwchar()`, `fgetwc()`, `ungetwc()`, `putwc()`, `putwchar()`.
  - `fputwc()`, `getws()`, `fgetws()`, `putws()`, `fputws()`.
  - Extensions to `printf()` and `scanf()`.
  - New "japanized" functions `jconv` (like iconv), `jcode()` (like toupper), `jctype()` (like isalpha).

SIGMA has specified one of the primary deficiencies of UNIX to be the lack of support for Japanese language processing, and is working at present to address this.

Of great interest to software houses should be the fact that in Japan, customised software comprises about 94% of the market, as opposed to the corresponding figure in the US of 35%.

China

The Chinese market is almost non-existant, but, of course, has the potential to grow to be huge. It is, however, starting from behind the eight-ball, since there was no computer science activity in China from 1966 to 1976, due to the cultural revolution. However, there is now a national plan in place for software development, as well as national standards for software development, documentation, requirements analysis, project management, and cost control.

[26] Curiously, the SIGMA OS has been described as UNIX SVR2.0 plus those parts of BSD "which are beneficial to software development" – basically the networking.

Recent developments in China include:

- First National Software Technology Exchange Fair held March 1988.
- First Joint Conference for Computer Applications held February 1988.
- First National Software Working Conference held December 1988.

In summary, it would have been fair to say that China was a fair way behind the rest of the world, but advancing quickly. The repercussions of recent events in China are at this point unknown – it remains to be seen how much harm has been done.

Singapore

The computer industry in Singapore is recognised by the government as being of great strategic importance to the economy. As a result of this the government is offering all sorts of benefits to both local and foreign countries which set up software divisions in Singapore (including tax holidays of up to ten years!).

Several large vendors have set up operations in Singapore, including Data General, Ericsson, Hewlett-Packard, IBM, NEC, and Nixdorf. The work being done there includes (not surprisingly), the "asianisation" of software, development of multibyte software and localisation efforts, as well as UNIX system software development, office automation, networking, communications, knowledge-based systems, and real-time systems. Private-sector investment in software R&D in Singapore is expected to grow by $US 100 million per year over the next three years.

Taiwan

The Taiwanese software industry is another example of an industry which is small, but growing fast. Most products produced so far are for PCs (hardly surprising), but the number of UNIX products is increasing. The Taiwanese Institute for Information Industry has produced a series of CASE tools for the UNIX environment, which are marketed under the unlikely name of "Kanga Tools".

The government has launched a project known as "SEED" (Software Engineering Environment Development), which aims to offer local industry not only consulting services and standardisation, but also the specification of workstations for Taiwanese use.[27] The workstation plans at present are for 80386-based systems running SYSV and X. Current areas of interest in research include AI and expert systems (with an emphasis on such things as image analysis, optical character recognition, Chinese syntax analysis, etc), software automation, CAD, and CAE.

Korea

Industry in Korea is small and weak, but growth is very strong (40% p.a.) at present. As is not unusual in Asian countries, most activity taking place in nationally funded projects. It is interesting to note that in 1986, 45% of software demands were met by foreign products, despite lack of support for Hangul processing.

The Koreans are hard at work on automated language translation:

- Korean → Japanese – working.
- Korean → English – in place soon.
- English → Korean – under development.

There are problems in finding appropriate personnel in Korea, due to the high social value placed on being a teacher. This makes it very difficult to get people to leave university teaching posts and enter private industry.

Europe

The European market is of course huge, and is accustomed to using English-language products. This does not mean that there is no demand for localised products. The size of the European localised market has not to my knowledge been reliably estimated, but is certain to be large and growing, and will be stronger in some countries (Italy, for example) than in others.

[27] Shades of SIGMA!

What the Vendors are Doing

Many vendors are now actively involved in internationalisation, and standards meetings routinely include representatives from AT&T, Bull, DEC, Fujitsu, HP/Apollo, Hitachi, IBM, NCR, NEC, Nixdorf, Olivetti, Prime, Siemens, Sun, Toshiba,[28] Unisys, and UNIX International. In addition, the OSF members meet separately over internationalisation. In terms of individual contributions:

- Hewlett-Packard stands out as having had their system form the base of the original X/OPEN system, and having put forward new proposals (once again to X/OPEN) for the handling of multibyte characters. HP also has the Emperor/Era proposal before X/OPEN.
- IBM has implemented wide char and multibyte libraries, and offered these to OSF.
- Sun is working on a 16 bit universal code set which they refer to as "Uni-code". In order to squash this into sixteen bits there will have to be some rationalisation to only "frequently-used" characters. They are trying to get one **ISO 10646** plane (possibly the first) reserved for their codeset.
- Apple is working on Sun with the above, and seems to be quite serious about the area.
- AT&T has stated that SYSV.4 will be ANSI C, POSIX.1 and XPG3 conformant. POSIX.2 is scheduled for V.4.1, in the third quarter of 1990. System V.4 supports a "number" of multibyte commands and SIGMA (not truly multibyte) regular expressions.
- Fujitsu has (understandably) expressed concern about the multibyte market and is quite active in the area.
- DEC appears to have significant concerns in the area of directionality, and wants the facility for applications to be written in a directionality-independent manner. AT&T appears to be approaching this from a different direction.

Hardware Support

Some of the difficulties of hardware support have already been mentioned, and the list is by no means complete. The eight-bit area has been handled well for some years, and now several manufacturers are producing terminals and printers which support multibyte characters. This type of hardware is especially easy to get in the PC market, and the use of PCs to interface to larger systems because of the internationalisation support offered by PC hardware will probably continue.

The practical difficulty is that of typing a multibyte character. Provided you can understand them and your hardware supports them, they're easy enough to read, but normal keyboards don't cope at all well with non-alphabetic languages. A number of schemes have been devised, but none of them is totally satisfactory. Systems typically allow the specification of a multibyte character by entering the radicals one by one (since the radicals will, typically, fit on a keyboard), by entry of the standard hexadecimal code for the character, or by selection from a menu[29] based on statistical distribution of characters. The mechanics of this are invisible to the application, of course, but make an interesting challenge for the hardware designer, and the writer of the firmware![30]

The Revolution So Far

One of the interesting things about internationalisation is that it's a quiet revolution – for some years now it's been happening "under our feet", so to speak. It's possible right now to go out and buy an internationalised UNIX system from a number of vendors, and there are also several new releases waiting in the wings.

[28] Mainly through their involvement with the SIGMA project.

[29] Normally a small, status-line menu.

[30] One of this issues which has been skirted here is the right place for implementation. This has been substantially decided by the standards, although there are still places where the "put it in the kernel" crowd and the "keep the kernel small" people will have to fight it out. One of these is the area of support for different directionality.

The effect on the normal user isn't great, but if you alter your **LANG** environent variable, you may find some surprises in store:

```
% ls -l fred
-r--r--r--   1 adt     tech     11861 Jul  3 17:45 fred
% LANG=french
% ls -l fred
-r--r--r--   1 adt     tech     11861 juil  3 17:45 fred
% date
lun 03 juil 1989 17h45 58
% LANG=german
% date
Mo., 03. Juli 1989 17:46:04 EST
% LANG=finnish date
03-07-1989 17:46:42
```

X/OPEN conformant systems have a minimum of sixty-nine fully 8-bit internationalised utilities, and XPG4 is going to specify more, as well as multibyte internationalisation requirements.

Conclusion

Internationalisation is one of the areas of greatest flux and greatest interest in the UNIX world at present. Many of the vendors are working hard, and often working closely with standards groups to provide open, flexible, and powerful systems for the writing of internationalised software.

The result will hopefully be a better UNIX system, especially for end-users, who have in the past always been trapped in the mould of ASCII and English, whether they liked it or not.

The next several years should see the emergence of an almost completely new look for software from the perspective of the non-English speaker. There are efforts to localise significant pieces of software under way at present, and this trend will only get stronger as more and more software is built with localisation in mind, and as more vendors realise that the incremental investment required is small in comparison to the gain to be made.

From the technical view there are some challenges, and the size of the challenges depends on your target system, since some vendors have done most of the work already. Beyond that, though, there are still difficulties and uncertainties. Issues like support for **ISO 10646** are still be be resolved, and it will be interesting to see what happens. Whatever happens, however, from the user's point of view, as well as from the programmer's, UNIX will not look the same. It seems we've finally reached the stage where:

```
main()
{
        printf("hello, world\n");
}
```

isn't good enough any more!

UNIX and Artificial Intelligence in the ´90s

James Catlett
Basser Deparment of Computer Science
Sydney University

Abstract

Artificial Intelligence is a collection of computing technologies, some of which offer a treasure-trove of opportunities for innovative applications software. Getting to this treasure will be an important goal for most large companies in the '90s, and a few have reached some. But many captains have set sail into the newly-charted waters of AI with a ragged crew, a faulty course, and poor equipment. I hope this talk will help the audience reduce the risk of shipwreck in future AI projects, especially for those sailing in a UNIX-powered ship. I will address four questions: what is AI, where is it going commercially, what do we need to make it work, and is UNIX the right choice as an AI software environment.

First, what is AI? This will be answered both in terms of its fields of research, and its areas of commercial application. I will concentrate on its most popular field, Expert Systems, although other such as vision, robotics, and natural language processing will be briefly addressed.

Secondly, where is AI as a commercial technology up to, and where is it going? Many large companies are reporting big successes with AI, while specialist suppliers of AI software tools are having difficulty keeping their heads above water. This shows similarities with the markets for other software productivity tools, such as 4GLs. I will broadly characterise the currently available AI tools and their suppliers, with emphasis on their availability on UNIX versus other platforms. The question of where the customers are and what applications they have built will be answered with respect to both Australia and the world. Applications are found in almost every industry and country, but the numbers lean towards each nation's high-leverage industries, such as manufacturing in Japan, and banking and mining in Australia. I will sketch a few success stories, and characterise the attributes of the problems that make them suitable for the most popular AI techniques.

Finally, what can companies do to make the sailing through the seas of AI smoother, and is UNIX the right ship to choose? Obviously the answer depends on the particular destination, but in general UNIX is looking good. Much of the range of off-the-shelf AI software is already available on UNIX. Two important factors for most AI projects, integration and portability, are well satisfied by UNIX's capabilities across a diverse range of hardware and software products. Rapid prototyping, a now watchword in AI and Expert Systems, has always been a strength of UNIX developers. One area where UNIX's offering has only more recently come up to speed is the user interface (for novices) and graphics. There is also a favourable coincidence for UNIX in the demographics of new graduates, since they now commonly have some education in both AI and UNIX. Unfortunately the shortfall of very skilled people in both these areas remains a major bottleneck.

As to the question of whether anyone will get fired for buying UNIX for AI, I couldn't answer no because AI will remain a high-risk area for some time, so even the lowest-risk choice can still blow up. But failure to make adequate preparations for the AI opportunities of the 90's could have much greater costs. But as the saying goes, "chance favours the prepared mind", and I think it will favour the well-prepared software environment too.

Storage and Retrieval Methods for an Interactive Spelling Corrector

Sunil K Das and Philip M Sleat

City University London
Computer Science Department
EC1V 0HB, United Kingdom

Email: {sunil,phils}@cs.city.ac.uk

Abstract

The *size* of a data file and its *speed* of access are two important properties to be considered when designing information processing applications. The design of a software tool to perform interactive spelling correction necessitated an investigation into various storage methods for large data files and a comparison of retrieval techniques for individual records. Primary goals included efficiency of dictionary storage, efficiency of dictionary access and ease of use. The most appropriate data structures were adopted for the compression of a dictionary file. Techniques were developed to reduce the size of file even further, thus enabling it to be completely stored in main memory. This resulted in very fast record access when compared to dictionary files which are so large that they have to be stored on disc. The compacted dictionary was used to implement a user-friendly, spelling corrector for UNIX and PC based systems. Careful consideration was given to the Human Computer Interface, the screen display and user input. Provision was made for a user to create an *addenda* dictionary for proper names and acronyms.

1 Introduction

The UNIX software tool *spell* compares words from the named text file with those found in a spelling list. The output is those words which are not in the dictionary and hence assumed to be misspelled. Thus *spell* is a non-interactive, spelling *checker* which can prove cumbersome to use. The user must take the generated list of misspelled words and use a standard text editor to locate each one in the text file to correct them. The spelling checker does not supply a list of suggested correct spellings; if the user is unsure of the correct spelling of a word, it must be looked up manually.

A useful tool to help with document preparation within the UNIX programming environment would be an interactive, spelling *corrector.* Amongst other facilities, the tool would open the named text file, enable simple access to misspelled words (typically via cursor keys), offer suggestions for correct spellings, permit alterations to the file and allow the adding of words to a user's *addenda* dictionary file.

This paper describes the development and implementation of a spelling corrector for UNIX and PC based systems which incorporated the following tasks:

- investigate and extend tree storage techniques to facilitate the compression of a dictionary file and speed up word access; and
- study the design of human-computer interfaces [Shneiderman 1987] to produce a 'user-friendly', interactive spelling corrector using the compressed dictionary tree.

2 The Data Structures

Files for interactive, information processing applications are often large. Ideally, a file should hold data in as compact form as possible. It is desirable that the structure of a file helps efficient searching to extract pertinent data. Access times should be as small as possible so that a given record in a file may be retrieved quickly. Moreover, its organisation should facilitate adding new data. Very often, a trade-off is made between the compactness of a data file and the speed with which records can be accessed. Generally highly compact data files have slow access times; the compressed nature of the file making searching algorithms complicated.

Linear, binary and hashing techniques have been investigated [Knuth 1973] for storing and accessing data files. Of these, only binary techniques use the underlying structure of the data file to improve access times. However, whilst the advantages of record ordering is exploited, there is no compression of the data file and only small files can be in main memory for searching. If a large file is to be accessed, the search would have to take place externally or be hampered by swapping parts of the file into main memory for scrutiny. A method based upon tree structures [Sussenguth 1963] out-performs other techniques in terms of data file economy of size and speed of access [Stanfel 1970]. It permits a more compact file, thus making it possible to store the entire data structure in main memory which yields very quick file access.

2.1 Tree-structured Files

Figure 1 introduces the tree terminology [Iverson 1962] used in this paper. **A** to **K** represent *nodes*, with **A** as the *root.* **A-E-G-H** is a *path* connected by three *branches* and therefore, has a *length* of three. **C** and **D** are examples of a *leaf.* The *filial* set of **E** is the set of nodes one path length away from **E**, namely the three nodes **F**, **G** and **I**.

```
A ___ B ___ C
|
|____ D
|
|____ E ___ F
      |
      |____ G ___ H
      |
      |____ I ___ J
            |
            |____ K
```

Figure 1: Tree Terminology

Compression of the 'flat' data file of Figure 2 into the tree-structured file Figure 3 can be illustrated by using information about some fictitious students. The primary key to each record is the course code — CS stands for Computer Science. Successive keys are a student's year of study, surname, initials and other information.

```
.
.
.
CS,1,Kernighan,BW,....information for Brian Kernighan
CS,1,Lesk,ME,.........information for Michael Lesk
CS,2,McIlroy,J,.......information for Joe McIlroy
CS,2,McIlroy,MD,......information for Doug McIlroy
CS,3,Thompson,K,......information for Ken Thompson
CS,3,Ritchie,DM,......information for Dennis Ritchie
.
.
.
```

Figure 2: A Data File of Student Records

In the tree-structured file, the course code is placed at the root of the tree. The second level of nodes represents the year of study. Student names and initials are on the next two levels with information on each student stored at a leaf of the tree.

```
CS __ 1 ___ Kernighan ___ BW ___ information for Brian Kernighan
|     |
|     |____ Lesk ________ ME ___ information for Michael Lesk
|
|____ 2 ___ McIlroy _____ J ____ information for Joe McIlroy
|                 |
|                 |_________ MD ___ information for Doug McIlroy
|
|____ 3 ___ Thompson ____ K  ___ information for Ken Thompson
      |
      |____ Ritchie _____ DM ___ information for Dennis Ritchie
```

Figure 3: A Tree-structured File of Student Records

The tree representation of student records is more compact than the 'flat' data file since the same information is never stored twice or more. For example, the name 'McIlroy' is stored once only in the tree.

A search of the tree-structured file is straight forward. It begins by finding the tree with the root that matches the primary key. The filial set of this node is then scanned linearly to find the node which matches the secondary key. This process is repeated for all keys until leaf information is reached. Tree searching in this manner can be seen to be very fast.

2.2 Dictionary Trees

How can tree structures be used to store an English dictionary? Figure 4 shows a sample of a typical dictionary file.

```
I
IMMACULATE
IMMENSE
IMMIGRANT
IN
INSERT
INSERTED
INSERTING
```

Figure 4: Part of a Dictionary File

A convenient way to store an English dictionary using a tree structure is to have each letter of a word at a different node. Each of these letters then becomes one of the keys on which the words are scanned when checking if a given word is in the dictionary.

There is a subset of nodes which can be both at the end of a word and part way through a word. For example in the dictionary tree of Figure 5, the node 'T' in the word 'INSERT' belongs to this subset. The word 'INSERT' is both a word in its own right, as well as a prefix for the words 'INSERTED' and 'INSERTING'. Thus, to represent the end of a word diagrammatically, an asterisk has been used to flag the terminal letter of a word. All leaf nodes represent the end of a word.

```
I*_ M _ M _ A _ C _ U _ L _ A _ T _ E*
|       |
|       |__ E _ N _ S _ E*
|       |
|       |__ I _ G _ R _ A _ N _ T*
|
|__ N*_ S _ E _ R _ T*_ E _ D*
                    |
                    |__ I _ N _ G*
```

Figure 5: Dictionary Tree Representation

The word 'IMMIGRANT' can be used to demonstrate a tree search to see if it is in the dictionary and hence spelled correctly. Starting at the root 'I' the filial set 'M' and 'N' would be scanned. Since 'M' is the first letter in the filial set, this node's filial set is scanned looking for a match to the third letter in 'IMMIGRANT', i.e. 'M'. The filial set of this 'M' node is 'A', 'E' and 'I' so the 'I' node is eventually located. From this point, the letters in 'IMMIGRANT' can be checked off from the 'G' onwards until the leaf at the end of this path is reached.

If the word being searched for is misspelled as 'IMJIGRANT', it would be found that 'J' was not in the filial set of the 'M' node. Hence, the word is not in the dictionary tree and is potentially misspelled or requires adding to the addenda dictionary file.

2.3 Common Word Endings

In many tree structures, there are common subtrees which are stored many times, thus wasting space. For an English language dictionary tree, these subtrees manifest themselves as common word endings. For example, many English verbs can end with the suffix 'ING'. Other common word endings are 'ATE', 'TION', 'ED'. All of these common word endings are stored at the end of a path in the tree representation of a dictionary. The space occupied by a dictionary tree can be considerably reduced by replacing common word endings with a pointer into some predefined table as displayed in Figure 6.

```
I*_ M _ M _ A _ C _ U _ L _ #3        Common Word
|       |                             Ending Table
|       |__ E _ N _ S _ E*            #1 ANT
|       |                               ...
|       |__ I _ G _ R _ #1            #3 ATE
|                                       ...
|__ N*_ S _ E _ R _ T*_ #7            #6 ING
                    |                 #7 ED
                    |__ #6              ...
```

Figure 6: Common Word Ending Table

The algorithm of Figure 7 was devised to build up a frequency table of word endings of up to four characters in length from the 'flat' dictionary file. It could be used going forward in the dictionary file followed by going backward in an attempt not to 'lose' a common word ending.

```
for (each word in the dictionary)
    if (word length > 4 characters)
       /* x is the character length of the word ending */
       for (x = 2; x <= 4; x++)
          if (word ending is in frequency table)
             frequency count ++;
          else if (there is space in the table) {
             insert word ending;
             frequency count := 1;
          } else {
             locate word ending with lowest frequency count;
             use this location for the new word ending;
             frequency count := 1;
       };
```

Figure 7: Building a Frequency Table of Common Word Endings

Having generated a frequency table of common word endings, the most popular four letter word endings were chosen, followed by three and two letter word endings to give the best compression of the final dictionary tree. The frequencies for the three and two letter word endings had to be adjusted dynamically to reflect the fact that we had chosen some four letter word endings. For example, consider the word endings 'TING', 'ING' and 'NG'. Having selected 'TING' to enter into the word ending table, the frequencies of 'ING' and 'NG' had to be adjusted because of the overlap.

A further possible source of space saving would occur if there are more common word beginnings than there are common word endings, and the words were stored backwards. A brief investigation into the English language established that there are more common word endings than beginnings. For the English language, it is sensible to store the words forward.

2.4 Filial Set Node Ordering

A major source of inefficiency in tree-based dictionary storage is the need to search the filial set of a node in a linear manner to see if a given character is included. Using the principle of *Zipfian* distribution on the nodes of filial sets would speed up this search. Zipfian distribution arranges the nodes of a filial set in order of greatest frequency. For example, suppose the most common and uncommon letters following the letter 'A' in the English language are 'N' and 'J', respectively. The node 'N' would be the first node in the filial set of 'A' so that it would be checked first. The least frequently occurring letter 'J' will be stored last in the filial set of 'A'. In this way, the most common letters are checked earliest to speed up the process of searching the filial set. For example in Figure 8, a sorted subset of words is shown with their corresponding Zipfian sorting. The ordering is caused because more words begin with 'AB' than 'AA' and more words begin with 'ABO' than 'ABL'.

```
A                                   A
AARDVARK                            ABOUND
AARDVARKS                           ABOUNDED
ABLE         would generate         ABOUT
ABOUND                              ABLE
ABOUNDED                            AARDVARK
ABOUT                               AARDVARKS
```

Figure 8: Zipfian Distribution

However, there is an inherent disadvantage associated with Zipfian distribution. If the filial set is stored alphabetically and the current letter in the filial set is higher in the collating sequence than the letter being searched for, then it is known that the letter is not in the filial set. With a filial set ordered according to Zipfian distribution, a search to the end of the filial set is necessary to discover that a given letter is not included. Therefore, using Zipfian distribution will be slower than alphabetic ordering for checking misspelled words. For correctly spelled words, a Zipfian based dictionary tree can be searched faster. This is acceptable however, because locating a misspelled word will occur much less often than locating a correctly spelled word.

An algorithm was devised to sort an English dictionary according to a Zipfian distribution.

1. Construct a dictionary tree from the dictionary file. As each node in the tree is passed through, increment a count associated with the node. This count is the frequency of use of the node.

2. Visit each node in the tree in turn (using a pre-order search). Sort the filial sets of each node according to the frequency counts.

3. Perform another pre-order search of the whole tree to re-construct the words in the tree. Write these words out to the Zipfian sorted dictionary file.

2.5 Inherent Spelling Correction

There is no inherent spelling correction when using a dictionary file. It is possible to determine whether or not a word is in the dictionary, but there is no 'pointer' to where the correctly spelled word of a misspelled word is located.

One of the advantages of tree-based dictionary storage is that there is an element of spelling correction built into the structure of the dictionary tree. Suppose a dictionary tree is being searched for a word. For example, let the first four letters be valid but the fifth letter be invalid. Therefore, the fifth letter will not be in the filial set of the fourth. A pre-order search of the tree can be performed from the fourth letter's node. In this way, all legal words beginning with the first four letters of the word being checked can be reconstructed. Hopefully, the correct spelling of this incorrect word would be amongst those generated.

Clarification of the idea can be achieved by recalling the dictionary tree of Figure 5. Suppose the word 'INSERTING' has been misspelled 'INSERQING'. A path could be followed in the tree as far as 'INSER'. If a pre-order search from this 'R' node is performed, the words 'INSERT', 'INSERTED' and 'INSERTING' could be found. Hence, a list of possible correct spellings can be generated.

This method of spelling correction, although very fast, suffers from one major drawback. Suppose the word 'INSERTING' was misspelled as 'IMSERTING'. Trying to generate correct words would yield 'IMMACULATE', 'IMMENSE', and 'IMMIGRANT'. A totally incorrect part of the tree has been traversed. This inherent spelling correction works well when the spelling mistake is toward the end of the word, but can fail badly when the mistake is near the beginning of the word. A solution to this drawback is discussed next.

2.6 Reversing the Dictionary

Assume the word 'INSERTING' has been misspelled as 'INSTRTING'. In suggesting a list of correctly spelled words, the wrong path of the dictionary tree would be examined. For example, words like 'INSTRUCT' and 'INSTRUCTION' instead of the correct word 'INSERTING' would be retrieved. All of the letters up to and including the incorrect letter form the start of a legal word. With the misspelled word 'IZSERTING', no suggested list would be generated at all. Therefore, when the letter in error is near to the end of a word, it is more likely that the correct spelling will be suggested.

If storage space is less of a constraint two copies of the compact dictionary tree could be retained. In the application described, this would still take up less space that the original dictionary tree. The first dictionary would be stored as a dictionary tree. The second dictionary would have all the words of the 'flat' dictionary file reversed and re-sorted before compression. In this way, the words are stored in the dictionary tree backwards.

With words such as 'IZSERTING', the position of the incorrect letter is known exactly. So, if a spelling mistake occurs at the beginning of a word, the word can be reversed and the *backwards* dictionary examined to generate a list of suggested correct spellings. When a spelling mistake occurs toward the end of a word, the normal copy of the dictionary tree can be examined whereas if it is near the middle of the word, the best strategy would appear to be to generate two lists of suggested correct spellings. The first list comes from using the misspelled word and the *forwards* dictionary while the second from using the reverse word and the *backwards* dictionary. These two lists can then be merged. This process should generate the correct spelling in the majority of cases.

3 Implementation Details

The method chosen to represent a dictionary tree in main memory allowed each node to use two pointers. Each node points to the node which begins *its* filial set and to the next node of the filial set of which it is a member (if one exists). Figure 9 represents the transformed dictionary tree of Figure 5. Note that the first letter 'A' in 'IMMACULATE' points at 'C', the first and only member of its filial set and to 'E', the next member of the filial set it belongs to.

```
I*_ M _ M _ A _ C _ U _ L _ A _ T _ E*
    |       |
    |       E _ N _ S _ E*
    |       |
    |       I _ G _ R _ A _ N _ T*
    |
    N*_ S _ E _ R _ T*_ E _ D*
                        |
                        I _ N _ G*
```

Figure 9: A Dictionary Tree with Two Pointers

This can be represented in tabular form as shown in Figure 10. One of the most notable things is that the first member of the filial set of a node is always immediately after it in the table. Note that the first 'A' in 'IMMACULATE' is at index 4 and 'C' is at index 5. Therefore, if the filial set of a node is always assumed to follow immediately after the node itself, the pointer to the filial set field is redundant.

```
        -- Pointer to next member of the
        |  same filial set.
        |
        |   -- Node
        |   |
        |   |   -- Ptr to start of filial set
        |   |   |
        v   v   v
       -----------              -----------
    1 |     I*  2 |       17 |     R  18 |
    2 | 21  M   3 |       18 |     A  19 |
    3 |     M   4 |       19 |     N  20 |
    4 | 11  A   5 |       20 |     T*    |
    5 |     C   6 |       21 |     N* 22 |
    6 |     U   7 |       22 |     S  23 |
    7 |     L   8 |       23 |     E  24 |
    8 |     A   9 |       24 |     R  25 |
    9 |     T  10 |       25 |     T* 26 |
   10 |     E*    |       26 | 28  E  27 |
   11 | 15  E  12 |       27 |     D* 28 |
   12 |     N  13 |       28 |     I  29 |
   13 |     S  14 |       29 |     N  16 |
   14 |     E*    |       30 |     G* 17 |
   15 |     I  16 |             -----------
   16 |     G  17 |
       -----------

     * = end of word
```

Figure 10: Tabular Representation of a Dictionary Tree

Two bytes could be chosen to represent each table entry. Only five bits are needed to store the 26 letters of the English alphabet, thus three bits were available in each byte for control. The first of these three control bits was used as a flag indicating whether the current node is the end of a word (1) or within a word (0). Nodes which are both at the end of a word and within a word have this field set to 1.

3.1 Variable Length Pointers

After analysis of a 20,000 word dictionary, it was found that the average value held in the pointer byte was 19 with the largest value being 3237. Approximately, only one percent of pointer values are greater than 255. Using this information, the surplus two control bits in the byte holding a letter were used to indicate the number of bytes used to hold the pointer value. No pointer was coded as (00), a one byte pointer as (01) or a two byte in pointer as (10). Using this method of variable length pointers, there is provision to have a pointer value up to 64K. (11) was reserved for future use.

3.2 Common Word Endings

The common word ending table was considered next. The range of values in use were from '00000xxx' (representing 'A') to '11001xxx' (representing 'Z'). It was decided to use values in the range '11100000' to '11111111' to represent a pointer to the word ending table. This gives 32 common word endings, so the 10 most common four and three character word endings, and the 12 most common two character word endings were selected for inclusion in the table.

The following is a summary of the field meanings as used by the final dictionary tree storage method:

00000xxx The node represents the letter 'A'.

11001xxx The node represents the letter 'Z'.

010001xx The node represents the letter 'I' and this is the end of a word.

00001x01 The node represents the letter 'B'. There is a one byte pointer in the next memory location.

00001x10 The node represents the letter 'B'. There is a two byte pointer in the next two memory locations.

11100000 The node represents the first entry into the common word endings table.

11111111 The node represents the last (32nd) entry into the common word endings table.

Using these storage representation techniques, together with the use of common word endings, filial sets based on Zipfian distributions, and variable length pointers, etc the sample dictionary is shown in Figure 11.

```
I              Word endings table
IN             ------------------
INSERT                 .
INSERTED               .
INSERTING              .
IMMACULATE            13  ING
IMMENSE               14  ATE
IMMIGRANT             15  ANT
                       .
                       .


 1    I* 01000100        15  20   00010100
 2    N* 01101101        16   C   00010000
 3  12   00001100        17   U   10100000
 4    S  10010000        18   L   01011000
 5    E  00100000        19 218   11101110 ATE
 6    R  10001000        20   E   00100001
 7    T* 10011100        21  25   00011001
 8    E  00100000        22   N   01101000
 9  11   00001011        23   S   10010000
10    D* 00011100        24   E*  00100100
11 217   11101101 ING    25   I   01000000
12    M  01100000        26   G   00110000
13    M  01100000        27   R   10001000
14    A  00000001        28 219   11101111 ANT
```

Figure 11: Complete Representation of a Dictionary Tree

3.3 Searching the Dictionary Tree

How is this dictionary tree searched to find the presence or absence of a word? The method to search the tree is illustrated in Figure 12.

```
Suppose we are scanning the dictionary to see if the
word made up of characters A1..An is in the dictionary.

Go to the tree whose root matches A1

set a counter c to 1

repeat

  if current letter is An and the current node has the
  end of word flag set then
    word A1,A2,A3...An is correctly spelled
  else if current letter is a common word ending pointer then
    if Ac..An match the word ending then
      word A1,A2,A3...An is correctly spelled
    else
      add one to c to point to the next letter in A1..An

  if there is a pointer from the current node then
    add x to the current pointer where x is the
    size of the pointer (1 or 2 bytes). This takes
    us to the start of the filial set of the
    previous node.

  while current node isn't the same as Ac and there
  is a pointer from the current node
    follow down the pointer
  endwhile

  if the current node is not Ac then
    word is incorrectly spelled

until we know if the word is correctly spelled or not
```

Figure 12: Searching the Compacted Dictionary Tree

As an example of a correctly spelled word, the following steps would be undertaken in searching for the word 'IMMENSE' with reference to Figure 12.

1. The first letter 'I' will be compared with the root of the tree. The two letters are the same.
2. The second letter 'M' will be compared with the 'N' at location 2. They do not match but the 'N' contains a pointer to the next member of its filial set (i.e. at location 12).

3. The second letter 'M' will be compared with the 'M' at location 12. They match.

4. The third letter 'M' will be compared with the 'M' at location 13. They match.

5. The fourth letter 'E' will be compared with the 'A' at location 14. They do not match but the 'A' contains a pointer to the next members of it's filial (i.e. at location 20).

6. The fourth letter 'E' will be compared with the 'E' at location 20. They match.

7. The fifth letter 'N' will be compared with the 'N' at location 22 (not location 21 since this contains a pointer). They match.

8. The sixth letter 'S' will be compared with the 'S' at location 23. They match.

9. The seventh letter 'E' will be compared with the 'E' at location 24. They match.

10. The seventh letter 'E' is the last letter in the word and the last letter matched from the dictionary ('E' at location 24) has the end of word pointer set so the word 'IMMENSE' is in the dictionary.

3.4 Detection of Word Boundaries

There is a problem with the algorithm given in Figure 12. The algorithm as it stands does not detect the end of word boundaries and hence, for example, the word 'INSERTEDING' will not be flagged as being incorrect. The solution to the problem is however relatively easy to implement. During the scan through the dictionary, a record must be kept of the smallest pointer that is 'jumped over'. In the above example using the word 'IMMENSE', the pointer to location 25 which was at location 21 was jumped over. The smallest of these pointers that is jumped over in searching for a word is the end of word boundary, and it must be ensured that any of the dictionary beyond this boundary is not considered. For the word 'INSERTEDING' the smallest of the pointers jumped over is to location 11. If location 11 is reached when considering the word (which is the case for 'INSERTEDING') then the word is incorrect.

4 The Spelling Corrector

The following *C* programs have been written:

- *zipdict.c* — This program will take a 'flat' dictionary file and re-sort it according to the Zipfian distribution.

- *supcom.c* — This program will take a file sorted according to a Zipfian distribution and compact it according to the principles of the tree based storage method with common word endings, variable length pointers and Zipfian sorted filial sets.
- *examine.c* — The algorithm to search the compact dictionary tree and detect the end of word boundaries has been implemented as a set of *C* functions which are in the program *examine.c*.

To use the functions, an initial call must be made to *set_up_dict()*. This simply reads the dictionary tree into memory from the file specified by the string 'DICT_NAME'.

When the dictionary has been set up, the function *correctly_spelt()* can be used. This function is passed a string of up to 30 characters representing the word in upper case to be checked; e.g. *correctly_spelt("HELLO")*. The function returns TRUE if the word is correctly spelled, otherwise FALSE.

The spell correction algorithm also appears in the file *examine.c* as a function *find_correct_spelling()*. Three parameters are required: a string representing the incorrect word; a pointer into an array of strings to hold the suggested spellings; and the size of this array. For example the statement:

find_correct_spelling("HELLJ", sugg_spell, 30)

would fill the array *sugg_spell* with up to 30 suggested spellings.

4.1 The User Interface

The spelling corrector has a full screen presentation of the user's document. The document is displayed a page at a time under the control of the user. Each word that is unknown to the spelling corrector (i.e. those that are not in the dictionary), is displayed in *reverse video*. Therefore, the user can locate misspelled words quickly and decide upon the correct spelling based on the context of the complete sentence.

The page from the user's document is displayed in the middle 23 lines of the screen. The top line contains a copyright message. The bottom line contains a *function bar* giving a summary of the current keys that can have effect. The function bar changes according to the operation *mode* of the spelling corrector. The first mode is *word select* mode in which the cursor keys are used to move around the screen. The second mode is *word correct* mode in which the user can correct a word based on suggested correct spellings from the program. There are two function bars, one for word select mode and one for word correct mode. In word correct mode there are two ways in which a misspelling can be corrected. The first of these is where the user types in the correct spelling directly. The second is where the spelling corrector displays a list of suggested correct spellings from which the user can select the appropriate word.

4.2 Addenda Dictionaries

In a single user PC type environment, each user would have his or her own copy of the compact dictionary tree. Each user would be responsible for the maintenance of the dictionary. However, in a multi-user UNIX environment it makes no sense for each user to have a separate copy of the dictionary. Multiple copies of the dictionary would waste storage space.

Using a common dictionary implies its maintenance must be the responsibility of one person. Therefore, the interactive spelling corrector does not contain a facility for the user to add words to the main dictionary. If anyone could add words to the dictionary, there would be a danger that it could eventually contain misspelled words. Instead a user can invoke the spelling corrector with an optional *addenda* file. This will contain correctly spelled words not in the dictionary, for example names or acronyms. The spelling corrector will then check this addenda dictionary as well as the main dictionary. At any point the user may add a misspelled word to the current addenda dictionary by moving the cursor to the word and pressing the 'A' key. Any future occurrence of this word in the document will not then be flagged as incorrect.

5 Conclusions

Techniques for file storage and record retrieval based on trees are definitely useful for certain types of file. The English language dictionary is a prime candidate for compression in this manner. Indeed any file that has many records with repeated keys would be suitable for storage in a tree. Improvements to the tree-based storage method such as the grouping together of all common subtrees and ordering of the keys within the filial sets were considered. Both of these additions gave significant improvements over the original tree structure.

As a practical example, an efficient implementation to compress an English language dictionary has been coded. A 36,000 word English dictionary file was compressed from 320K to a dictionary tree of 117K; this compact file being just over one third the size of the original file. It has been shown that this storage method was more suitable for dictionaries than other methods. The dictionary tree is extremely quick to access; on an IBM PC, the presence or absence of a given word can be checked in 0.006 seconds (this is an average access time per word). Also, the generation of correct spellings was easy since it was discovered that the nature of a dictionary tree has an element of 'built-in' spelling correction.

This compact dictionary was used as the basis of a fully interactive, user-friendly, spelling corrector for UNIX and PC based systems. It is written in *C* and the UNIX version has been *termcapped.*

The primary goals of the spelling corrector were efficiency of dictionary storage, efficiency of dictionary access and ease of use. The first two were achieved by extending the tree structured techniques mentioned above. The latter was achieved by careful consideration of the human-computer interface, the screen display and user input.

6 References

- Iverson, K
  A Programming Language Wiley, New York, 1962

- Knuth, D
  The Art of Computer Programming Volume 3: Sorting and Searching, Addison Wesley, Reading, Massachusetts, 1973

- Shneiderman, B
  Designing the User Interface Addison Wesley, Massachusetts, 1987

- Stanfel, L
  'Tree Structures for Optimal Searching' *Journal of the ACM, Vol 17, No 3* July 1970

- Sussenguth, E
  'Use of Tree Structures for Processing Files' *Communications of the ACM, Vol 6, No 5* May 1963

Interprocess Communication
in the Ninth Edition Unix System

D. L. Presotto
D. M. Ritchie

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

When processes wish to communicate, they must first establish communication, and then decide what to say. The stream mechanisms introduced in the Eighth Edition Unix system,[1] which have now become part of AT&T's Unix System V,[2] provide a flexible way for processes to speak with devices and with each other: an existing stream connection is named by a file descriptor, and the usual read, write, and I/O control requests apply. Processing modules may be inserted dynamically into a stream connection, so network protocols, terminal processing, and device drivers separate cleanly.

Simple extensions provide new ways of establishing communication. In our system, the traditional Unix IPC mechanism, the pipe, is a cross-connected stream.

A new request associates a stream with a named file. When the file is opened, operations on the file are operations on the stream.

Open files may be passed from one process to another over a stream.

These low-level mechanisms allow construction of flexible and general routines for connecting local and remote processes.

Introduction

The Ninth Edition version of Unix® operating system is used in the Information Sciences Research Division of AT&T Bell Laboratories, and at a few sites elsewhere. It is named, by our custom, after its manual.

The work reported here provides convenient ways for programs to establish communication with unrelated processes, on the same or different machines. The communication we are interested in is conducted by ordinary read and write calls, occasionally supplemented by I/O control requests, so that it resembles—and, where possible, is indistinguishable from—I/O to files. Moreover, we wish to commence communication in ways that resemble the opening of ordinary files, or at least takes advantage of the properties of the file system name space.

In particular, we study how to

1) provide objects nameable as files that invoke useful services, such as connecting to other machines over various media,
2) make it easy to write the programs that provide the services.

Unix is a trademark of AT&T.

Recapitulation

The Eighth Edition system introduced a new way of communicating with terminal and network devices,[1] and a generalization of the internal interface to the file system.[3,4] Because the new mechanisms build on these ideas, we review already-published nomenclature and mechanisms of our I/O and file systems.

Streams

A *stream* is a full-duplex connection between a process and a device or another process. It consists of several linearly connected processing modules, and is analogous to a Shell pipeline, except that data flows in both directions. The modules in a stream communicate by passing messages to their neighbors. A module provides only one entry point to each neighbor, namely a routine that accepts messages.

At the end of the stream closest to the process is a set of routines that provide the interface to the rest of the system. A user's *write* and I/O control requests are turned into messages sent along the stream, and *read* requests take data from the stream and pass it to the user. At the other end of the stream is either a device driver module, or another process. Data arriving from the stream at a driver module is transmitted to the device, and data and state transitions detected by the device are composed into messages and sent into the stream towards the user process. Pipes, which are streams connecting processes, are bidirectional; a writer at either end generates stream messages that are picked up by the reader at the other.

Intermediate modules process messages in various ways. They come in pairs, for handling messages in each of the two directions, and each pair is symmetrical; their read and write interfaces are identical.

The end modules in a device stream become connected automatically when the process opens the device; streams between processes are created by a *pipe* call. Intermediate modules are attached dynamically by request of the user's program. They are addressed like a stack with its top close to the process, so installing one is called 'pushing' a new module.

For example, Figure 1 shows a stream device that has just been opened. The top-level routines, drawn as a pair of half-open rectangles on the left, are invoked by users' *read* and *write* calls. The writer routine sends messages to the device driver shown on the right. Data arriving from the device becomes messages sent to the top-level reader routine, which returns the data to the user process when it executes *read*.

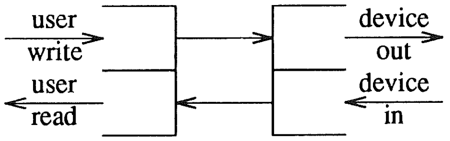


Figure 1. Configuration after device open.

Figure 2 shows a stream with intermediate modules. This arrangement might be used when a terminal is connected to the computer through a network. The leftmost intermediate module carries out processing (such as character-erase and line-kill) needed for terminals, while the rightmost intermediate module does the flow- and error-control protocol needed to interface to the network.

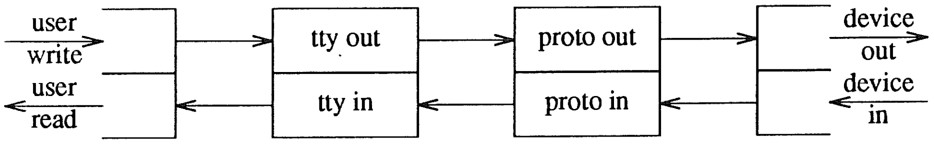


Figure 2. Configuration for network terminals.

Finally, Figure 3 shows the connections for a pipe.

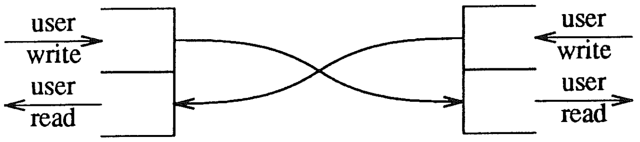


Figure 3. A pipe.

File Systems

Weinberger[3] generalized the file system by identifying a small set of primitive operations on files (read, write, look up name, truncate, get status, etc.: a total of 11) and modifying the *mount* request so that it specifies a file system type and, where appropriate, a stream. When file operations are requested, the calls to the underlying primitives are routed through a switch table indexed by the type. Where the standard file system type performs operations directly on a disk, a second type generates remote procedure calls across the associated stream. At the other end of the stream, which usually goes over a network to another machine, is a server process that answers the calls to read and write data and perform the other operations. This scheme thus provides a remote file system. In general structure, this arrangement is analogous to that used by AT&T's RFS and Sun Microsystems' NFS.

Pike[5] took advantage of the remote file system type, but his server simulates a disk containing images classified by machine, person's name, and resolution.

Killian[4] added a file system type that appears to be a directory containing the names (process ID numbers) of currently running processes. Once a process file is opened, its memory may be read or written, and control operations can start it or stop it. This simplifies the construction of sophisticated debuggers, for example Cargill's process-inspector *pi*.[6]

Establishing Communication

Traditional Unix systems provide few ways for a process to establish communication with another. The oldest one, the pipe, has proved astonishingly valuable despite its limitations, and indeed remains central in the design we shall describe. Its cardinal limitation is, of course, that it is anonymous, and cannot be used to create a channel between unrelated processes.

More recently, AT&T's System V has offered a variety of communication mechanisms including semaphores, messages, and shared memory. They are all useful in certain circumstances, but programs that use them are all special-purpose; they know that they are communicating over a certain kind of channel, and must use special calls and techniques. System V also provides named pipes (FIFOs). They reside in the file system, and ordinary I/O operations apply to them. They can provide a convenient place for processes to meet. However, because the messages of all writers are intermingled, writers must observe a carefully designed, application-specific protocol when using them. Moreover, FIFOs supply only one-way communication; to receive a reply from a process reached through a FIFO, a return channel must be constructed somehow.

Berkeley's 4.2 BSD system introduced *sockets* (communication connection points) that exist in domains (naming spaces). The design is powerful enough to provide most of the needed facilities, but is uncomfortable in various ways. For example, unless extensive libraries are used, creating a new domain implies additions to the kernel. Consider the problem of adding a 'phone' domain, in which the addresses are telephone numbers. Should complicated negotiations with various kinds of automatic dialers be added to the kernel? If not, how can the required code be invoked in user mode when a program calls 4.2 BSD's *connect* primitive?

Another problem with the socket interface is that it exposes peculiarities of the domain; various domains support very different kinds of name (for example, Internet address versus character string), and it is difficult to deal with the names in a general way. Similarly, the details of option processing and other aspects of each domain's protocols complicate an interface that attempts generality.

New System Mechanisms

Two small additions to the operating system allowed us to build a variety of communication mechanisms, which will be described below.

Generalized Mounting

Traditionally, the *mount* request attaches a disk containing a new piece of the file system tree at a leaf of the existing structure. In the Ninth Edition, it takes the form

```
mount(type, fd, name, flag);
```

in which *type* identifies the kind of file system, *fd* is a file descriptor, *name* is a string identifying a file, and *flag* may specify a few options. Like its original version, this call attaches a new file system structure atop the file *name* in the existing file hierarchy. The operating system gains access to the contents of newly-attached file tree by communicating over the descriptor *fd,* according to a protocol appropriate for the new file system *type*. For example, ordinary disk volumes have type *ordinary,* and the file descriptor is the special file for the disk, while remote file systems use type *remote,* and the descriptor refers to a stream connection to a server that understands the appropriate RPC messages. Some types are handled entirely internally; for example, the 'proc' type does not need the file descriptor.

Recently, we added a new, very simple, file system type. Its *mount* request merely attaches the file descriptor (which must be a stream) to the file. Subsequently, when processes open and do I/O on that file, their requests refer to the stream mounted on the file. Often, the stream is one end of a pipe created by a server process, but it can equally well be a connection to a device, or a network connection to a process on another machine. The effect is similar to a System V FIFO that has already been opened by a server, but more general: communication is full-duplex, the server can be on another machine, and (because the connection is a stream), intermediate processing modules may be installed.

Passing Files

By itself, a mounted stream shares an important difficulty of the FIFO; several processes attempting to use it simultaneously must somehow cooperate. Another addition facilitates this cooperation: an open file may be passed from one process to another across a pipe connection. The primitives may be written

```
sendfile(wpipefd, fd);
```

in the sender process, and

```
(fd1, info) = recvfile(rpipefd);
```

in the receiver. By using *sendfile,* the sender transmits a copy of its file descriptor *fd* over the pipe to the receiver; when the receiver accepts it by *recvfile,* it gains a new open file denoted by *fd1.* (Other information, such as the user- and group-id of the sender, is also passed.) The facility may be used only locally, over a pipe; we do not attempt to extend it to remote systems.

A similar facility is available in the 4.3 BSD system,[7] but is little-used, possibly because in earlier versions the related socket facilities were buggy.

Simple Examples

A graded set of examples will illustrate how these mechanisms can solve problems that vex other systems.

Talking to Users

When a user logs in to traditional Unix systems, an entry is made in the */etc/wtmp* file, recording the login name and the terminal or network channel being used. Although this file is often used merely to show who is where, it is also used to establish communication with the user. For example, the *write* command, and a variety of mail-notification services, find a user's terminal by looking up the name, and send a message to the terminal. This simple scheme does not work well with windowing terminals, because the messages disturb the protocol between the host and the terminal, and because there is no obvious way to relate the terminal's special file to a particular window. Even without windows, there are security problems and other difficulties that follow from letting users write on each other's terminals.

Instead, we use stream-mounting to interpose a program between a terminal special file and the terminal itself. The program, called *vismon*, mounts one end of a pipe on the user's terminal. Normally it occupies an inconspicuous window, displaying system activity and announcing arriving mail. When some other process opens and writes on the special file for the terminal, the mounted stream receives the data; *vismon* creates a new window, and copies this data to it. The new window has a shell, so that if the message was from a *write* command, the recipient can write back.

Ordinary communication between the terminal and the windowing multiplexor on the host is not disturbed; it continues to flow to the terminal itself, not to *vismon,* because that connection was already in place at the time the *mount* was done.

Network Calling: Simple Form

Making a network connection is a complicated activity. There is often name translation of various kinds, and sometimes negotiations with various entities. With the Datakit® VCS network,[8] for example, a call is placed by negotiating with a node controller. When dialing over the switched telephone system, one must talk to any of several kinds of automatic dialers. Setting up a connection on an Internet under any of the extant protocols requires translation of a symbolic name to a net address, and then special communication with the remote host. These protocols should certainly not be in kernel code. It is usual to put setup negotiations in user-callable libraries, but it is better to have all the code for each network in a single executable file. In this way, if something in the network interface changes, only one program needs to be fixed and reinstalled.

With our primitives, it is straightforward to move most parts of network-connection algorithms to a single program. A program desiring to make a connection calls a simple routine that creates a pipe, forks, and in the child process executes the network dialer program. The dialer either returns an error code, or passes back a file descriptor referring to an open connection to the other machine. The pseudo-code for the library routine, neglecting error-checking and closing down the pipe, is:

```
netcall(address)
{   int p[2];

    pipe(p);
    if (fork()!=0)
        execute("/etc/netcaller", address, p[0]);
    status = wait();
    if (bad(status))
        return(errcode);
    passedinfo = recvfile(p[1]);
    return(passedinfo.fd);
}
```

The /etc/netcaller program can be arbitrarily complicated, but does not occupy the same address space as its caller. Its job is to create the connection and either fail, returning an appropriate error code, or succeed, and pass its descriptor for the open connection; it then terminates, and is no longer involved in the connection. Along the way, it may negotiate permissions and provide the caller's identity reliably, because it can be a privileged (set-uid) program. Thus, the segregation of the program that does the actual call setup from its client is important. There are techniques, for example shared libraries, that can reduce the code included with each program that makes network connections, but such libraries run in the protection domain of the person who executes them. This means that in library-based implementations, the operating system must know enough about the call setup protocols to authenticate the caller to the target system. In the method described above, this task need not be done in kernel code.

Process Connections

Suppose you are writing a multi-player game, in which several people interact with each other. One solution uses a controller process that is prepared to receive asynchronous connection requests from new players and coordinates the play of the game. In this scheme, there are two programs: the controller, set up initially, and a player program, executed by users as they enter the game. When the controller starts, it creates a conventionally-located file, stream-mounts one end of a pipe on this file, and waits for connection messages to arrive.

When the player program is run, it opens the communication file, and creates its own pipe. It starts communication by sending one end of this pipe to the game controller over the communication file.

When the controller notices that there is input on its connection stream, it accepts the connection with *recvfile*. The player program can then close the communication file, and thereafter transmits moves and receives replies over its end of the pipe; the controller reads the player's moves and transmits replies over the end it received.

The Connection Server

The final example illustrates a general connection server that we have recently installed. It combines ideas used by the initial network-calling scheme and the game-master design, described above, to create a flexible switchboard through which programs can connect.

Two things are necessary for handling server-client relationships: first, some program must establish itself as a server, and wait for requests for the service; and second, programs must make requests. We will first describe the external appearance of the scheme (the library entry points), then the addressing and naming, and then the implementation.

A program like *rlogin* or *cu* makes a connection by calling the routine *ipcopen,* passing a string of characters that specifies the address and the desired service at that address.

```
fd = ipcopen(service);
```

The *ipcopen* routine returns a file descriptor connected to the requested server. If it fails, a string describing the error is available.

In order to announce a service, *ipccreat* is used; its argument is a string that names the service. The return value is a file descriptor *fd* that is a channel on which connection requests will be sent.

```
fd = ipccreat(service);
```

To wait for requests, the server uses the *ipclisten* routine. Its argument is the same *fd* returned by *ipccreat:*

```
ip = ipclisten(fd);
```

Ipclisten returns when some program calls *ipcopen* with an argument corresponding to the service, in a way discussed below. The return value is a structure containing information about the caller, such as the user name, and, where relevant, the name of the machine from which the call was placed. This new connection may be accepted:

```
fd = ipcaccept(ip, cfd);
```

or it may be rejected:

```
ipcreject(ip, errcode);
```

The *ipcaccept* routine returns a file descriptor over which the server may communicate with its client. If the call is purely local, the *fd* returned by *ipcaccept* refers to one end of a pipe whose other end is the *fd* returned by the corresponding call to *ipcopen.* If the client and server are on separate machines, then the connection refers to a stream that passes over a network. *Ipcaccept* also takes a file descriptor as argument. It is used when the service being provided is to make another connection. A network dialing server, for example, receives the desired address in the *ip* structure returned by *ipclisten,* makes the connection with network-specific primitives, and if the connection succeeds, returns the descriptor for the connection to the client using the *cfd* argument of *ipcaccept.* In the simpler case in which the server itself communicates directly with the client, *cfd* may be empty.

Addresses

The arguments supplied to *ipcopen* and *ipccreat* are strings with several components separated by exclamation mark '!' characters. The first part is interpreted as a file name. If it is absolute, it is used as is; otherwise, it is interpreted as a file in the directory */cs,* which we use, conventionally, as a rendezvous point. In the case of *ipcopen,* any remaining components are passed to the server as part of the *ip* structure returned by *ipclisten.* For example, a game controller like that discussed in a previous section might announce itself with

```
ipccreat("mazewar");
```

The player program could then connect to the controller with

```
ipcopen("mazewar");
```

In this simple case, the IPC routines merely accomplish a convenient packaging of the scheme discussed above.

More interesting are the network servers. Here, the first component of the string names a kind of network, and, conventionally, the remaining components supply an address within that network, and possibly a service obtainable at that address. For example, we have three kinds of networks: *tcp* (TCP/IP Internet connection), *dk* (Datakit connection), and *phone* (dial-up telephone). Each network server adopts the convention that a missing service name means a connection to an end-point that allows one to log in by hand. Therefore, calling *ipcopen* with the strings

```
tcp!research.att.com
dk!mh/astro/research
phone!201-582-5940
```

gets connections over which one will receive a 'login:' greeting, each over a different kind of network. The servers are responsible for the details of name translation, performing the appropriate connection protocol, and so forth. Some examples of named services at particular locations are

```
dk!dutoit!whoami
tcp!research.att.com!smtp
```

The first is a debugging service that echoes facts about the connection and the user ID of the person who requests it. The second is the way mail is sent between machines: by connected to the *smtp* server (mail receiver) on the machine.

IPC Implementation

The *ipccreat* routine, for a simple service, works just like the game-manager program described above; it first creates a file in the */cs* directory corresponding to the name of the service, then makes a pipe and stream-mounts one end of the pipe on this file. For complex services, which have a '!' in their names, the simple service named to the left of the '!' must be created first; when *ipccreat* is handed the name of such a service, it uses a version of *ipcopen* referring to the simple, underlying server, and passes it the remainder of the name. In either case, *ipccreat* returns its own end of its pipe, ready to receive requests.

The *ipcopen* routine uses a technique that resembles that used by the simple network calling routine described above, but differs in detail. The following scheme suffices: *ipcopen* opens the file in */cs* corresponding to the desired service, makes a pipe, and hands one end of the pipe to the server. It then sends the actual contents of the request (the full address) to its end of the pipe, and waits for an acceptance or rejection message to appear on this pipe.

The server *ipclisten* call waits for a passed stream on the file descriptor mounted on its */cs* communication file, and when it appears, *ipclisten* knows that someone has called *ipcopen,* so it reads the request block from this passed stream. After analyzing the request, the server calls either *ipcaccept* or *ipcreject;* each sends an appropriate message back to the client over the passed stream. *Ipcaccept* has two cases:

when its *cfd* argument is empty, the same pipe sent to the server by the client is used for communication; when *cfd* is non-empty, that file descriptor is sent. *Ipcopen* returns the appropriate descriptor.

Network Managers

The IPC routines discussed above handle both clients and servers that are local to a single system, and we also showed how to accomplish outgoing network connections. One missing piece is how to write network servers, the programs that accept connections from a network, and arrange that the appropriate local programs are invoked. We call such programs *managers.* The networking part of a manager is specific to the network. For example, the manager for a TCP/IP network must arrange to receive IP packets sent to certain port numbers, and analyze the packets to determine what service is being requested. Often, it must conduct a dialogue both with the operating system and with its remote client; for a TCP connection, it must select a port number for the conversation, communicate it to the peer, and prepare the system to route packets on this port number to it. Finally, the manager must invoke the selected local service. Each manager could use *ad hoc* code for this part of its job; instead, we take advantage of the IPC mechanisms, and use a more general program called the *service manager.*

The Service Manager

By using *ipccreat,* a process establishes itself as a server and receives requests. While it is serving, it must remain in existence. For some servers, like the multi-player game controller that continues to run as users enter and leave the game, the longevity of the server is appropriate. However, many, or even most, useful services do not necessarily need a long-lived process, because the service merely involves execution of a particular program. For example, services like *rlogin, telnet, smtp* and *ftp,* as well as simpler ones that merely provide the date, or send a file to a line printer, can all be accomplished merely by running the appropriate program with input and output connected to the right place. Moreover, even when the characteristics of such services differ in detail, there are general patterns. Some for example, require no authentication, some require checking of authentication according to an automatic scheme, and others always insist on a password.

The observation that many services share a common structure suggested a common solution: the Service Manager. It is started when the operating system is booted, and is driven by a specification file; each entry in the file contains the name of the service, and a list of actions to be performed when that service is requested. The service manager issues *ipccreat* for the name given in each entry;. when another process uses *ipcopen* to request the service, the service manager carries out the specified actions.

The most important action specifies the command to be executed; for example, the line

```
date      cmd(date)
```

means that connecting to the service *date* would run the 'date' command. Other actions may specify the user ID under which the program is run:

```
uucp      user(uucp)+cmd(/usr/lib/uucp/uucico)
```

This service specifies a passwordless connection to the *uucp* file-transfer program; a locally-conventional TCP/IP port number is used for such connections, and a corresponding convention is used on our Datakit network. There are other built-in actions:

```
login          ttyld+password
```

means that the *login* service needs to install the line discipline module for terminal processing, and also to execute the *login* command without the special flag that causes it to avoid demanding a password;

```
oklogin        auth+ttyld+login
```

is similar, but allows passwordless login. Authorization is checked by the *auth* specification, which determines whether the call came from a trusted host on a trusted network, so that the passed user ID can be believed.

Uses

The techniques described in this paper permit a general approach to network and local connections in which most of the work is done in a few user-mode programs. As an example of the benefits of the scheme, we have unified various commands that do remote login over two kinds of networks (TCP/IP and Datakit). A single command, *con,* tries various networks and uses the first over which a connection can be made. The traditional names (like *rlogin*) are retained as links, but the only effect of using them is to influence the order in which networks are tried. The stream implementation makes the transport layers of the networks sufficiently similar that the same code can be used once the connection is established; and even the connection interface itself becomes uniform.

The same techniques extend well to inter-network connectivity. For example, although almost all of our machines have a Datakit interface, only a few have Ethernet connections. Nevertheless, from a Datakit-only machine, it is easy to connect to another machines that has only Ethernet (whether it runs the Ninth Edition system or not). One of two methods is used. For the first, which depends on the flexibility of the stream I/O system, the local operating system contains the TCP/IP protocol code, and below the TCP/IP level, the 'device' interface is actually a Datakit connection to another local machine on an Ethernet. The usual IP network code there gateways IP packets appropriately. The other scheme uses the methods described in this paper. Here, the TCP network manager and dialout programs do not use TCP/IP at all, and indeed TCP/IP code need not be configured into the operating system; instead, they make Datakit transport-level connections to a protocol-conversion server on a gateway machine. The difference between the two schemes is invisible to users of the service.

Conclusion

Unix has always had a rich file system structure, both in its naming scheme (hierarchical directories) and in the properties of open files (disk files, devices, pipes). The Eighth Edition exploits the file system even more insistently than its predecessors or contemporaries of the same genus. Remote file systems, process files, and the face server all create objects with names that can be handed as usefully to an existing tool as to a new one designed to take advantage of the object's special properties. Similarly, the stream I/O system provides a framework for making file descriptors act in the standard way most programs already expect, while providing a richer underlying behavior, for handling network protocols, or processing appropriate for terminals.

The developments described here follow the same path; they encourage use of the file name space to establish communication between processes. In the best of cases, merely opening a named file is enough. More complicated situations require more involved negotiations, but the file system still supplies the point of contact. Moreover, the necessary negotiations may be encapsulated in a common form that hides the differences between local and any of a variety of remote connections.

References

References

1. D. M. Ritchie, "A Stream Input-Output System," *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, October 1984.
2. AT&T, *UNIX System V Release 3 STREAMS Programmer's Guide*, 307-227, 1986.
3. P. J. Weinberger, "The Version 8 Network File System," *USENIX Summer Conference Proceedings*, Salt Lake City, UT, June 1984.
4. T. J. Killian, "Processes as Files," *USENIX Summer Conference Proceedings*, Salt Lake City, UT, June 1984.
5. R. Pike and D. L. Presotto, "Face the Nation," *USENIX Summer Conference Proceedings*, Portland, OR, June 1985.
6. T. A. Cargill, "The Blit Debugger," *Journal of Systems and Software*, vol. 3, no. 4, pp. 277-284, December 1983.

7. Computer Systems Research Group, U.C. Berkeley, *Unix Programmer's Reference Manual: 4.3 Berkeley Software Distribution*, Berkeley, California, April, 1986.

8. A. G. Fraser, ''Datakit–A Modular Network for Synchronous and Asynchronous Traffic,'' *Proc. Int. Conf. on Commun.*, Boston, MA, June 1980.

Security in Standard Environments

Robert A. Michael
The Santa Cruz Operation, Inc.

Preliminary Paper

Introduction

As more and more business, government, and military information is handled by machine, security is taking a place alongside cost-effectiveness as a major concern when investing in a computer system.

Computer system security has traditionally been associated with expensive, proprietary hardware, and with software that is difficult to use and administer. But as microcomputers have grown in power and flexibility, another option has come to the fore. A standard hardware platform such as the 80386-based microcomputer solves many of the traditional problems of cost and obsolescence. The UNIX System V operating system is built to provide many security features without sacrificing ease of use. The combination forms a computing system that is at once cost-effective, flexible, and secure.

In order to control costs, it is important to be able to choose a platform with computing power appropriate to the application. An ability to selectively upgrade components of the system whenever necessary is also crucial, so that new operational requirements can be met without discarding those parts of the system which are still useful, The key to achieving these goals is the recognition and support of industry standards.

Many new standards are emerging today to address these needs at a number of levels, with the most important feature to consider being binary portability. Systems with binary portability allow the applications written for those systems to be transported between them without changes of any kind. Either proprietary hardware design or proprietary operating systems restrict or eliminate binary portability, especially between different systems vendors. Until recently trusted systems have relied on such proprietary designs, which has meant that not only the system itself but the applications have had to be modified or replaced when upgrading to new products. Also, the various different products which have been offered to date had little or no ability to share applications between different operational units, effectively locking a trusted systems user into a single vendor for a given application.

SCO UNIX System V/386 Release 3.2 will maintain full binary compatibility with the system that has become the de facto standard for the Intel 80386 hardware environment, SCO XENIX. All the popular enhancements to UNIX which have made SCO the number one supplier in the microprocessor UNIX software market will also be present.

Trusted Systems Background

A "Trusted" system is one which achieves a specific level of control over access to information, providing mechanisms to prevent (or at least detect) unauthorized access. In the United States, the standard criteria for evaluating the level of trust granted a computer system are listed in the Trusted Computing Systems Evaluation Criteria (TCSEC) issued by the National Computer Security Center (NCSC).

The TCSEC (also known as the "Orange Book" because of the color of its binding) does not contain specific "how to" rules for building trusted systems. What it does provide are guidelines for asking intelligent questions about the nature of the system, the answers to which may reveal potential security risks.

When all the criteria have been applied, the system receives a security rating. A rating of D indicates the lowest level of trust, with security levels increasing through C1, C2, B1, B2, B3, to reach A1 at the top of the scale. Claims of a specific level of trust are validated by an independent agency qualified to perform a complete evaluation. In the United States that agency is the NCSC.

In general, the UNIX operating system meets the requirements of a C1 system. It has password protection on accounts, a discretionary access control mechanism, and some limited authentication features. With proper system administration it can be used to maintain sensitive information with some degree of trust. Most sensitive environments, however, need at least the features of a C2 system. A C2-trusted system has strengthened password protection and authentication, protected subsystems (such as printers and tape drives), and makes administration tools available. While there is nothing about the basic design of UNIX that interferes with the requirements for C2, it is a substantial amount of work to implement them all.

Several companies have announced C2 functionality as a future option for their UNIX product offerings. SCO's entry in the C2-trusted field is SCO UNIX System V/386 Release 3.2, which will be available this year.

The areas in which SCO UNIX has been enhanced to satisfy the requirements of C2 include protected subsystems, administrator interfaces, auditing of system events, and removal of the encrypted passwords into a protected database. Many standard commands and utilities have been enhanced to operate correctly in the trusted environment as well. THis has been achieved in a manner which preserves application compatibility with the operating system, enabling users of the trusted environment to select from the thousands of currently-available commercial and business applications.

An optional extension to standard SCO UNIX System V/382 Release 3.2 is under development to satisfy the requirements of a B-level trusted system. Generalised access control mechanisms, mandatory access control, establishment of security category and level concepts, increased use of protected subsystems, removal of root as the all-powerful superuser, and numerous extensions and enhancements to standard utilities are all required to achieve this functionality. The Kernel mechanisms and implications for complete system evaluation provide interesting topics for further discussion.

Enhanced Error Processing for UNIX Parsers

A.N. Pears and R.S. Francis,
Concurrency Research Group
Department of Computer Science
La Trobe University, Bundoora, Australia 3083

Abstract

This paper describes the use of an error recovery filter designed for use with standard lex, yacc based parsers. Placement of actions within the grammar provides follow set error recovery similar to that used in recursive descent parsers. This paper presents an overview of the filter, and demonstrates how a grammar should be modified to provide effective error recovery. The definition of halting and follow sets is made within the yacc input grammar file. This definition includes an error message appropriate to each set, an optional insertion or deletion operation, and an error message for each symbol in a set. The result is a robust controlled error recovery of the parse to a precision determined by the implementor of the grammar.

Index terms Error Recovery, Parsing, Follow Symbols, Lex, Yacc.

Contact

Dr. R.S. Francis,
Concurrency Research Group,
Dept. of Computer Science,
LaTrobe University,
Bundoora, Vic 3083, Australia.

ACSnet/CSNET: rhys@latcs1.oz
ARPA: rhys%latcs1.oz@uunet.uu.net
JANET: latcs1.oz!rhys@ukc
UUCP: {enea,hplabs,mcvax,nttlab,ukc,uunet}!
munnari!latcs1.oz!rhys

ISD +613 **desk** 479 2504 **dept** 479 2598 **fax** 478 5814 **TELEX** AA 33143

Mr. A.N. Pears,
Concurrency Research Group,
Dept. of Computer Science,
LaTrobe University,
Bundoora, Vic 3083, Australia.

ACSnet/CSNET: pears@latcs1.oz
ARPA: pears%latcs1.oz@uunet.uu.net
JANET: latcs1.oz!pears@ukc
UUCP: {enea,hplabs,mcvax,nttlab,ukc,uunet}!
munnari!latcs1.oz!pears

ISD +613 **desk** 479 1144 **dept** 479 2598 **fax** 478 5814 **TELEX** AA 33143

1 Introduction

The UNIX[1] parser generators lex[5] and yacc[4] have been in regular use for many years. They simplify the implementation of programming languages by eliminating much of the labour involved in parser construction. Yacc allows an experienced programmer to rapidly construct a parser from syntax diagrams. Lex aids in the recognition of primitive symbols, which can then be supplied to yacc for parsing.

This paper reports on the design and use of an error recovery facility which, in conjunction with lex and yacc, provides robust and controllable error recovery. In common with others[2, 3], our work has found the error recovery support available in yacc to be deficient. It is difficult to apply, and in response to a single error is likely to produce many spurious error reports, or skip large parts of the input. Such action detracts from the usefulness of a parser in command and programming systems. In particular, inaccurate information arising from inadequate error recovery leads to significant user frustration.

The technique described in this paper has been successfully applied to the yacc grammar of a Modula II based systems programming language used for user and kernel programming in the **Threads** simulation system[1]. Our primary motivation in developing the strategy, was to increase the level of user information available during the normal syntax debugging of programs. A design constraint was the use of standard lex and yacc. The result is a filter which can be easily installed between lex and yacc, and which allows the grammar designer to include error recovery controi within the yacc grammar specification.

The technique described in this paper does not support advanced error repair strategies but rather concentrates on providing a robust parser. As in all LR parsers, good error recovery involves the resynchronisation of the input stream with the parse state. The simpler strategies attempt to delete symbols from the input and states from the parse stack until valid processing of the input file can be resumed. The objective in this case is to minimize both the number of symbols and parse states deleted.

Some previous attempts to provide full error recovery in yacc based parsers[6, 2] have concentrated on modifying the implementation of the yacc machine. These approaches, while partially successful, are complex and require considerable knowledge of the implementation of the yacc state machine. Others have attempted the formulation of a strategy relying on structured use of the existing error facilities. Such approaches require the partial restructuring of the grammar, and the introduction of complex agglomerations of *error* clauses.

Our approach departs from those already stated by providing extra facilities for error recovery to the standard yacc machine, using an active filter. No modifications are made to the yacc implementation, and the use of our facilities reduces the complexity of *error* clauses. The filter provides yacc with a library of actions, which, when inserted into the grammar, simulate follow symbol error recovery similar to that used in recursive descent parsers. The filter uses follow sets provided by these actions to force the parser stack to a state appropriate to the first follow symbol

[1] UNIX is a registered trademark of AT&T Bell Laboratories

located in the input. This ensures that the parser discards the minimum input necessary to reach a state known to be recoverable. The precision of the recovery is controlled by the number of error control actions included in the grammar. Fine levels of recovery may require significant grammar modifications. This paper will concentrate on techniques for the effective use of the facilities provided by the filter, coupled with an overview of the implementation.

2 An Overview of Errec

The positioning of the filter between yacc and lex is diagrammatically represented in Figure 1.

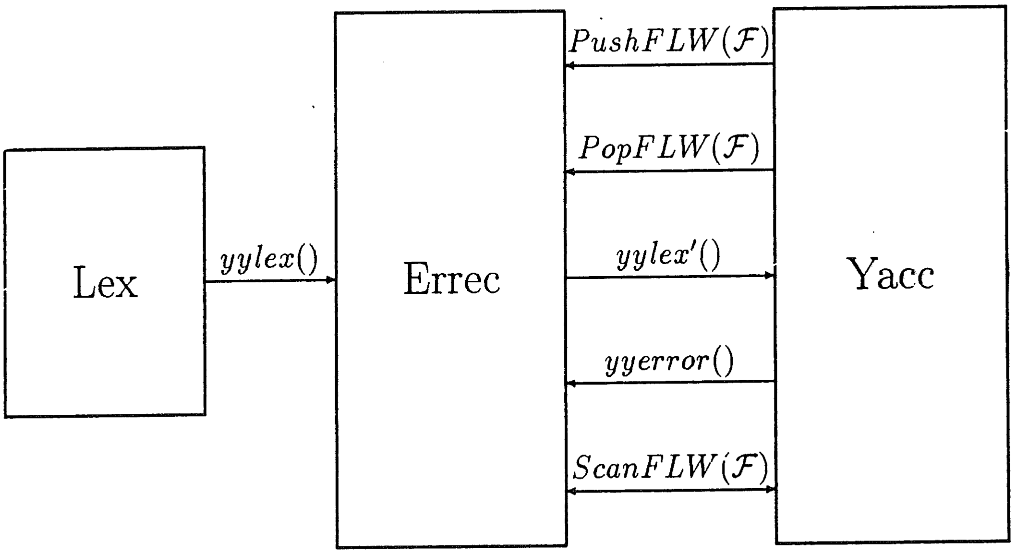


Figure 1: Errec Functional Interface

Yacc obtains its input using the function $yylex'()$, which **errec** maps to calls to $yylex()$ in accordance with it's operating state. **Errec** also traps calls to $yyerror()$ in the code generated by yacc in order to force a state change for error processing. In addition **errec** provides three new error recovery support functions for use in yacc actions.

1. PushFLW($\mathcal{F}$) extends the follow symbol stack to include the list of valid follow symbols associated with the set identifier $\mathcal{F}$. The association of set identifier and follow symbols is made in the yacc grammar file.

2. PopFLW($\mathcal{F}$) removes the top set of follow symbols from the stack. The identifier of the set removed is compared to the set identifier $\mathcal{F}$ to check for internal errors. A set identifier of a different follow set should never be popped if the error resynchronisation is operating correctly and the recovery actions are inserted at the correct positions in the grammar.

3. ScanFLW($\mathcal{F}$). Resynchronisation is provided by actions invoking this function at appropriate locations in the grammar. ScanFLW($\mathcal{F}$) pops the follow stack back to the set $\mathcal{F}$ and then searches the input stream, commencing with the symbol that caused the error, for the next symbol which is a member of an active follow set. If the matched symbol's set identifier is $\mathcal{F}$, **errec** commits to the recovery, performs the recovery action specified in the follow table and returns true as the function result. If the set identifiers are different, **errec** returns false. The returned value is used to optionally force yacc to pop its parse stack.

Errec is normally invisible to yacc, and merely relays the tokens produced by lex. When a parsing error occurs, **errec** intercepts the call to yyerror() and enters ERROR mode. Control is then returned to yacc which recovers by popping the parse stack until it can shift one of the recovery error actions. All error actions contain a call to ScanFLW($\mathcal{F}$). There are two forms of this call, one is used in situations where recovery to a particular level of the grammar is required, while the other provides scanning of simple linear constructs. The set $\mathcal{F}$ consists of all the symbols which can legally occur following the position where the error was detected. **Errec** pops its stack of follow sets to the first occurance of the set $\mathcal{F}$, and scans for the next input symbol which is a member of any follow set presently on its follow stack. If the symbol located is in the set $\mathcal{F}$ **errec** returns the value *true*, otherwise it returns the value *false*. If the value *true* is returned, yacc clears its error mode and resumes parsing. If the value *false* is returned, yacc remains in error mode removes a parse state from its stack, and commences a scan to locate an earlier *error* rule at an outer level of the grammar. This process continues until a location in the grammar is found where a ScanFLW($\mathcal{F}$) call locates a symbol in the set $\mathcal{F}$.

3 Follow Sets

Follow sets required by the **errec** filter, and used in the error recovery actions in the grammar, are defined as a multi–dimensioned array at the end of the yacc file. This array structure is pictorially represented in Figure 2.

Each major entry in the array represents a follow set. This follow set is composed of multiple elements, each of which defines an expected symbol, an operation on the input stream appropriate to the detection of that symbol at this stage in the parse, and a specific error message. The operation code associated with each element of the follow set has the following significance.

- Zero, (0), indicates that the follow symbol is to be returned on the next call of $yylex'()$, and therefore no modification of the input stream is required.

- A yacc token, (> 0), indicates that the supplied token is to be inserted into the input stream before the follow symbol. The next call to $yylex'()$ will return the supplied token and the subsequent call to $yylex'()$ will return the token at which the error was recovered.

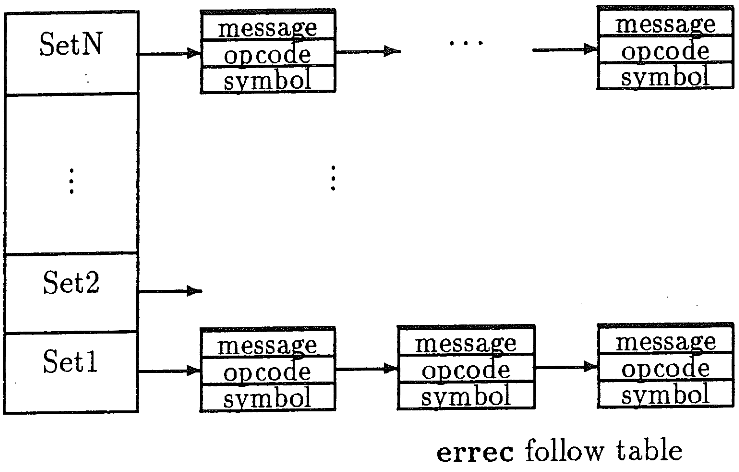


Figure 2: Errec Follow Set Table

- Other, (< 0), indicates that the follow symbol is to be removed from the input stream. The next call to *yylex'*() will return the next symbol from the input stream.

The errec follow stack holds pointers to these sets, and varies in depth with the depth of the parse. The use of a stack allows the detection of symbols valid in enclosing constructs, and the depth in the stack at which a follow symbol is located determines the level to which yacc must unwind it's parse to recover.

4 A Simple Example

Type declaration lists are parsed using a recursive grammar structure. In such lists the follow symbol set is generally composed of the element separator, or terminator, and any other symbols valid at that level of the parse. The separator and terminator must be handled at this level while the follow symbols associated with enclosing constructs are present deeper in the follow stack. The grammar segment used in the **Threads** compiler for type declarations is:

```
typedecls
    : /* empty */
    | typedecls typedecl

typedecl
    : ideq typeerr

typeerr
    : type TSEMICOLON
    | error { LeafScan( Dsemi ); }

ideq
    : TIDENT TEQ
    | error { LeafScan( Deqsemi ); }
```

Two error recovery actions have been inserted into this grammar segment to allow efficient recovery. The recovery process depends on the structure of the grammar, the placement of the error rules, and the definition of the follow sets. The follow sets required by the above grammar fragment are:

```
    { Deqsemi, "should be <simple identifier> =", {
        { TSEMICOLON, 0, "missing =" },
        { TEQ, -1, "skipped past =" },
        {0, 0, 0 } } },
    { Dsemi, "error in declaration", {
        { TSEMICOLON, -1, "skipped past ;" },
        {0, 0, 0 } } },
```

The error action in the rule ideq locates the next equality or semicolon symbol before reduction of the state occurs. This allows input processing to resume at the following type specification in the case where an identifier is missing, and an equality symbol is located before the next semicolon.

The error action in the rule typeerr locates the next semicolon on the input stream, and by forcing the reduction of a type declaration allows further type declarations to be processed.

As can be seen from this example the set $\mathcal{F}$ must be constructed to force the grammar along the rule. This is achieved through selecting key symbols which will allow components of the current rule to be reduced, and the use of LeafScan. LeafScan($\mathcal{F}$), pushes the set $\mathcal{F}$ onto the follow symbol stack, performs a Scan($\mathcal{F}$), and then removes the set.

5 A More Complex Example

Statement list error recovery is an example in which the follow set stack must be manipulated to reflect the nesting of the control structures in the input. The idea is that the parse should recover to a position above the level of the current structure if a relevant symbol is unexpectedly encountered. This requires the use of Pop, Push and Scan.

In most grammars where several different control structures are provided it is necessary to design several statement list grammars which ensure that the correct follow symbol sets are present during the parse. The grammar for a typical statement list is of the form:

```
statelist
    : statement
    | statelist TSEMICOLON statement
    | error { Scan( Fstatement ); }
```

The follow set for any statement list is defined by the control structure in which it is used. The semicolon statement separator will always be present, to allow recovery to the start of the next statement.

To ensure that all errors at the current parse level are processed correctly a terminal symbol must follow the statement list in the parent construct. This prevents the reduction of a statement list, on an error, before a valid follow symbol for the encompassing control structure has been located.

Any error which occurs during the processing of a statement list will be trapped by the local error rule, and either a semicolon or one of the follow symbols related to the control structure will be found. In the former case, parsing will continue with the next statement, while for the latter, the current rule will be reduced and the next rule in the encapsulating control structure will be expanded.

The use of statement lists with varying follow sets is best understood by considering examples from a grammar. The grammar used for the **Threads repeat** statement is:

```
repeatstatement
    : TREPEAT repeatloop TUNTIL { Pop( Frepeat ); }
      booleanexpr

repeatloop
    : { Push( Frepeat ); } repeatstatelist

repeatstatelist
    : statement
    | repeatstatelist TSEMICOLON statement
    | error { Scan( Frepeat ); }
```

While parsing any control structure the valid follow symbols comprise the key words denoting the start of a following statement, the keywords expected to follow the current position in the current construct, and the terminating symbol; in this case **end**.

As the **repeat** statement grammar is simple it requires only one recovery action located in the statement list. With this grammar in mind, and considering the other statements provided in the **Threads** language, the following set of symbols, and associated input stream actions, would be appropriate.

```
    { Frepeat, "invalid statement in repeat", {
        { TUNTIL,     0, "skipped to UNTIL" },
        { TIF,        TSEMICOLON, "probable missing semicolon" },
{ TWHILE,     TSEMICOLON, "probable missing semicolon" },
{ TLOOP,      TSEMICOLON, "probable missing semicolon" },
{ TFOR,       TSEMICOLON, "probable missing semicolon" },
{ TCASE,      TSEMICOLON, "probable missing semicolon" },
{ TWITH,      TSEMICOLON, "probable missing semicolon" },
{ TSEMICOLON, 0,          "skipped to ;" },
{ TEND,       0,          "skipped to END" },
{ 0, 0, 0 } } },
```

When a statement symbol is located the action for that symbol inserts a **semicolon** to allow the reduction of the erroneous statement, and correct parsing of the next statement. If the symbol **until** is detected it is left on the input stream to force the parser to reduce the statement list and then parse the boolean expression. A **semicolon** found during recovery is treated as the list separator, and must remain on the input stream. The detection of the symbol **end** indicates the termination of some higher level construct. This symbol is left on the input and yacc will discard states until the correct level of the grammar is located and parsing can resume.

The requirement that all exit transitions from a parse state be guarded by a terminal symbol results in an unusual grammar for the **if** statement.

```
ifstatement
    : TIF ifstate

ifstate
    : ifexpr { Push( Fthen ); } TTHEN thenstatelist elsepart

ifexpr
    : expression
    | error { LeafScan( Fif ); }

elsepart
    : TELSE { Pop( Fthen ); Push( Fstmnt ); }
      statelistend { Pop( Fstmnt ); } TEND
    | TELSIF { Pop( Fthen ); } ifstate
    | TEND { Pop( Fthen ); }

statelistend
    : statement
    | statelistend TSEMICOLON statement
    | error { Scan( Fstmnt ); }

thenstatelist
    : statement
    | thenstatelist TSEMICOLON statement
    | error { Scan( Fthen ); }
```

Insertion of follow set actions into this grammar segment was undertaken in several steps. First the major terminal symbols of the construct were identified. These were then divided into sets according to the position in which they occur in the grammar. For example, after the token TTHEN if an error occurs in the statement list valid follow symbols are ELSE, ELSIF,and END. With this in mind a follow set Fthen is created in the yacc file, and the action Push(Fthen) added just prior to entry into a thenstatelist.

Once the parse proceeds to an elsepart, these symbols are no longer valid, and so the set is removed using Pop(Fthen). If an error occurs, yacc remains in the thenstatelist, shifts the error token, and executes the associated action. When one of the follow symbols has been located, using the Scan(Fthen) in thenstatelist, yacc can reduce the state thenstatelist, and proceed to recognise an elsepart. Similar arguments apply to the positioning of the other Push and Pop actions.

Because the valid follow sets and input stream operations required for specific symbols varies depending on the position of the parse within the rule, it is necessary to define several follow sets.

```
{ Fif, "error in IF expression", {
    { TTHEN,  0,      "skipped past THEN" },
    { TELSIF, TTHEN, "probable missing THEN" },
    { TELSE,  TTHEN, "probable missing THEN" },
    { TIF,    TTHEN, "probable missing THEN" },
    { TWHILE, TTHEN, "probable missing THEN" },
    { TLOOP,  TTHEN, "probable missing THEN" },
    { TFOR,   TTHEN, "probable missing THEN" },
    { TCASE,  TTHEN, "probable missing THEN" },
    { TWITH,  TTHEN, "probable missing THEN" },
    { TSEMICOLON, TTHEN,       "skipped to ;" },
    { TEND,   TTHEN,       "skipped to END" },
    { 0, 0, 0 } } },
{ Fthen, "invalid then clause, possible missing keyword", {
    { TELSIF, 0, "skipped to ELSIF" },
    { TELSE,  0, "skipped to ELSIF" },
    { TEND,   0, "skipped to END" },
    { TIF,         TSEMICOLON, "probable missing semicolon" },
    { TWHILE,      TSEMICOLON, "probable missing semicolon" },
    { TLOOP,       TSEMICOLON, "probable missing semicolon" },
    { TFOR,        TSEMICOLON, "probable missing semicolon" },
    { TCASE,       TSEMICOLON, "probable missing semicolon" },
    { TWITH,       TSEMICOLON, "probable missing semicolon" },
    { TSEMICOLON, 0,       "skipped to ;" },
    { 0, 0, 0 } } },
{ Fstmnt, "invalid statement syntax", {
    { TIF,         TSEMICOLON, "probable missing semicolon" },
{ TWHILE,       TSEMICOLON, "probable missing semicolon" },
{ TLOOP,        TSEMICOLON, "probable missing semicolon" },
{ TFOR,         TSEMICOLON, "probable missing semicolon" },
{ TCASE,        TSEMICOLON, "probable missing semicolon" },
{ TWITH,        TSEMICOLON, "probable missing semicolon" },
{ TSEMICOLON, 0,            "skipped to ;" },
{ TEND,       0,         "skipped to END" },
{0, 0, 0 } } },
```

The set Fif is used to recover from errors in guard expressions. It allows error recovery to locate the key word **then**, or if the start of a statement, or branch, is suspected, a **then** is inserted to allow parsing of this rule to proceed.

There are two follow set required for statement lists in the **if** statement. This condition occurs because the symbol **end** is only valid in the statement list of an **else** clause. As is the case in the other follow sets described, the expected symbols are retained if discovered during error scanning allowing parsing to continue through the construct. Symbols indicating the suspected start of another statement cause the the insertion of a **semicolon** to allow the parsing of further statements in that block.

The LeafScan in the if expression rule uses a follow set containing the symbol THEN. Any error in an if expression will locate the next THEN in the input stream or any earlier symbol specified in a more global and still active follow set. This allows yacc to reduce the ifexpr, and continue parsing at the earliest possible point.

6 Summary

The previous discussion highlights the major considerations in applying error recovery using **errec** in yacc grammars. It is easy enough to generate grammars that work in most cases but occasionally result in strange transitions within the grammar. Our experience is that such faults have three major causes.

1. Inappropriate *error* transitions are visible as yacc unwinds its stack

2. Transitions out of nested or recursive structures occur on errors because their reduction is followed by actions before tokens are required.

3. The follow sets in LeafScans do not cover all possibilities or do not specify the correct recovery action to force the parse along a linear construct.

It is important to test the error recovery installed into a grammar by parsing input streams which force errors going 'in to' and 'out of' every rule which contains an *error* transition. Faults in the error procesing actions which might exist in a grammar only result in inappropriate error reports during error processing. Once such faults have been identified, the grammar can be restructured to remove them.

The implementation of the active filter **errec** allows yacc users to design flexible parsers with robust error recovery, using standard yacc actions. The development of follow sets, and insertion of our enhanced error processing facilities into the grammar, can be undertaken with most parsers implemented using these tools.

The demonstrated improvement in recovery, and accurate error reportage, makes yacc more attractive to the serious commercial designer.

The application of the techniques described in this report has produced a tremendous improvement in error recovery and greatly enhanced user interaction with the **Threads** compiler. The appendix contains a listing file for an input to the **Threads** compiler containing multiple errors (indicated by braces). In the example, the parse recovers correctly and diagnoses subsequent errors.

References

[1] I.D. MATHIESON and R.S. FRANCIS, "A Dynamic-Trace-Driven Simulator for Evaluating Parallelism", *Proceedings of the 21st Hawaii Int. Conf. on System Sciences: Vol. 1 (Architecture track)*, Kailua-Kona, HI: IEEE Computer Soc. Press, Jan. 1988, pp. 158–166.

[2] A.T. SCHREINER and H.G. FRIEDMAN JR, "Introduction to Compiler Construction with UNIX", Prentice–Hall, 1985.

[3] K.J. GOUGH, "Syntax Analysis and Software Tools", Addison–Wesley, 1988.

[4] "YACC- Yet Another Compiler Compiler", *Support Tools Guide UNIX System*, Western Electric, pp. 131–167, 1983.

[5] "LEX- Lexical Analyser Generator", *Support Tools Guide UNIX System*, Western Electric, pp. 113–124, 1983.

[6] "Error Recovery in LR Parsing: A Case Study Using YACC", S. IYENGAR, *The SoftLab Project (Internal Working Paper)*, Univeristy of North Carolina at Chapel Hill, 1986.

7 Appendix A: A Sample Listing

```
      1   MODULE ThenTest;
      2
      3   VAR
      4     one  :   REAL;
      5     two  :   INTEGER;
      6     T    :   REAL;
      7     c    :   INTEGER (*  ;  *)
ERROR   1                      <— error in declaration
      8     b (*  :  *) REAL;
                <— skipped past ;
      9     a    :   REAL (*  ;  *)
ERROR   2                  <— error in declaration
     10
     11   BEGIN
          <— skipped to BEGIN
     12
     13      IF ( T >= 12.0 ) THEN
     14         two (*  :  *) = 3;
ERROR   3            <— missing := or missing procedure declaration
ERROR   4            <— invalid then clause, possible missing keyword
                 <— skipped to ;
     15         one := 5.23 (*  ;  *)
ERROR   5                      <— invalid then clause, possible missing keyword
     16         T := 12.3
     17      ELSIF (*  T = 1.3  *) THEN
          <— skipped to ELSIF
ERROR   6          <— error in IF expression
                      <— skipped past THEN
     18         two := one;
     19         T := 43.0
ERROR   7                  <— invalid then clause, possible missing keyword
     20         one := 5.23
     21   (*  ELSIF  *) T = 9.3 THEN
     22         one := 5.23;
                <— skipped to ;
     23         T := 12.3
     24      ELSE
     25         two := 3;
     26         one := 5.23;
     27         T := 12.3;
     28      END;
     29
     30   END ThenTest.
     31
```

Macintosh® Toolbox Emulation under A/UX®

Kent Sandvik, Apple Computer Australia Pty. Ltd.
ksand@appleoz.oz.AU

Philip Cookson, Apple Computer Australia Pty. Ltd.

Abstract

This paper provides a technical overview of the implementation of the Macintosh Toolbox routines in a UNIX® environment, specifically A/UX Apple's implementation of the UNIX operating system on the Macintosh hardware platform. It is a case study of how a different environment and User Interface can be mapped on top of a UNIX operating system; and serves to demonstrate the flexibility of UNIX as an operating system platform. The key areas discussed include : differences between the Macintosh OS and A/UX execution environments, such as the method of accessing hardware devices, allocating system resources, especially memory management and memory addressing schemes; fundamental differences in the File System Structures, File Formats and File naming conventions; and differences between the Pascal calling conventions used by the Macintosh ROM and the C programming language typically used in A/UX. The implementation of the AppleTalk® protocols in the UNIX environment are also described.

Introduction

A/UX Version 1.1, is Apple's implementation of the UNIX operating system on the Macintosh platform[1,2]. A/UX is a POSIX and FIPS #151 (Federal Information Processing Standard) compliant implementation of AT&T's UNIX System V2.2 (with Berkeley BSD 4.2/3 enhancements). The implementation of the Macintosh User Interface and Toolbox under A/UX, is achieved with a Toolbox emulation module. This paper will examine how the different components of this Toolbox emulation are implemented, and will describe how the Macintosh Toolbox routines and A/UX (UNIX) libraries can be combined to develop application software which combines the functionality of UNIX with the graphically oriented Macintosh User Interface.

The Macintosh Toolbox

The Macintosh Toolbox is a library of routines built into the Macintosh ROM (Read Only Memory), which provide the support code required to generate the Macintosh Interface "look and feel" [3,4,5,6]. The Macintosh Toolbox can be logically divided into sets of functionally related routines which support various features, these are referred to as Managers of the features that they support. For example, the Window Manager which is a set of routines which manage the creation, activation, movement and resizing of windows.

The Macintosh Toolbox can be divided into two distinct groupings of routines :

(a) *Native Macintosh Operating System (OS) Routines*
These routines provide low level operating system functions such as : input and output to physical devices, memory management and interrupt handling.

(b) *Macintosh User Interface Toolbox Routines*
These routines provide the basic building blocks to implement the standard Macintosh User Interface.

Note that the a Macintosh User Interface Toolbox routine may itself make calls to the Native Macintosh OS routines.

The A/UX Toolbox

The A/UX Toolbox is a library of A/UX routines that enables a program running under A/UX to make direct calls to the Macintosh User Interface Toolbox routines and to remap all calls to the Native Macintosh OS Toolbox routines (including those made by the Macintosh User Interface Toolbox), to an equivalent set of A/UX (UNIX) operating system routines [7]. The A/UX Toolbox supports almost all of the Macintosh User Interface Managers and has functional equivalents for most of the Native Macintosh OS Managers, except for those which manage hardware devices; the function of these Managers is handled by conventional UNIX device drivers. The A/UX Toolbox enables UNIX applications which utilize the Macintosh User Interface to be developed and it's implementation is robust enough to allow well behaved Macintosh OS application binaries to run unchanged in the A/UX environment.

A/UX Application Development/Execution Options

Application software which uses the Macintosh User Interface may be developed under either the Macintosh Operating System or A/UX; and using the A/UX Toolbox library it is possible to execute applications under one environment that were originally developed under the other. The four possible application development/execution options are summarized in Figure1.

	Execution Environment	
Development Environment	MacOS	A/UX
MacOS	Develop, debug and run program with Macintosh Tools	Develop and debug program with Macintosh Tools. Transfer binary file to A/UX, then launch with A/UX Toolbox utility
A/UX	Develop and debug program with A/UX Tools. Use A/UX Toolbox calls. Transfer source file to Macintosh, then compile and link to run in native Mac environment	Develop, debug and run program with A/UX tools.

Figure 1

Because of the fundamental differences between the Macintosh Operating System and A/UX, not all of the Macintosh Toolbox routines are available through the A/UX Toolbox; therefore applications that are intended to run in both environments may only access Macintosh Toolbox ROM routines which are common to both. The Macintosh Finder is not currently implemented under A/UX; however most standard Macintosh Desk Accessories are supported, including the Chooser for access to network resources such as AppleTalk connected printers. Macintosh OS Desk accessories and application binaries which attempt to directly manipulate the hardware will not function properly under A/UX.

Launching Macintosh Application Binaries under A/UX

The *launch* command is used to start a Macintosh application. The user must pass the command the name of the application, together with any files that the user wants to open at the same time. An example of this is :

```
$ launch MacDraw "my picture"
```

Note that names containing spaces will need to be enclosed in quotes to prevent the shell from interpreting it as two separate files. The launch program is able to initialise the Standard Toolbox Managers to handle Macintosh OS application binaries which expect this. The *launch* command is also able to set up and maintain many of the Macintosh OS global variables, such as Ticks variable (time variable).

How the A/UX Toolbox works

When an application issues a call to one of the Macintosh User Interface Toolbox routines, the A/UX Toolbox intercepts the call and, if necessary, translates the parameters into a form usable by the Macintosh ROM routines (which use Pascal calling conventions); it then invokes the appropriate Macintosh ROM routine. When an application (or Macintosh User Interface Toolbox routine) issues a call to one of the Macintosh OS routines, the A/UX Toolbox diverts the call to a substitute routine in its own library, the A/UX Toolbox OS routines in turn make calls to the standard A/UX library routines to perform the A/UX equivalents of the Macintosh OS functions. The interaction between the A/UX Toolbox library and the Macintosh Toolbox is illustrated in Figure 2. Note that the Macintosh Operating System Toolbox routines are never used when running under A/UX.

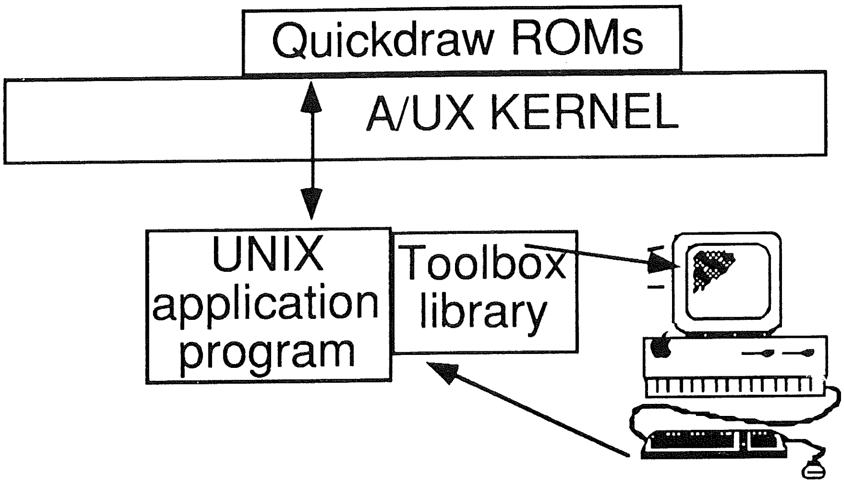


Figure 2

The remapping between the A/UX Toolbox library and the Macintosh Toolbox routines in ROM is performed by the A/UX daemon *toolboxdaemon* which is normally running when A/UX is active. This Toolbox daemon is responsible for setting up the shared memory used by the A/UX kernel and the A/UX Toolbox application environment, and for removing screen windows and data structures left in the shared data segment when an A/UX Toolbox application exits.

Communication between the Toolbox and the A/UX kernel

The A/UX kernel contains a special user interface device driver, */dev/uinter0*, that handles communication between an A/UX Toolbox application and the kernel. The A/UX Toolbox library routines make calls to this device driver the same way ordinary UNIX calls, for instance like calls to the device driver for a disk.

The user interface device driver, */dev/uinter0*, performs the following functions:

(a) When an application is started, the device driver establishes memory segments for the screen buffer and ROM code.

(b) The driver contains its own so called event queue handler (the Macintosh OS, like other window oriented systems are event driven, and have an event queue for events). This driver posts mouse and keyboard events.

(c) The device driver enables vertical retrace interrupts (this is the Macintosh OS interrupt for the system, one interrupt for each vertical retrace of the screen). It also tracks the cursor at each interrupt. This cursor data is in shared memory, so it is accessible to both the kernel and the application.

(d) During startup, the driver installs in shared memory a pointer to the A-line trap handler. When the kernel identifies an exception as a Macintosh ROM call, it copies the return address from the kernel stack to the user stack and invokes the trap address.

Once an A/UX Toolbox application is running, most A/UX Toolbox functions are called through Motorola 68020/68030 exception vectors, that is Motorola 68k opcodes in the range 0xA000 to 0xAFFF. These are known as A-line traps, because the instructions start with a hex A. Under the standard Macintosh OS these A-line trap calls are routed by the CPU to an exception handler that resides in the ROM. This exception handler uses a pair of dispatch tables (one for the Toolbox graphical routines and one for the OS routines) to route the A-line traps . Because all exceptions put the CPU into supervisor mode with Motorola 68k systems, an A-line trap in A/UX must be routed to the kernel. In A/UX and other UNIX systems ported to the Motorola 680XX architecture, the kernel runs in Supervisor mode and the user processes run in User mode. In the A/UX environment, trap handling must always be routed through the kernel itself. In a similar fashion to the Macintosh OS, the ROM dispatch tables in A/UX uses two sets of dispatch tables. If the trap represents a Macintosh User Interface routines, the table points to the relative ROM code. If the trap represents a Macintosh OS routine the table points to an alternative routine in user RAM. The Toolbox emulation sets up a memory map that is close enough to the one used by the native Macintosh OS that it fools the code in ROM and in applications. Before transferring control of a program, the Toolbox emulation makes several calls to the A/UX kernel to change the memory map making the ROM and the screen's frame buffer accessible.

An A/UX Toolbox application uses a special initialisation routine that opens the user interface device driver and issues a series of setup instructions before starting the program itself. The initialisation routine is */usr/lib/maccrt.0.o.* Each A/UX Toolbox application (including launch, see later), is linked to this file instead of */usr/lib/crt0.o*, which is normally used by UNIX programs. The A/UX Toolbox initialisation routine, */usr/lib/maccrt0.o*, performs the following functions:

(a) Invokes the BSD 4.2 signalling scheme as default (note that A/UX supports both System V, POSIX as BSD signals);

(b) Attaches the shared data segment;

(c) Opens the device driver and invokes the necessary initialisation steps;

(d) Initialises a low-memory segment that will hold the dispatch tables and Macintosh oriented global variables;

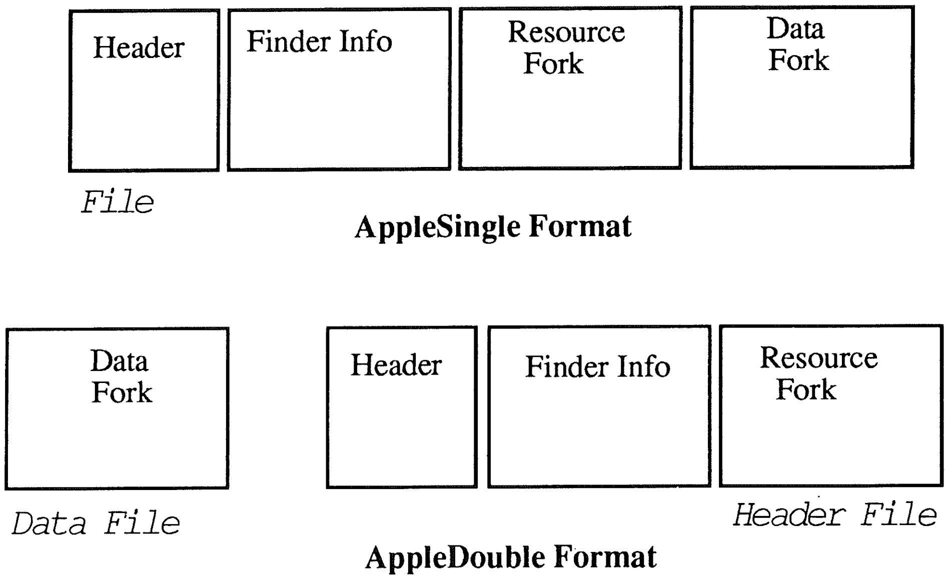
(e) Initalises various A/UX Toolbox modules as well as the dispatch table;

(f) Finally, it calls the application's main routine.

Differences between the Macintosh OS and A/UX File Systems

In the Macintosh OS, a file consists of two forks (binary file partitions) : the Data Fork and the Resource Fork. The Data Fork consists of application specific data (in the case of an application file), or User data/text (in the case of a document file). The Resource Fork holds all of the resources associated with the application or document, such as Window sizes and types, Static Text, Menu Items and, in the case of an application, the acutal application CODE resources. Although a file can contain two forks, one or the other of the two forks may be empty. The Macintosh OS also records other information about the file, such as its position on the Macintosh Desktop. This information is stored in a separate (invisible) file called the Desktop file. Under the Macintosh OS this file is represented by a single icon on the Finder Desktop.

The A/UX file system is a true UNIX file system, and it makes no distinction between the Data and Resource Forks; and the A/UX directory structure has no provision for storing the Desktop file information. In order to allow Macintosh OS files to coexist in a UNIX file system, Apple developed two standard file format structures for representing Macintosh OS files : *AppleSingle* Format and *AppleDouble* Format. Figure 3 illustrates the typical contents and structure of *AppleSingle* and *AppleDouble* file formats.



AppleSingle Format

AppleDouble Format

Figure 3

Another important difference between the Macintosh OS file system and the A/UX file system is in the definition of a volume. Under the Macintosh OS, each physical volume appears as a single-rooted tree which can be separately mounted. Under A/UX (and UNIX in general), all physical volumes are mounted at a specific mount directories in the directory hierarchy, that is the file system appears as a single-rooted tree. Thus under A/UX, all volumes appear as part of a single-rooted tree whose name is / (slash). This volume cannot be mounted, unmounted, or taken on- or off line, though component physical volumes mounted beneath the root directory can be mounted and umounted using the standard UNIX utilities *mount* and *umount* .

File name conventions are different under the two operating systems; A/UX file names are case sensitive and limited to 14 characters. Macintosh file names can be up to 32 characters long, and are not case sensitive. Directory names are separated by a / under UNIX, and by : (colon) under the Macintosh OS. These differences only affect hard-coded file names. The A/UX Toolbox File Manager emulation understands the UNIX file system structure, so a Macintosh application binary application can access files through the Standard Macintosh File Dialog Box, even though they exist under a UNIX File System. The A/UX Toolbox library contains a number of utility routines for transferring files between the A/UX and Macintosh Operating System (*hfx*, *mfs*) and for manipulating their formats (*fcnvt*, *settc*).

Differences in the Execution Environment

The Macintosh Operating System was originally conceived as a single user operating system and in this environment individual applications and utilities have complete control over the system's resources and are able to directly access the hardware. However, in a UNIX environment, the kernel arbitrates all access to the hardware and System resources including memory allocation.

Memory Management

The Memory Management under A/UX consists of code completely different from the code used by the Macintosh OS. A/UX uses the standard UNIX system calls and libraries, such as *malloc* and *free* , to do the work of the Memory Manager emulation. These routines simply maintain a circular list of memory blocks. When a Macintosh application makes an allocation request, the system searches the list to find the first free block large enough. If there is no free block large enough, the *sbrk* system call asks the kernel to extend the heap, thus creating more virtual memory. Thus this is a normal memory allocation scheme under UNIX , with the addition of virtual memory for Macintosh binaries. None these routines perform memory compaction, so they are much simpler and faster than the algorithms used in the Macintosh OS Memory Manager. There are however cases when compaction helps with paged systems. Without compaction, the memory fragmentation can cause paging even when there is enough physical memory. Applications can start paging and run more slowly even if they allocate less memory than the system has. However, there are advantages to the UNIX memory allocation scheme. It does not spend time moving blocks around in an effort to compact the heap.

Some of the Macintosh OS Memory Manager functions are not relevant in a virtual memory environment. In a virtual memory environment, it is not clear what value the call to determine the amount of free memory available should return. When a Macintosh application is launched, A/UX behaves as if there is 1 Mb of free memory. This is a compromise, because A/UX has virtual memory and the Macintosh Memory Manager was designed to manage a finite amount of physical memory. With the new SIZE resource a Macintosh application could be defined to have allocated a certain amount of heap space. This is tunable and the user could tune the size of the heap to larger than 1 Mb, and thus make use of the A/UX virtual memory capability.

The internal data structures are different as well. Macintosh OS has master pointers that point to the block of data reserved in memory. These master pointers under Macintosh Os are 4 bytes long (long word). A/UX master pointers are 8 bytes long (2 long words), this is due to the fact that A/UX uses the full 32-bit address space, whereas the current version of the Macintosh OS only uses 24 bit addressing (refer to Figure 4).

the current version of the Macintosh OS only uses 24 bit addressing (refer to Figure 4).

One of the major reasons that Macintosh OS application binaries fail to run under A/UX is that they assume that the master pointer is 4 bytes. There exist flag bits in the master pointer that for instance indicate if the reserved data block is locked or not. When Macintosh OS application binaries try to toggle these flag bits directly, without using the correct A-line trap call, they indadvertantly toggle the address information and cause the application process to abort.

Another issue is heap zones, the Macintosh OS has two heap zones, one for the system and one for the application. A/UX does not distinguish between the application and the system heap zones. Macintosh applications which make assumptions about the order or location of various sections of the memory will not be able to run under A/UX. The shared common data and code segments are managed by the A/UX Toolbox daemon, which runs in the background. The launch utility initializes these segments since it is replacing some of the function of the Macintosh OS and the Finder™.

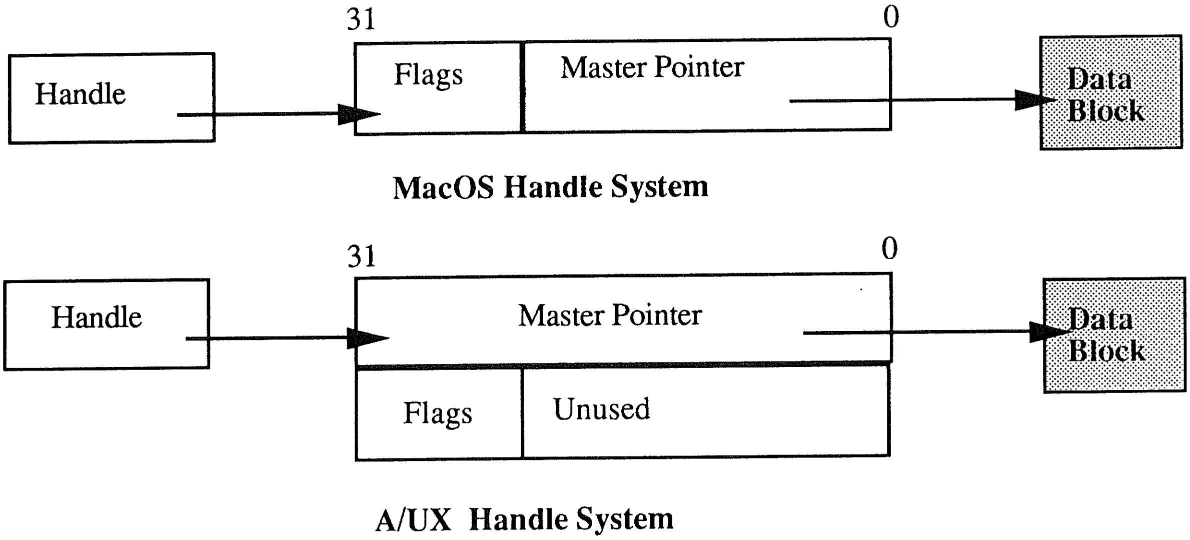


Figure 4

Process Scheduling

Macintosh applications are event-driven, meaning that there is an event loop that waits for different kinds of events and triggers off functions according to the events. Typically the main body of an application is a loop that uses the *GetNextEvent* trap to look for events, such as keystrokes or mouse clicks. This busy-looping model works well under a single user, single task operating system. However, under a multi-tasking operating system such as UNIX, busy looping time takes processor time away from any other processes that are currently running. Also, the process scheduler makes scheduling decisions based on how much CPU time a process has recently used. Thus with constant CPU usage the priority level decreases dramatically. To solve this problem, both under Macintosh OS and A/UX Toolbox emulation, a new trap called WaitNextEvent has been added. This trap is similar to *GetNextEvent*, except that it yields the CPU until an event is available. Note that this works if you only make use of Macintosh Managers. If you write a combination of UNIX libraries and Macintosh libraries this trap only checks for Macintosh oriented events. With the Berkeley select(2) system call the program is able to monitor both Macintosh I/O activities as UNIX I/O activities. The *select()* utility examines a set of file descriptors that the programmer specifies through bit masks. For instance if the programmer wants to have the interface device driver checked (/dev/uinter0), he opens the file, gets a fd (file descriptor) and specifies this fd as part of the bit mask. Thus *select* will now check for user interface events from the Macintosh application point of view.

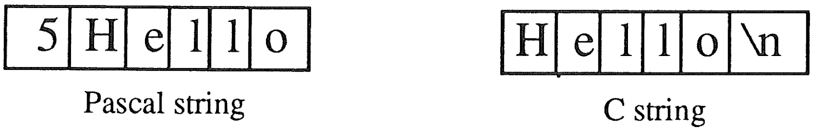
Even though A/UX is full multi-user, multi-tasking operating system, the A/UX Toolbox can only currently support one Macintosh binary at at time. This is due to the fact that the Multi-Layer Manager (used in the Macintosh OS MultiFinder), is not implemented under the current version of A/UX, (Version 1.1). Desk Accessories, small utility programs capable of being launched from within a Macintosh application, are however concurrently supported.

Emulation of Global Variables

The standard Macintosh environment includes a set of global variables used by different parts of the system. These are stored in the so called low memory of the memory map. To make room for these global variables, an A/UX toolbox application compiled in A/UX is linked at the virtual memory address 0x0040 0000. The launch program for executing Macintosh binary code (itself an A/UX Toolbox application), is linked at this address. Normally non-toolbox UNIX programs are typically linked at memory address 0x0000 0000. Note that not all global variables are supported under A/UX. In general, variables not related to hardware are supported.

Differences between C and Pascal Calling Conventions

Ordinarily, the A/UX Toolbox handles the difference between the C conventions used by A/UX and the Pascal calling conventions that are used by the Macintosh Toolbox ROM routines. However if you are writing your own definition functions or filter functions or are making direct use of data in structures, you must explicitly take these differences into account. The C and Pascal conventions differ in six primary ways, how strings are stored, how parameters are ordered, how the parameter types are stored, how points are passed, how function results are passed, and how registers are used. In C, strings are stored as an array of characters, of any length, terminated by the null byte ('\0'). In Pascal, strings start with a byte that specifies the length of the string, and the string plus length byte cannot be longer than 256 characters. When the Pascal string length is specified explicitly, a Pascal string is not terminated by a null byte. With two library routines, *c2topstr* and *p2cstr* , it is possible to translate between these two formats. Refer to figure 5.



Pascal string

C string

Figure 5

Parameters in C functions are evaluated right to left and are pushed into the stack in the order they are evaluated. Parameters in Pascal functions are evaluated left to right and are pushed into the stack in the order they are evaluated. The Pascal language always passes 4-byte structures by value rather than by pointer, unless the structure is declared as a VAR. Pascal treats registers D0, D1, D2, A0 and A1 as scratch registers. All other registers are reserved. A/UX C treats only registers C0, D1, A0 and A1 as scratch registers. An A/UX Toolbox routine automatically saves and restores register D2 when using ROM code. There are other registers in the Toolbox emulation reserved for different functions. Register A5 is the global frame pointer, register A6 is the local frame pointer, and register A7 is the stack pointer.

When necessary, the A/UX Toolbox interface routines convert C program calls to a form usable by the ROM, and then convert the ROM's output to a form usable by the C program. The A/UX Toolbox routines that perform this conversion have three parts, the entry conversion code, the A-line trap, and the exit conversion code.

The libraries under A/UX include two version of all routines that take strings or points or return strings. One version, spelled exactly like it appears in the Apple documentation (Inside Macintosh), uses Pascal string format and point passing notations. The second version, spelled in all lowercase letters, uses C string format and point passing conventions. This lowercase version converts input parameters from C format to Pascal format before passing them to the ROM and converts return values back to C format. Both versions use interface routines to adjust for other differences in parameter passing and return-value conventions.

Environment Tuning and Manipulation

There are many ways to change the Macintosh emulation environment, either with the help of shell variables , or global variables set in programs. For instance by defining noEvents as 1 inside the program, the system does not initialise the Event Manager. This status lets mouse and keyboard events go through normal UNIX event handling channels instead of having them captured by the Event Manager

The Toolbox emulation uses several shell environment variables to modify it's actions under certain circumstances. Most of these variables are useful only during program development and testing. These A/UX Toolbox environment variables are set and read like other environment variables. For instance if TBCORE is true the Toolbox causes a core dump if a fatal error occurs. Compared with Macintosh OS the kernel OS memory is protected. If TBRAM is set the ROM code is copied into a memory segment when a program run. This variable lets the programmer set a breakpoint in the ROM code for debugging. TBTRAP indicates that the system writes debugging information to standard error every time an A-line trap is executed.

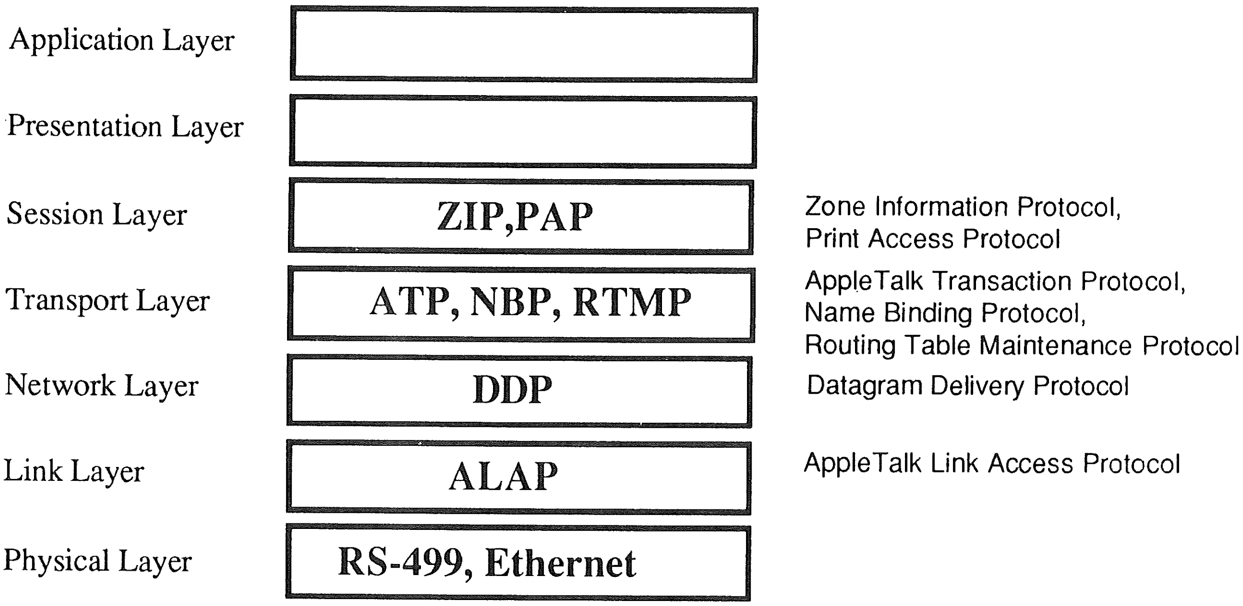
It is possible to use UNIX system calls in an application that is originally developed under the Macintosh OS. This application could end up as a binary file that can be executed in both environments. The basic procedure is to translate the UNIX system calls into assembly language routines and make these routines available to the compiler. First the programmer has to define the assembly line sequence that is generated by the A/UX compiler when it encounters the system call that is required. Then the programmer will create an assembly line routine that performs the same function. This call should be conditionally be inserted into the application. Finally the program has to check for the value of bit 9 in the global variable HWCflFlag, a 16-bit word at memory location 0x0B22. If this bit is 1, the A/UX operating system is active and the program can use this UNIX routine. If the bit is 0, the application is running under Macintosh OS and the application should use an equivalent Macintosh OS routine.

AppleTalk® Networking

AppleTalk is the general name for the AppleTalk suite of protocols that conform to the ISO OSI seven layer model. When AppleTalk is used over serial lines and RS-499 connections it is called LocalTalk®, when AppleTalk is used over Ethernet is it called EtherTalk® (and when using it over Token Ring systems it is called Token Talk®!).

The A/UX AppleTalk support is split between the AppleTalk Manager emulation and a special device driver written for A/UX. The AppleTalk manager redirects AppleTalk protocol handling to this special device driver. Because of the high processing requirements of both AppleTalk and the A/UX multitasking when using LocalTalk and serial cables,LocalTalk under A/UX currently requires a separate coprocessor card to handle the AppleTalk protocols. The main reason for this requirement is the high rate of interrupts coming from the network to the motherboard where the hardware for LocalTalk is situated (as well as a 2 byte buffer in the SCC chip).

AppleTalk is implemented under A/UX as a protocol stack, consisting of a set of layers, with one or more protocols per layer. This set of protocols corresponds roughly to the layers of the OSI model. At present not all of the AppleTalk protocols are supported, the supported protocols are illustrated in Figure 6.



Application Layer		
Presentation Layer		
Session Layer	ZIP,PAP	Zone Information Protocol, Print Access Protocol
Transport Layer	ATP, NBP, RTMP	AppleTalk Transaction Protocol, Name Binding Protocol, Routing Table Maintenance Protocol
Network Layer	DDP	Datagram Delivery Protocol
Link Layer	ALAP	AppleTalk Link Access Protocol
Physical Layer	RS-499, Ethernet	

Figure 6

Most AppleTalk protocol layers are implemented as Streams modules. The two exceptions are the DDP and ALAP layers, which are implemented as Streams drivers. The majority of applications require the programmer to push one or more modules into the open stream in order to achieve the proper layering for that application. Figure 7 describes the A/UX implementation of AppleTalk protocols.

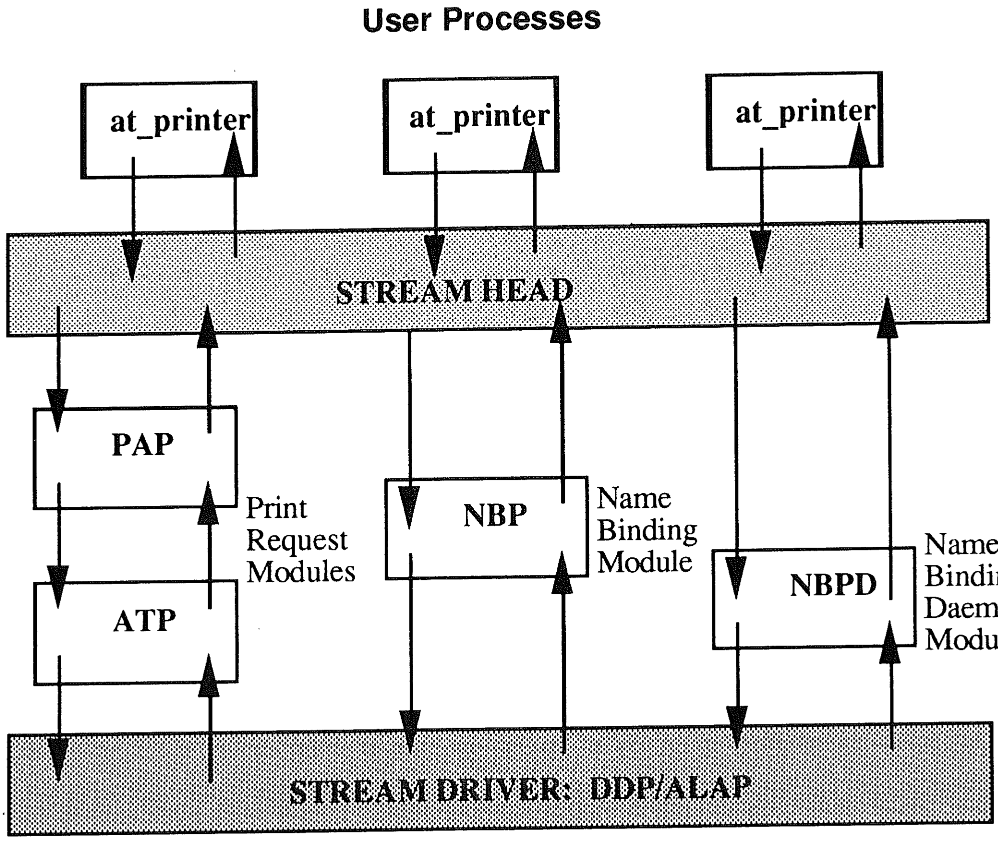


Figure 7

The first application, *at_printer,* shows the configuration for communicating with a network print server. Note that the ATP module must be pushed before the PAP module. While it is possible to reverse the pushing order, unpredictable results can occur if this is done!

The second and third application, *at_cho_prn* and *at_npd,* are normally used together. When AppleTalk is brought up, a special application daemon, *nbpd,* is invoked. It opens an AppleTalk socket and pushes the module *at_nbpd* into the stream. This application daemon is used by subsequent applications to open a socket and push the module *at_nbp* into the stream. Modules *at_nbp* and *at_nbpd* communicate at the ALAP level to complete users' requests for name binding information.

The C interfaces and libraries for all of these protocols are provided in the standard A/UX system distribution.The lowest layer (ALAP/DDP) C interfaces are normally used for new network testing and development, such as building a new layer using TCP/IP on top of DDP. The AppleTalk A/UX routines automatically set up and invoke the correct *ioctl* requests that are necessary for most AppleTalk requirements. While the ioctls give the programmer more control than the AppleTalk library routines, they require a much greater understanding of the A/UX implementation of AppleTalk.

Final Words

The difficult task of migrating applications developed for a single user, single task, graphically based environment to an UNIX platform has been described, and the particular implementation of the Macintosh User Interface in the A/UX environment has been discussed. The impact of the Macintosh emulation on the performance of the A/UX system in running Macintosh OS application binaries is very small, and in some cases Macintosh application binaries run faster under UNIX (mostly due to the file buffer cashing, virtual memory and faster memory allocation, that is provided by a UNIX operating system).

The successful implementation of the Macintosh OS User Interface under A/UX demonstrate the inherent flexibility of the UNIX operating system to adapt to different hardware platforms and characteristics and to implement a range of user interface technologies.

The remaining work to be done concerning Macintosh User Interface emulation under A/UX is to continue to write emulation modules to for existing Macintosh Toolbox Managers that have not yet been implemented, and to develop support for new Managers as the Macintosh OS itself evolves. The overall aim is to provide an emulation which will allow the complete implementation of the Macintosh Finder/MultiFinder™ Desktop interface to supplement the standard UNIX command shell interfaces.

References

[1] **A/UX System Overview** 1989 Apple Computer Inc.

[2] **Technical Introduction to the Macintosh Family** 1987, Addison-Wesley. Apple Computer Inc. An introduction to the hardware and software design of the Macintosh family of computers.

[3 **Programmer's Introduction to the Macintosh Family** 1987, Addison-Wesley. Apple Computer Inc. A programmer's technical overview of the Macintosh system introducing the most important Macintosh Toolbox and Macintosh Operating System features.

[4] **Inside Macintosh Volumes I through III**, Addison-Wesley, 1985. Apple Computer Inc. A complete description of the architecture and operation of the 128K and 512K Macintosh, including the ROM routines.

[5] **Inside Macintosh Volume IV**, Addison-Wesley, 1986. Apple Computer Inc. An update to Volumes I through III, covering the Macintosh 512K Enhanced and the Macintosh Plus.

[6] **Inside Macintosh Volume V**, Addison-Wesley, 1987. Apple Computer Inc. An update to Volumes I through IV, covering the Macintosh SE and Macintosh II.

[7] **A/UX Toolbox : Macintosh ROM Interface** 1989 Apple Computer Inc. A description of the Macintosh ROM interface.

About the Authors

Kent Sandvik

Kent Sandvik is the A/UX Systems Engineer with Apple Computer Australia Pty. Ltd. Prior to joining Apple, Kent worked as a UNIX development engineer with UNISYS. He holds a BSEE (Vasa Technical College, Finland).

Philip Cookson

Philip Cookson is a Systems Engineer with Apple Computer Australia Pty. Ltd. Prior to joining Apple, Philip was a lecturer in Computer Science at the Australian Defence Force Academy. He holds a Masters degree in Computational Physics (Monash University) and a Post Graduate Diploma in Computer Science (Melbourne University).

Special Thanks

Special thanks to Kevin Nietzke, Apple Computer Australia Pty. Ltd. for assistance in the production of this paper.

TradeMark Acknowledgements

UNIX is a registered trademark of AT&T Information Systems.
A/UX, Apple, Macintosh, AppleTalk, EtherTalk, TokenTalk and LocalTalk are registered trademarks of Apple Computer Inc.

User Mode File Servers

Bruce Janson
bruce@basser.cs.su.oz

Basser Department of Computer Science
University of Sydney

1. Introduction

In many older operating systems, including UNIX, the kernel, acting as a single large subroutine library, is the primary service provider. Modification of such monolithic kernels is an error prone activity. Nevertheless, such kernels do need to be modified whenever a new service is to be provided. Recent releases of the UNIX operating system have begun to provide support for *network file systems* (e.g. **netfs**[Wei84a] in AT&T Bell Laboratories' Eighth and Ninth Edition UNIX, **NFS**[San84a, San87a] from Sun Microsystems', and **RFS**[Rif86a] in AT&T's System V UNIX). Similar functionality may also be found in many non- UNIX operating systems[Ric79a, R.79a, Dio80a, K.81a, Tan81a, A.83a, Zwa84a, R.85a, B.89a]. Network file systems extend the hierarchical file system name space by allowing remote file systems to appear as subtrees within the local file system. In this article we show how the basis of a user mode file server which supports such a network file system may also be used as a server for a variety of other, non-standard, file system types.

Under UNIX and UNIX-like operating systems, file system access requests from user processes are usually served by the operating system kernel of the machine on which the requesting process executes. We call a file system supported in this way a *kernel mode file system*. In Figure 1, a kernel mode file server *FS* receives file system access requests from a user process *p*, converts these to I/O requests and passes them on to a disk drive *d* via disk device interface *i*.

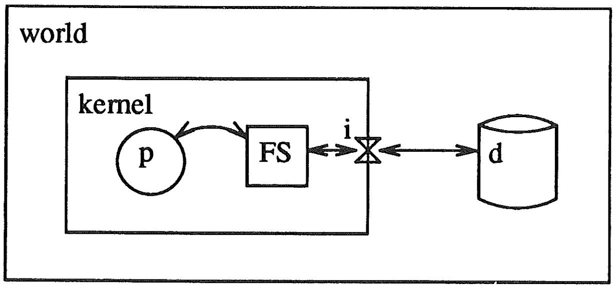


Figure 1 - FS: A Kernel Mode File Server

In a *user mode file system* such requests are not served directly by the kernel but are instead redirected to and served by another process. In Figure 2, a user mode file server *ufs* receives file system access requests from user process *p*, converts these to other file system access requests and passes them on to kernel mode file server *FS* which converts these to I/O requests and passes them on to disk drive *d* via disk device interface *i*.

Access requests for either kernel or user mode file systems may be served on the machine on which the process is running or they may be redirected to another machine. We label the former case a *local file system* and the latter case a *network* or *remote file system*. Clearly, these independent characteristics may be combined and we can speak of, say, a *local user mode file system* or a *remote kernel mode file system*. In Figure 3, remote user mode file server *rfs* running under kernel *k1* receives file system access requests via a network connection from user process *p*, running under kernel *k0*, converts these to other file system access requests and passes them on to local kernel mode file server *FS* which converts these to I/O requests and passes them on to disk drive *d* via disk device interface *i*. *n0* and *n1* are network device interfaces.

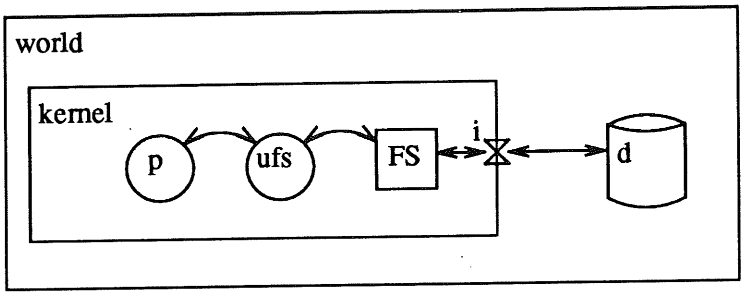


Figure 2 - ufs: A User Mode File Server

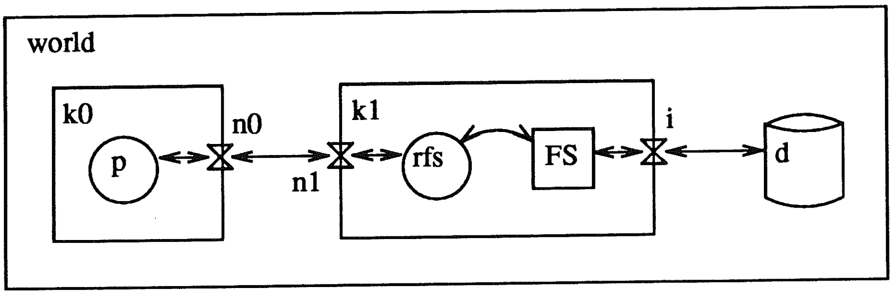


Figure 3 - rfs: A Remote User Mode File Server

Most network environments allow the establishment of (virtual) connections between any pair of nodes. In particular, a *loopback* connection from a machine to itself is usually permitted. In such an environment, a local user mode file server is easily implemented as a remote user mode file server which uses a network self-connection. Given these two preconditions: 1) the existence of a remote user mode file server and 2) sufficient flexibility of extant networking software, it becomes straightforward to implement a general user mode file server which may execute either locally or remotely. On *basser* (a DEC VAX 11/780 running Eighth Edition UNIX), this first condition was met by Weinberger's *netfs* network file server and the second was met by Ritchie's *stream* I/O system[Rit84a] and Morris and Presotto's streams-based Internet protocol software[J.87a]. Initial applications using user mode file systems were constructed on this platform. In UNIX environments other than Eighth or Ninth Edition UNIX, similar functionality may be obtained through the use of the NFS protocol coupled with a suitably enhanced version of Shand's user mode NFS server[Sha88a] and Berkeley's Internet networking software[Lef82a]. This second implementation platform has the advantage of being more portable as it is based on freely available software and on published, widely distributed protocol descriptions[Mic87a, Mic88a, Mic89a]. Our later user mode file servers have been based on this NFS platform.

Several kinds of user mode file servers have been implemented. Our initial application was essentially Weinberger's Eighth Edition *netfs* server and provided support for a remote file system. The communication medium used was an RS-232 asynchronous line. More recently, we have implemented (NFS-based) file servers running under Eighth Edition UNIX (on a VAX 11/780), under SunOS 3.5.2 (on a Sun 3/260), under MIPSCo's System V.3 RISC/os UNIX (on a MIPSCo M/120) and under IBM's AIX Operating System 2.1.2 (on an IBM RT System Model 125). However, our main interest lies not in the implementation of such standard UNIX file servers, but rather in the discovery and implementation of interesting and/or useful non-standard file systems. For example, such file systems include a kernel name list file system and a line printer spooler file system. These and other examples will be described in detail below.

As Presotto and Ritchie have observed ".. it requires a sophisticated server to simulate a file system."[Pre85a]. We have managed to conceal some of this sophistication by hiding details common to a large class of file servers within a library. To use this library a server must provide definitions for a small set of functions. These functions are (essentially) replacements for a subset of UNIX system calls which access the file system. Thus, implementation of a user mode file server is reduced to the reimplemention of a subset of the familiar UNIX system calls.

In the remainder of this article we survey existing types of file servers, both local and remote and implemented in both kernel and user mode. We also mention aspects of the NFS protocol and describe our own (NFS-based) user mode file server in some detail. We show how the basis of such a server may be used as a skeleton from which may be easily constructed other, non-standard, user mode file servers. We give examples of its use in this way and present the results of some performance measurements. As a file system served by a user mode process is usually noticably slower than one served by an operating system kernel, we expect that user mode file systems will be most useful in applications where flexibility and ease of implementation are more important than performance.

2. Standard and Non-Standard File Systems

By a *standard* UNIX file system we mean a file system which *behaves in the normal way*. Unfortunately, there has been little published which describes formally what "behaves in the normal way" means (although Morgan and Sufrin[Mor83a] and Declerfayt et. al.[Dec88a] provide formal but incomplete specifications while the System V Interface Definition[ATT86a], the "POSIX" standard[IEE88a] and the various online manual entries are useful, albeit informal guides). Thus UNIX file system behaviour is defined by the various, differing file system implementations themselves.

File servers for standard UNIX file systems usually convert file system access requests into one or more disk I/O requests. Most file system access requests originate as system calls from user processes. The I/O requests are served directly by kernel software device drivers and eventually by disk device hardware.

When we refer to a *non-standard* file system we mean a file system which breaks the rules in some sense, perhaps by imposing unexpected restrictions or by causing interesting side-effects to accompany file system accesses or by remapping file system structure or by reformatting file data. In general, file system requests need not be mapped onto disk I/O operations at all and may instead be interpreted in a variety of ways. User mode file servers allow greater flexibility of implementation of file system semantics than their kernel mode counterparts.

3. The User Mode File Server

3.1. Kernel Support

3.1.1. The Client-Kernel Interface

The interface between client processes and a file system supported by a user mode file server is simply the subset of the normal UNIX system calls which access the file system. Ideally, a user process should not be able to determine whether its file system specific system calls are being served by a local or remote kernel or user process.

3.1.2. The File System Switch

Just below the client-kernel interface is the file system switch. The file system switch is an array of file system information structures. Each structure describes a particular type of file system by providing a set of pointers to functions which are that file system type's implemention of the basic file system operations. UNIX files are represented by *inode* s (aka. vnodes, gnodes, rnodes) within the kernel. Which file system type to apply is a function of the inode referenced in the system call.

The user mode file system is reached via the NFS entry of the file system switch. The functions pointed to by this file system switch entry do little more than encapsulate the client's data, write it to a communications channel - in our case a UDP[Pos80a] socket - await a reply and pass this reply, suitably interpreted, back to the client process. The encapsulation method referred to here is called External Data Representation (XDR)[Mic87a]. It ensures that all data on the network is expressed in an agreed normal form. The software which handles the details of sending a remote procedure call and interpreting its reply is called Remote Procedure Call (RPC)[Mic88a]. We note that on this, the client side, the NFS, RPC and XDR routines are implemented within the kernel.

3.1.3. The Kernel-Server Interface

The server process reads from a UDP channel, decodes the data stream using XDR then interprets the result as an RPC call and further interprets the RPC data as an NFS request. After performing the requested

NFS operation a reply is encoded and written back to the UDP channel.

3.2. Server Process Architecture

The server process consists of three sections. The first of these is the RPC/XDR library and the **rpcgen** program distributed by Sun. The second is an NFS-specific server library derived from Shand's unfsd. The final section is the set of plug-in system call replacements which define the characteristics of the particular file server. The plug-in system call replacements are:

u_access, u_chmod, u_chown, u_close, u_creat, u_link, u_lseek, u_lstat, u_mkdir, u_open, u_read, u_readlink, u_rename, u_rmdir, u_stat, u_symlink, u_truncate, u_unlink, u_utime, u_write

And in fact there are seven other routines which must also be supplied which perform any necessary initialisation, handle directory operations and return global information about the file system in question:

u_closedir, u_opendir, u_readdir, u_seekdir, u_telldir, u_init, u_statfs

Each plug-in replacement routine (except u_init and u_statfs) replaces the system call or directory library function named by deleting the "u_" prefix from the replacement's name.

4. Two Applications

4.1. A Standard User Mode File System

4.1.1. Description

This file server is perhaps the simplest which may be constructed using the above machinery. Each "u_" routine simply passes its arguments on uninterpreted to its system call/directory library function namesake and returns the result returned by that call as its result. For example, the u_stat function definition looks like this:

```
int
u_stat(path, buf)
char          *path;
struct stat   *buf;
{
        return stat(path, buf);
}
```

4.1.2. Performance

Here is a transcript of a simple experiment comparing the performance of a local user mode NFS server with that of a local kernel mode file server (on *harpo* - an idle MIPS M/120):

```
% # Start the user mode NFS daemon.
% ./nfsd &
25115
% # Create a directory on which we will mount a file system.
% mkdir mnt
% # Where are we?
% hostname
harpo
% # Mount a copy of the local root.
% ./mount -o intr,hard harpo:/ ‘pwd‘/mnt
% # Find a big enough file.
% ls -las /unix
3001 -rwxr-x---  2 root    daemon  1523276 Apr 18 23:32 /unix
% # Time reading the file the fast way.
% time dd bs=32k < /unix > /dev/null
46+1 records in
46+1 records out

real     1.6
user     0.0
sys      0.5
% # Do it again now that the file is in the buffer cache.
% time dd bs=32k < /unix > /dev/null
46+1 records in
46+1 records out

real     0.3
user     0.0
sys      0.3
% # Now time reading the same file while it is still
% # in the buffer cache via the user mode NFS server.
% time dd bs=32k < mnt/unix > /dev/null
46+1 records in
46+1 records out

real    24.0
user     0.0
sys      3.0
%
```

Next, we compare the performance of a remote user mode NFS server with that of a remote kernel mode file server (on *harpo* and *chico* - two idle MIPS M/120’s):

```
% # Where are we?
% hostname
chico
% # Mount a copy of harpo's root.
% ./mount -o intr,hard harpo:/ 'pwd'/mnt
% # Time reading the same file using a
% # remote kernel mode NFS server.
% time dd bs=32k < /n/harpo/unix > /dev/null
46+1 records in
46+1 records out

real     4.2
user     0.0
sys      0.6
% # Do it again just to check that the file is
% # in the local buffer cache.  Actually, it is
% # also in the remote buffer cache.
% time dd bs=32k < /n/harpo/unix > /dev/null
46+1 records in
46+1 records out

real     0.3
user     0.0
sys      0.3
% # Now time reading the same file while it is still
% # in the remote buffer cache via the user mode NFS server.
% time dd bs=32k < mnt/unix > /dev/null
46+1 records in
46+1 records out

real    18.3
user     0.0
sys      2.1
%
```

Interestingly, reading from a remote user mode file server is faster than reading from a local user mode file server. A plausible explanation of this behaviour would be that in the remote case, the processing is shared between two processors rather than one and some overlapping of processing occurs.

4.2. Kernel Name List File System

The executable image of the UNIX kernel is typically stored in the file system as "/unix". As with other UNIX executables a symbol table (aka. name list) is usually appended to this file. During normal system operation the special file "/dev/kmem" allows suitably priviledged processes to access the (dynamic) contents of kernel memory.

File systems served by the kernel name list user mode file server appear to be a single directory containing one regular file for every name in the kernel name list. The modes, uid's and gid's of the files are just copies of the mode, uid and gid of /dev/kmem. The size of each file is just the number of bytes separating its name from the following name in the kernel name list. The last modified, access and inode-change times always reflect the current time. read()'s, write()'s and lseek()'s of these files are translated into equivalent operations on /dev/kmem but offset by the address of the file's name in the kernel name list. Here are some examples of its use:

```
% hostname
harpo
% # An instance of the daemon is already running on mango.
% # We will look at the mango kernel's memory from harpo.
% ./mount -o intr,hard mango:/ /nl
% cd /nl
% ls
9$0000000000
9$0000000001
9$0000000002
9$0000000003
9$0000000004
9$0000000009
BADADDRFRM
BSD_alloc
BSD_alloccg
BSD_alloccgblk
BSD_badblock
BSD_blkpref
BSD_bmap
BSD_clrblock
..
% # And so on -- there are 2521 entries in mango's
% # /unix name list so we wont list them all.
% # The first four bytes at the location named "time" contain
% # the number of seconds since Jan 1, 1970.
% ls -las time
  1 -rw-r-----   1 root     sys          8 Jul 10 02:51 time
% # (its size is 8 because it is really a "struct timeval")
% # We use a command called "printf" here to format some data.
% # printf is, in this incarnation, a program that reads
% # from its standard input and formats it according to
% # its first argument -- very like printf(3).
% printf "%d\n" < time
616006620
% # printf has one or two extra features.
% printf "%T\n" < time
Mon Jul 10 02:57:51 EST 1989
% # Time waits for no man.
% printf "%T\n" < time
Mon Jul 10 02:58:18 EST 1989
% # Let's make it generally available
% chmod o+r time
% ln -s /nl/time /dev/date
% printf "%T\n" < /dev/date
Mon Jul 10 03:01:28 EST 1989
%
```

Clearly this file system can be used for more than just reading the time. It is a useful aid during kernel debugging -- shell scripts can be used instead of one-off feedback programs. Note also that it allows remote access to a system object that is normally only accessible from the local machine.

5. Some Potential Applications

5.1. Line Printer Spooler

5.1.1. Description

This is the line printer spooling file system. We assume that such a file system is mounted on /lp and that, as is customary at Basser, our line printers are identified by single letter names.

To spool a file called **paper** for printing on line printer 9 one might type something equivalent to:

```
% cp paper /lp/9/paper.'cat /lp/9/nextname'
```

The contents of the file **/lp/9/nextname** appears to be a new short, string guaranteed to be unique (by the line printer file system server) each time it is read. This unique name used as a suffix provides a simple way to avoid unfortunate name clashes. Any file copied to the directory **/lp/9** is automatically queued for printing. Files are queued in the order in which they arrive.

To inspect the queue for line printer 9 one might type:

```
% ls -last /lp/9
```

The file at the bottom of the listing would be the file currently being printed and the position of **paper.*** would indicate its position in the queue.

Files in **/lp/9** are automatically removed as soon as they have been printed. However, if it is desired to stop a print job one would type something like:

```
% rm /lp/9/paper.*
```

The directory **/lp/9** would allow file creation by anyone but would allow only the owner of a file (and root, of course) to unlink it. Adopting recent custom this would be represented by causing the *sticky* and public write bits to be set in the modes of these directories.

To enquire of the status of the actual printer device one could type:

```
% cat /lp/9/status
```

The output would be a printable ASCII representation describing the current status of line printer 9.

Finally, we note that once again, network access is implicit in this scheme.

5.2. Password File (and Other Potentially Single-File Data Bases)

File systems supported by this user mode server contain just one (regular) file. This is a distributed password file daemon. When applied to a distributed password file the user mode file server supports just a single file mounted at /etc/passwd. One host in a network is designated the holder of the real password file for all. This real password file might be /etc/real_passwd on the designated host. All hosts, including the server, mount this file on their /etc/passwd.

When the remote password file is not mounted (as would be the case just after a reboot but before the various rc scripts have been run) a minimal password file can exist as /etc/passwd containing perhaps just a root entry. When this file is subsequently mounted over by the remote password file it will disappear (as per normal mount behaviour) until the remote password file is unmounted again.

It is not clear that the current behaviour of NFS mounts during remote server crashes is appropriate for such a scheme. Nevertheless, the return to a world where password files (and hosts files, services files, protocols files, ethers files, etc.) live within the file system and accurately reflect the data they are supposed to contain, would appear to be a very worthwhile goal.

5.3. Logging or Multi-Version File Systems

A logging file system automatically and transparently keeps a log of changes made to any files within it. The granularity of such logging may be very coarse - a differential file comparison performed whenever the file system is unmounted - or it may be very fine - an exhaustive transaction trail which records every file system operation performed.

5.4. Redundant File System

This file system type is similar to the logging file system but the intention here is to replicate file system data in such a way as to minimise file system data loss during a system failure and also to ensure data consistency after such a failure. This might be accomplished by duplicating data at different points in the local file system or it may be done by writing the data to some remote device or file system.

5.5. Network Bin

The contents of the executables files in this kind of file system depend on the architecture of the client's machine. In a heterogeneous network environment, for example, executables can be made to always match their executing CPU.

5.6. Customisable File System

In many UNIX programming environments workstations are connected via a network to other workstations and perhaps to machines designated as shared file servers. Often the policy which decides when to swap or page a process out of the local workstation and off to some remote point in the network is out of the control of the person using the workstation. It may be that the installed policy is not even generally known by users of such workstations. A user mode file server which either bypassed the kernel file servers completely and interfaced with the raw devices directly or which allowed the workstation user to at least experiment with file caching might be worthwhile.

6. Problem Areas and Work In Progress

The NFS protocol, although a useful workhorse is far from ideal as a basis for user mode file servers. Its deficiencies are well known and will not be completely reiterated here although we will mention some of its problems. Nevertheless, it continues to gain popularity - at Basser it was the only network file system protocol available on more than one machine.

Under NFS many of the basic file system operations are assumed to be idempotent. They are not. This is the source of the sometimes puzzling error messages which indicate that a file system operation has failed when in fact it has not. Some NFS implementations attempt to maintain a log of recent file system operations and check for duplicates when in doubt. This reduces the probability of failure but does not eliminate it.

An NFS file server is not supposed to preserve any state information other than that actually kept in the file system. Unfortunately, in a standard (non-network) UNIX file system, important state information is kept in the in-core inode structures which never reach the disk. Thus permission checking under NFS necessarily differs from standard UNIX file systems and unlinking open files is not really possible. Similarly, the NFS protocol provides no **open** or **close** primitives - such primitives would require state information to be kept - thus it is not possible to even determine when a file is no longer open in any process, let alone when it has no more file system links.

NFS servers cannot distinguish between a file being read for execution and a file being read normally. Thus, read access is always allowed even if a file's mode only permits execute access. Similarly, NFS servers cannot distinguish between a file which was once readable but is no longer and a file which is still readable to processes in which it is open despite its current mode. Thus, read access is always allowed to the owner of a file even if the file's mode does not permit it.

NFS and some other network file system protocols do not support operations on files which may block indefinitely. In particular, remote special files are not handled and neither are remote *ioctl* (2)'s.

ioctl's are also difficult to implement because they often expect or generate amounts of data which are strongly dependent upon the particular *ioctl* call. RFS solves this problem by allowing the server to engage in a two-way dialog with the client host.

Certain NFS file system operations (e.g. create) carry potentially more initialisation commands than a user mode file server can effect in a single UNIX system call. For example, if a file is to be created with uid 7 and the user mode server is executing under root's uid then the server either has to change its uid temporarily while it is creates the file or it has to perform the operation in two stages (a create with uid 0 and then a chown to uid 7). Kernel mode file servers do not have this problem as they are free to enforce much stronger locking and thus ensure that intermediate states are never seen or, they can very easily change their uid for the duration of the transaction.

At present the user mode server assumes that all participating machines share the same uid and gid space. This is not entirely satisfactory, particularly as the distance between client and host increases.

To gain full benefit from user mode file servers it should be possible for non-root users to mount and unmount such file systems. However, at this time *mount* (2) and *umount* (2) are root-only system calls.

Some kernels do not allow mounts to succeed on files other than directories. This is unfortunate as it further restricts the applicability of user mode file servers. Furthermore, many kernels do not allow mounts using sockets from other than the INET domain.

The pseudo-inumbers generated internally by the file handle package within the current user mode file server libraries are not guaranteed to be unique. This is particularly true when several instances of user mode file servers have been mounted from different machines.

Benchmarking of user mode file servers is made unnecessarily difficult under NFS due to enthusistic client caching of the results of previous NFS calls.

Further work will involve attempts to fix some of the problems mentioned above as well as the implementation of Nowicki's suggested NFS performance enhancements[Now89a].

7. Related Work

We mention here, in varying levels of detail, a selection of existing UNIX user mode file system types which might be considered noteworthy.

7.1. neta

Weinberger's initial, remote user mode file system. It is restricted to Eighth Edition UNIX. This was followed by ...

7.2. netb

Weinberger's second, remote user mode file system. Again, this is restricted to Ninth Edition UNIX.

7.3. /n/face

This is Pike's user mode *face server* file system[Pik85a]. It provides a database of 48x48x1 and 512x512x8 ikons stored in what looks like a normal (read-only) file system. It uses Weinberger's network protocol (and is therefore restricted to Eighth and Ninth Edition UNIX). It divides the work between a remote user mode file server and a local user mode intermediary. Data is actually stored on only one machine.

7.4. /mail

This is Hitz and Honeyman's mail delivery agent file system[Hit86a]. It is suppored by a local user mode file server which uses a protocol similar to Weinberger's network protocol (it required a new file system switch entry). The primary difference in the protocol is that it passes the whole namei pathname at once rather than component-by-component. Interestingly, different files may have the same name. Again, it is restricted to Eighth and Ninth Edition UNIX.

7.5. Watchdogs

Essentially this is a user mode file server which relies on fairly extensive kernel support[N.88a]. The subset of file system operations handled by the server is easily configurable.

7.6. TFS

Hendricks' (Sun Microsystems) translucent file service[Hen88a]. This is a user mode NFS-based file server. It allows per user views of file system hierarchies.

7.7. The Newcastle Connection

The Newcastle Connection is a network file system based on trapping certain file system calls within client programs and redirecting them to be served on a remote host[F.86a]. It uses many, short-lived server processes.

7.8. Extended File System

This is Masscomp's implementation of a network file system[T.85a]. It uses many kernel mode processes on the server machine. There is a meta-server (called the "lifeguard") which redirects file system requests to the actual server processe ("agents"). These agents change their own u_area to match the client process before serving the client's system call. The underlying communication protocol is their own RDP (Reliable Datagram Protocol) which apparently avoids the overhead of setting up virtual circuits for each transaction but still guarantees in-order, reliable delivery of messages and timely indication of communication failures. Interface is via the SOCK_RDM 4.2 BSD socket mechanism and Internet addressing domain. It also allows easy independent reply demultiplexing to separate clients. They insist that uids must be the same on different machines. It is stateful. They have modified some of the system calls to cope with a network environment. Their *stat*(2) returns a structure with an extra machine id field. (So cp can still detect when it is about to copy a file over itself.) Similarly, *ustat*(2) has became *rustat*(2).

7.9. Pseudo-File-Systems

Pseudo-file-systems are implemented as user mode file servers in the Sprite distributed file system[B.89a]. Their first application was as a local translator from Sprite file system operations to remote NFS file system operations.

8. Conclusion

We have presented a method of reducing the complexity of the task of implementing a user mode file server under UNIX. File servers based on Sun's popular NFS protocol may take advantage of this technique. It should also be possible to apply this method to environments using other network protocols. We have described and implemented several user mode file servers and have described other existing user mode file servers. Such file servers provide a convenient and flexible means of extending the functionality of the UNIX file system name space.

9. References

References

A.83a.Malcolm, Michael A. and Dyment, D., "Experience Designing the Waterloo Port User Interface," *Proceedings of the 1983 ACM Conference on Personal and Small Computers*, pp. 168-175, 1983.

ATT86a.
ATT, *The System V Interface Definition, Issue 2, Volumes I, II and III*, AT&T Customer Information Center, Customer Service Representative, P.O. Box 19901, Indianapolis, IN 46219, U.S.A., +1-317-352-8557, 1986.

B.89a.Welch, Brent B. and Ousterhout, John K., *Pseudo-File-Systems*, Computer Science Division, Electrical Engineering and Computer Sciences, University of California, Berkeley, California, April, 1989.

Dec88a.
Declerfayt, O., Demeuse, B., Wautier, F., Schobbens, P.-Y., and Milgrom, E., "Precise Standards through Formal Specifications: A Case Study: The UNIX File System," *EUUG Autumn Conference Proceedings*, Cascais, Portugal, 1988.

Dio80a.
Dion, Jeremy, *The Cambridge File Server*, pp. 26-35, >= 1980. Computer Laboratory, University of Cambridge

F.86a.Marshall, Lindsay F., "Remote File Systems Are Not Enough!," *EUUG Conference Proceedings*, September, 1986.

Hen88a.
Hendricks, David, *The Translucent File Service*, Sun Microsystems, Inc, >=1988.

Hit86a.Hitz, D. and Honeyman, P., "A Mail File System for Eighth Edition UNIX," *USENIX Association Summer Conference Proceedings*, Atlanta, June, 1986.

IEE88a.
IEEE, *1003.1 "POSIX" Full Use Standard*, IEEE P1003 Portable Operating System Interface for Computer Environments Committee, IEEE Service Center, 445 Hoes Ln., Piscataway, NJ 08854, 201-981-0060, October, 1988.

J.8?a. Morris, Robert J. and Presotto, David L., 198?. Apparently Morris did the initial port from 4.2bsd as part of a summer job and Presotto has worked on it subsequently.

K.81a.Ousterhout, John K., *Medusa A Distributed Operating System,* Computer Science: Distributed Database Systems, No. 1, UMI Research Press, 1981.

Lef82a.
Leffler, S.J. and Fabry, R.S. et. al., *4.2 BSD Interprocess Communication Primer*, Computer Systems Research Group, Electrical Engineering and Computer Sciences, University of California, Berkeley, California, June, 1982.

Mic87a.
Sun Microsystems, Inc, *XDR: External Data Representation Standard,* RFC 1014, SRI Network Information Center, Menlo Park, CA, June, 1987.

Mic88a.
Sun Microsystems, Inc, *RPC: Remote Procedure Call Protocol Specification*, RFC 1057, SRI Network Information Center, Menlo Park, CA, June, 1988.

Mic89a.
Sun Microsystems, Inc, *NFS: Network File System Protocol Specification,* RFC 1094, SRI Network Information Center, Menlo Park, CA, March, 1989.

Mor83a.
Morgan, Carroll and Sufrin, Bernard, *Specification of the Unix Filing System,* Programming Research Group, Oxford University Computing Laboratory, Oxford, July, 1983.

N.88a.Bershad, Brian N. and Pinkerton, C. Brian, ''Watchdogs - Extending the UNIX File System,'' *USENIX Winter Conference Proceedings*, Dallas, Texas, February, 1988.

Now89a.
Nowicki, Bill, ''Transport Issues in the Network File System,'' *Computer Communication Review*, vol. 19, no. 2, pp. 16-20, April, 1989.

Pik85a.
Pike, R. and Presotto, D.L., ''Face the Nation,'' *USENIX Summer Conference Proceedings*, Portland, OR, June, 1985.

Pos80a.
Postel, Jon, *UDP: User Datagram Protocol*, RFC 768, SRI Network Information Center, Menlo Park, CA, August, 1980.

Pre85a.
Presotto, D.L. and Ritchie, D.M., ''Interprocess Communication in the Eighth Edition UNIX System,'' *USENIX Conference Proceedings*, June, 1985.

R.79a.Cheriton, David R., Malcolm, Michael A., Melen, Lawrence S., and Sager, Gary R., ''Thoth, a Portable Real-Time Operating System,'' *Communications of the ACM*, vol. 22, no. 2, pp. 105-115, February, 1979.

R.85a.Cheriton, David R. and Zwaenepoel, Willy, ''Distributed Process Groups in the V Kernel,'' *ACM Transactions on Computer Systems*, vol. 14, pp. 77-107, 1985.

Ric79a.
Richards, M., Aylward, A.R., Bond, P., Evans, R.D., and Knight, B.J., ''TRIPOS - A Portable Operating System for Mini-computers,'' *Software - Practice and Experience*, vol. 9, pp. 513-526, 1979.

Rif86a.Rifkin, A.P., Forbes, M.P., Hamilton, Richard L., Sabrio, M., Shah, S., and Yueh, K., ''RFS Architectural Overview,'' *USENIX Conference Proceedings*, Atlanta, June, 1986.

Rit84a.Ritchie, D.M., ''A Stream Input-Output System,'' *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, October, 1984.

San84a.
Sandberg, Russel, *Sun Network File System Protocol Specification,* Sun Microsystems, Inc., Technical Report, 1984.

San87a.
Sandberg, Russel, ''The Sun Network File System: Design, Implementation and Experience,''

Australian Unix Users Group Newsletter, vol. 8, no. 5, pp. 96-111, 1987.

Sha88a.
Shand, Mark, 1988. Public domain software posted to Usenet news group comp.sources.misc(?)

T.85a.Cole, Clement T., Flinn, Perry B., and Atlas, Alan B., "An Implementation of an Extended File System For UNIX," *USENIX Conference Proceedings*, MASSCOMP Software Engineering, June, 1985.

Tan81a.
Tanenbaum, A.S. and Mullender, S.J., "An Overview of the Amoeba Distributed Operating System," *ACM Operating Systems Review*, vol. 15, no. 2, pp. 51-64, July, 1981.

Wei84a.
Weinberger, P.J., "The Version 8 Network File System," *USENIX Summer Conference Proceedings*, Salt Lake City, Utah, June, 1984.

Zwa84a.
Zwaenepoel, Willy and Lantz, Keith A., "Perseus: Retrospective on a Portable Operating System," *Software - Practice and Experience*, vol. 14, pp. 31-48, 1984.

Ace: a syntax-driven C preprocessor

James Gosling

Abstract

*This document presents the **ace** preprocessor for C programs. Unlike **cpp**, which operates on characters, **ace** operates on syntax trees. The user specifies syntax trees which are used as templates against which program fragments are matched. Positive matches cause trees to be rewritten. **Ace** can be used as a special-purpose optimizer that can be controlled by the programmer.*

1. Introduction

Ace is a preprocessor for C programs, a sort of "macro processor" in the spirit of cpp. Unlike cpp, ace "macros" do not operate on strings of characters, they operate on syntax trees. Instead of macros, ace has *rules*. A rule consists of a pattern in the form of a syntax tree, and a replacement, also a syntax tree. These rules cause instances of the pattern to be replaced in the program tree. Ace reads in a C source program, constructs its syntax tree, performs any replacements, and writes the tree out as a C program.

The design of ace was motivated by a desire to perform transformations on algorithms to improve their performance, without impacting their readability and maintainability. As an example of the kind of thing it can do, consider this code fragment:

```
for (i = 0; i<10; i++)
      if(da > 0) A[i] ++;
      else A[i] --;
```

The test in the inner loop, `da>0`, is loop-invariant: it doesn't change from one trip through the loop to another. This loop can be rewritten as:

```
if (da > 0)
      for (i = 0; i<10; i++) A[i]++;
else
      for (i = 0; i<10; i++) A[i]--;
```

Eliminating this is a form of code motion that no compilers use since it leads to an exponential growth in code size, but in some cases it is justified.The exponential code growth comes from the fact that much of the body of the loop is replicated. Each one of these invariant tests that is removed from a loop body doubles the size of the code. But in some circumstances, like the inner loop of a vector drawing routine, this cost in code expansion is gladly paid.

Often the way that people deal with optimizations like this is that they expand the code by hand with a text editor. But once this is done, the original code is destroyed and the relationships between the parts is obscured. If you wanted to change, for example, the upper bound on the loop from 10 to 11, you would now have to change it in two places rather than one. When the loop body becomes large, and the number of such special cases becomes large, doing this transformation by hand becomes a major undertaking. Using ace, the second piece of code can be generated from the first by prefixing it by one line, like so:

```
$pullout(da > 0)
      for (i = 0; i<10; i++)
            if(da > 0) A[i] ++;
            else A[i] --;
```

2. Rules

Ace understands the syntax of C with a few additions. To avoid name clashes, *ace* considers `$` to be a legal character in identifiers. By convention, names specific to *ace* start with `$`. The most important addition is the *`$replace`* statement. It looks like this:

```
$replace statement1 $with statement2
```

This defines a rule that causes all occurrences of *statement1* to be replaced by *statement2*. Statement1 is a *template*. Since expressions are syntactically statements in C, `$replace` can be used to define replacements for expressions as well as statements:

```
$replace sqrt(4); $with 2;
a = sqrt(4);
```

`sqrt(4)` in the second line will be replaced by `2`. *`$replace`* definitions are applied to the rest of the file. When multiple templates match a tree, the one from the earliest *`$replace`* statement applies.

The templates can contain unbound meta variables that match anything. These are the symbols `$0`, `$1`, `$2` ... For example:

```
$replace !($0<$1); $with $0>=$1;
if (!(a<b+3)) ....
```

`!(a<b+3)` will be replaced by `a>=b+3`. Sometimes it's necessary to restrict the matches of these meta variables. One restriction is to trees that are side-effect free. Such matches are indicated with `$f0`, `$f1`, `$f2`... For example:

```
$replace $f0 = $0; $with $0;
a = a;
*p++ = *p++;
```

The first assignment statement, `a=a` would be replaced by `a`, since evaluating `a` has no side effects. The second assignment wouldn't be replaced since `p++` has a side effect. *Ace* is reasonably clever about statements and will eliminate those that have no side effects, so replacing `a=a` in a statement context with `a` causes the whole statement to be eliminated. But `b=2*(a=a)` would become `b=2*a`.

`$LET` is a special function that *ace* understands:

$$\texttt{\$LET}(a_0,a_1,a_2,a_3,\ldots,a_n)$$

This temporarily defines rules that replace a_0 with a_1, a_2 with a_3, ... in a_n. For example `$LET(a, 1, a+b)` would expand to `1+b`.

As a useful piece of syntactic sugar, *ace* extends the C language with *prefix statements*. A prefix statement is just a statement that has been prefixed by something that looks like a procedure call. The statement becomes a last argument to the procedure call. This procedure call will normally be transformed into a statement by *ace* rules. These prefix statements are defined with the *`$defprefix`* procedure:

```
$defprefix($let, $LET);
```

This defines *`$let`* to be a prefix statement that is replaced by a call of the `$LET` function:

```
$let(a,1) {
      b = a+1;
      print(a);
}
```

becomes

```
        {
                b = 2;
                print(1);
        }
```

Unlike *cpp*, one doesn't have to insert parenthesis all over the place in ace rules:

```
        $replace angle($0); $with $0->angle;
```

When *ace* rewrites `angle(*p)`, `$0` matches `*p`. When the syntax tree is finally printed, *ace* correctly inserts parenthesis based on operator priorities to yield `(*p)->angle`.

3. Time/Space tradeoffs

Ace has a facility that allows you to make time/space tradeoffs:

```
        $tradeoff(code1,code2)
```

picks either code$_1$ or code$_2$ depending on a time/space tradeoff. Presumably, code$_1$ and code$_2$ perform the same computation, only in different ways. *Ace* will estimate the time used by each code fragment and the space used. To aid in its computation of a time estimate, it needs to know the probability that branches will go one way or another and it needs to know the expected number of trips through a loop. $Replace is used to tell *ace* the probability that a boolean expression will be true:

```
        $replace $P(e); $with c;
```

This says that the probability that *e* will be true is *c*. These probability specifications are used in **if** and **switch** statements to determine the probability of execution of each clause. If *ace* cannot determine the probability of some expression, it will assume that all clauses are equally likely.

The expected number of trips through a loop is specified by prefixing it with $trips:

```
        $trips(100)
                for(i=0; i<100; i++) { ... }
```

This tells *ace* that the for loop is expected to be executed about 100 times.

Based on this information, and two parameters, *ace* will pick one of the two code fragment parameters of $tradeoff to replace $tradeoff. The two parameters are *pthresh* and *mingain*. *Pthresh* is a probability threshold: If the probability of executing a particular $tradeoff exceeds *pthresh* then the time-efficient code fragment will be chosen, otherwise the space-efficient fragment will be chosen. *mingain* specifies a minimum percentage time gain. If the code fragment chosen by *pthresh* doesn't gain at least *mingain* percent in time, the space-efficient code fragment will be chosen.

4. Rule Application Order

Once the source has been parsed, *Ace* applies the rules that have been defined. They are applied by traversing the parse tree from the root. It attempts to apply rules to a node *both* before *and* after the rules have been applied to its subnodes. Rules are applied before so that rules which change the ruleset (e.g. those that use let) behave properly. They're also applied after in case the transformed subnodes expose new opportunities for rule application.

This can cause some subtle interactions. Consider the following:

```
        $replace log2(2); $with 1;
        $replace constant($c0); $with 1;
        $replace constant($0); $with 0;
        constant(1)
        constant(a)
        constant(log2(2))
```

The intent of the constant rule is that it should evaluate to true if it's argument is a constant, and false otherwise. Because of the ordering of the definitions, this should be so: If the argument is a constant, the second rule will be applied, yielding true. If it isn't, the third will be applied, yielding false. Constant(1) evaluates to true, and constant(a) evaluates to false. But constant(log2(2)) evaluates to false because when the rules are applied before reducing the argument to constant, the third rule is used since log2(2) isn't a constant. There is a way around this:

```
$replaceafter constant($0); with 0;
```

If a rule is defined with $replaceafter rather than with $replace, it will only be applied to a node after its arguments have been reduced.

5. Building on ace

Using ace, we can define DeMorgan's law:

```
$replace ! ($0 && $1); $with ! $0 || !$1;
$replace ! ($0 || $1); $with ! $0 && !$1;
```

Then there are a number of rules that are used in conjunction with these:

```
$replace ! ($0 == $1);          $with $0 != $1;
$replace ! ($0 != $1);          $with $0 == $1;
$replace ! ($0 >= $1);          $with $0 < $1;
$replace ! ($0 <= $1);          $with $0 > $1;
$replace ! ($0 > $1);           $with $0 <= $1;
$replace ! ($0 < $1);           $with $0 >= $1;
$replace ! !$0;                 $with $0;
```

Now we can define a more subtle rule:

```
$replace $assume($0, $1);
$with $let($0, 1, !$0, 0, $1);
```

This rule causes code fragment $1 to be compiled, assuming that $0 is true, and that !$0 is false:

```
$defprefix($ASSUME, $assume);
$ASSUME(a<0) {
      if (a>=0) print("true");
      else print ("false");
}
```

This is transformed into just print ("false"). $Assume will replace a<0 with 1, and ! (a<0) with 0. Other rules ensure that ! (a<0) is replaced by a>=0, which is replaced by 0. The *if* now has a constant to test, so the true clause is eliminated. There are many special cases of the assume rule that can be defined. Because of the ordering rule of template matching, they have to precede the general rule:

```
$replace $assume($0 == $1, $2);
$with $let($0, $1, $2);
$replace $assume($0 < $1, $2);
$with $let($0<$1, 1,
      $0>=$1, 0,
      $0<=$1, 1,
      $0==$1, 0,
      $0 != $1, 1, $2);
$replace $assume($0 > $1, $2);
$with $let($0>$1, 1,
      $0<=$1, 0,
      $0>=$1, 1,
      $0==$1, 0,
      $0 != $1, 1, $2);
$replace $assume($0 && $1, $2);
$with $assume($0, $assume($1, $2));
```

Using these, we can now define the *$pullout* prefix that was used at the beginning of this description:

```
$defprefix($pullout, $pulloute);
$defprefix($LET, $let);
$replace $pulloute($0, $2);
$with       if($0)
                  $assume($0, $2);
            else
                  $assume(!$0, $2);
```

In other words, to pull a test out of a code fragment, perform the test, and when it's true execute the code assuming that the test is true, and when it's false, execute the code assuming that it's false. Repeating the example from the beginning of this paper:

```
$pullout(da > 0)
      for (i = 0; i<10; i++)
            if(da > 0) A[i] ++;
            else A[i] --;
```

This gets expanded to:

```
if (da>0)
      $ASSUME(da > 0)
            for (i = 0; i<10; i++)
                  if(da > 0) A[i] ++;
                  else A[i] --;
else  $ASSUME(!(da > 0))
            for (i = 0; i<10; i++)
                  if(da > 0) A[i] ++;
                  else A[i] --;
```

The ASSUME clauses cause this to become:

```
if (da>0)
            for (i = 0; i<10; i++)
                  if(1) A[i] ++;
                  else A[i] --;
else
            for (i = 0; i<10; i++)
                  if(0) A[i] ++;
                  else A[i] --;
```

And constant collapsing eliminates the inner **if**s, yielding:

```
if (da>0)
            for (i = 0; i<10; i++)
                  A[i] ++;
else
            for (i = 0; i<10; i++)
                  A[i] --;
```

Tradeoff() can be used to make $pullout() much more powerful:

```
$replace $pulloute($0, $2);
$with       $tradeoff($0      ? $assume($0, $2)
                              : $assume(!$0, $2),
                        $2);
```

This pulls $0 out of $2 only if there is a useful performance gain. **Note:** *ace* treats ? and **if** identically.

6. More mundane uses

Ace can be used much like cpp to define procedures that are expanded inline, with the added attraction that it's easy to define special cases for parameters that are known at compile time:

```
/* bool(a,b,op) executes a boolean operation specified by op */
$replace bool($0,$1,0); $with $0|$1;
$replace bool($0,$1,1); $with $0&$1;
```

It can, of course, match parameters other than constants:

```
$replace Get_Context($0, CTX_CLIP); $with $0->CTX_CLIP;
```

It can provide default parameters to procedure calls:

```
$replace atan2($0); $with atan2($0, 1);
```

It can be used to define iterators for special data types:

```
/* shape iterator */
$defprefix($scanshape, $scanshapee);
$replace $scanshapee($0, $1);/* (shape, code) */
$with {
      register ENTRY *sptr = Get_Shape($0, SHAPE_DATA);
      short        x0,
                   y0,
                   x1,
                   y1;
      while (*sptr != Y_EOL) {
            y0 = *sptr++;
            y1 = *sptr++;
            while (*sptr != X_EOL) {
                  x0 = *sptr++;
                  x1 = *sptr++;
                  $1;
            }
            sptr++;
      }
}
```

This example uses the special prefix syntax so that it can be invoked this way:

```
$scanshape(thisshape) {
      printf("%d, %d, %d, %d\n", x0, y0, x1, y1);
      FillRectangle(x0, y0, x1, y1);
}
```

7. Acknowledgements

A special thanks to Patrick Naughton for being a guinea pig user. And to the whole "Shapes" team that put up with mysterious things breaking.

Appendix 1. Invoking *ace*

```
ace [-time] [-space] [-lnc] [-nln] [-pthresh n]
      [-mingain m] [-Qpath path] [-o ofile] ifile
```

-pthresh *n* Sets the *pthresh* parameter to n. See the section on time/space tradeoffs

-time Optimize for time. It's the same as **-pthresh 0.**

-space Optimize for space. It's the same as **-pthresh 1.**

-lnc Line numbers as comments: each line generated by *ace* will be prefixed with a comment that tells what line of the input file it came from. This is a good switch to use for debugging since dbx will step through the expanded output, but you'll be able to find the code in the original source.

-nln No line numbers. This should be used if you want to read the code generated by *ace*. It removes the clutter left by line numbers.

-mingain *m* Sets the *mingain* parameter to *m*. See the section on time/space tradeoffs.

-Qpath *path* Causes *ace* to look in *path* for *cpp*. Normally it just looks in /lib and /usr/lib.

-o *ofile* Sends the generated output to *ofile*. The default is standard out. *Ofile* will be unlinked before it is created, and it will be created with mode 444, to prevent accidental editing.

-ifc Includes comments after each **if** and **else** that indicate what's true and what's false, according to containing if statements.

Ace pipes its input through *cpp* and takes all parameters that *cpp* would accept and passes them on to it.

Appendix 2. A Large Example

As an example of how *ace* can be used in a real-world example, here is a routine for drawing vectors using Bresenham's algorithm. It is almost exactly the same as the vector routine that appears in the Shapes library, except that the code to support clipping has been eliminated. But is does handle several framebuffer depths (1, 8 and 32 bits per pixel), plane masks, and all 16 rasterop codes. Normally the inner loop is written out many times for the various special cases. Here, it is written once, and *ace* is used to generate all of the special cases:

```
sh_fb_VecPt(ras, X1, Y1, X2, Y2)
    RASTER      ras;
{
    register short count;
    register int err;
    register int erra;
    register int errb;
    int         plane_enable = FB_plane_enable,
                dx, dy, left, lineiny;

    if (Y1 > Y2) {
         swap_coord(X1, X2, left);
         swap_coord(Y1, Y2, left);
    }
    dy = Y2 - Y1;
    dx = X2 - X1;
    if (left = (dx < 0))
         dx = -dx;
    if (lineiny = (dy > dx)) {
         count = dy + 1; erra = dx << 1; errb = dy << 1;
         err = left ? 1 - dy : -dy;
    } else {
         count = dx + 1; erra = dy << 1; errb = dx << 1;
         err = -dx;
    }
    $switchout(DEPTH, ras->RAS_DEPTH, SH_SUPPORTED_DEPTHS) {
         register int bpsl = ras->RAS_LINEBYTES;
         PIXCOLOR(color, FB_col, DEPTH);
         PIXROP(ropcode, (int) FB_rop, color, DEPTH);
         PIXPTR(pix, DEPTH);
         PIXMASK(mask, DEPTH);
         initpixelpointer_no(ras, pix, mask, X1, Y1, DEPTH);
         --count;          /* $repeat(count) generates count+1 loops */
         $fastrops(ropcode, DEPTH)
             $alwayspulloutiff((FB_disp & FB_DRAW_PLANES) == 0,
                                 plane_enable == ~0)
             $alwayspullout(erra != 0)
             $alwayspullout(lineiny == 0)
             $alwayspullout(left == 0)
             $repeat(count) {
             writepixel(pix, mask, DEPTH,
                         ropcode, color, plane_enable);
             if (lineiny == 0)
                  if (left == 0)
                      RIGHTSTEP(pix, mask, DEPTH);
                  else
```

```
                       LEFTSTEP(pix, mask, DEPTH);
              else
                   DOWNSTEP_STRIDE(bpsl, pix, DEPTH);
              if (erra != 0)
                   if ((err += erra) >= 0) {
                       err -= errb;
                       if (lineiny != 0)
                            if (left == 0)
                                RIGHTSTEP(pix, mask, DEPTH);
                            else
                                LEFTSTEP(pix, mask, DEPTH);
                       else
                            DOWNSTEP_STRIDE(bpsl, pix, DEPTH);
                   }
         }
    }
}
```

Running this through *ace* yields a 20 page source file that contains expanded code for all the special cases. As you can see, the inner loops are all very tight. The following listing has been substantially abbreviated. The expansion of the `$repeat` macro is especially interesting: it is machine dependent. In this case it expands to `do { ... } while (--count != -1)`, which is compiled on a 68020 into a dbra instruction.

```
int sh_fb_Vect(ras, X1, Y1, X2, Y2)
  RASTER ras; {
  register short count;
  register int err;
  register int erra;
  register int errb;
  int plane_enable = sh_fb_attrs.plane_enable, dx, dy, left, lineiny;
  if (Y1 > Y2) {
      left = X1;
      X1 = X2;
      X2 = left;
      left = Y1;
      Y1 = Y2;
      Y2 = left; }
  dy = Y2 - Y1;
  dx = X2 - X1;
  if (left = dx < 0)
    dx =  - dx;
  if (lineiny = dy > dx) {
      count = dy + 1;
      erra = dx << 1;
      errb = dy << 1;
      err = left ? 1 - dy :  - dy; }
 else {
      count = dx + 1;
      erra = dy << 1;
      errb = dx << 1;
      err =  - dx; }
 switch (ras->RAS_DEPTH) {
 case 1: {
     register int bpsl = ras->RAS_LINEBYTES;
     int color = sh_fb_attrs.col;
     int ropcode = color ? mono_remap1[(int) sh_fb_attrs.rop] :
                          mono_remap0[(int) sh_fb_attrs.rop];
     register unsigned short *pix;
     register unsigned short mask;
     pix = (unsigned short * ) (ras->RAS_DATA + (short) ras->RAS_LINEBYTES*
       (short) Y1 + (X1 >> 3 & -2));
     mask = 32768 >> (X1 & 15);
     --count;
```

```
switch (ropcode) {
case 14:
  if (erra != 0) {
    if (lineiny == 0) {
      if (left == 0)
```

Monochrome (1 bit deep), going right, x is the major axis, the line is neither horizontal nor vertical, and the ropcode is SRC.

```
        do {
            *pix |= mask;
            if ((mask >>= 1) == 0) {
                mask = 32768;
                pix++; }
            if ((err += erra) >= 0) {
                err -= errb;
                pix = (unsigned short * ) ((int) pix + bpsl); } }
        while (--count != -1);
      else
```

Monochrome (1 bit deep), going left, x is the major axis, the line is neither horizontal nor vertical, and the ropcode is SRC.

```
        do {
            *pix |= mask;
            if ((mask = (unsigned short) (mask << 1)) == 0) {
                mask = 1;
                pix--; }
            if ((err += erra) >= 0) {
                err -= errb;
                pix = (unsigned short * ) ((int) pix + bpsl); } }
        while (--count != -1);
    else
```

Monochrome (1 bit deep), going right, y is the major axis, the line is neither horizontal nor vertical, and the ropcode is SRC.

```
      if (left == 0)
        do {
            *pix |= mask;
            pix = (unsigned short * ) ((int) pix + bpsl);
            if ((err += erra) >= 0) {
                err -= errb;
                if ((mask >>= 1) == 0) {
                    mask = 32768;
                    pix++; } } }
        while (--count != -1);
      else
        do {
            *pix |= mask;
            pix = (unsigned short * ) ((int) pix + bpsl);
            if ((err += erra) >= 0) {
                err -= errb;
                if ((mask = (unsigned short) (mask << 1)) == 0) {
                    mask = 1;
                    pix--; } } }
        while (--count != -1);
  else
    if (lineiny == 0) {
      if (left == 0)
        do {
            *pix |= mask;
            if ((mask >>= 1) == 0) {
                mask = 32768;
                pix++; } }
        while (--count != -1);
      else
        do {
            *pix |= mask;
```

```
                  if ((mask = (unsigned short) (mask << 1)) == 0) {
                      mask = 1;
                      pix--; } }
              while (--count != -1);
          else
            do {
                *pix |= mask;
                pix = (unsigned short * ) ((int) pix + bpsl); }
            while (--count != -1);
        break;
.
.
.
  case 8: {
      register int bpsl = ras->RAS_LINEBYTES;
      register int color = sh_fb_attrs.col;
      int ropcode = (int) sh_fb_attrs.rop;
      register unsigned char *pix;
      pix = ras->RAS_DATA + (short) ras->RAS_LINEBYTES*(short) Y1 + X1;
      --count;
      switch (ropcode) {
      case 12:
        if ((sh_fb_attrs.disp & 1) == 0) {
          if (erra != 0) {
            if (lineiny == 0) {
              if (left == 0)
```

8 bit deep pixels, going right, x is the major axis, the line is neither horizontal nor vertical, all planes enabled, and the ropcode is SRC.

```
                do {
                    *pix = color;
                    ++pix;
                    if ((err += erra) >= 0) {
                        err -= errb;
                        pix += bpsl; } }
                while (--count != -1);
              else
```

8 bit deep pixels, going left, x is the major axis, the line is neither horizontal nor vertical, all planes enabled, and the ropcode is SRC.

```
                do {
                    *pix = color;
                    --pix;
                    if ((err += erra) >= 0) {
                        err -= errb;
                        pix += bpsl; } }
                while (--count != -1);
.
.
.         else
           if (lineiny == 0) {
             if (left == 0)
```

8 bit deep pixels, going right, x is the major axis, the line is horizontal, not all planes are enabled, and the ropcode is SRC.

```
               do {
                   *pix = color & plane_enable | *pix & ~plane_enable;
                   ++pix; }
               while (--count != -1);
             else
               do {
                   *pix = color & plane_enable | *pix & ~plane_enable;
                   --pix; }
               while (--count != -1);
           else
             do {
                  *pix = color & plane_enable | *pix & ~plane_enable;
                  pix += bpsl; }
              while (--count != -1);
        break;
.
.
.
  case 32: {
      register int bpsl = ras->RAS_LINEBYTES;
      register int color = sh_fb_attrs.col;
      int ropcode = (int) sh_fb_attrs.rop;
      register unsigned char *pix;
      pix = ras->RAS_DATA + (short) ras->RAS_LINEBYTES*(short) Y1 + X1*
        4;
      --count;
      switch (ropcode) {
      case 12:
        if ((sh_fb_attrs.disp & 1) == 0) {
          if (erra != 0) {
            if (lineiny == 0) {
              if (left == 0)
```

32 bit deep pixels, going right, x is the major axis, the line is neither horizontal nor vertical, all planes enabled, and the ropcode is SRC.

```
                do {
                    *(int * ) pix = color;
                    pix += 4;
                    if ((err += erra) >= 0) {
                        err -= errb;
                        pix += bpsl; } }
                while (--count != -1);
.
.
.
       else
           if (lineiny == 0) {
             if (left == 0)
```

32 bit deep pixels, going right, x is the major axis, the line is horizontal, all planes enabled, and the ropcode is SRC.

```
               do {
                   *(int * ) pix = color;
                   pix += 4; }
               while (--count != -1);
             else
```

32 bit deep pixels, going left, x is the major axis, the line is horizontal, all planes enabled, and the ropcode is SRC.

```
               do {
                   *(int * ) pix = color;
                   pix -= 4; }
               while (--count != -1);
           else
```

32 bit deep pixels, going neither left nor right, y is the major axis, the line is vertical, all planes enabled, and the ropcode is SRC.

```
             do {
                 *(int * ) pix = color;
                 pix += bpsl; }
             while (--count != -1);
```

ANSI-C: So What?

Bruce Ellis
C-Side Software Services Pty. Ltd.
Box 207, Bondi Beach, NSW 2026
Australia

ABSTRACT

After years of often heated deliberation the X3J11 Technical Committee on the C Programming Language has spoken. Their goal was to "provide an unambiguous and machine-independent definition of the language C."[1] Such a document was needed because the de facto C standard, The C Programming Language by Brian W. Kernighan and Dennis M. Ritchie [K&R] was not precise, complete or prophetic enough to describe what we have come to know as the C programming language, a language that has evolved and mutated over the years.

Ever since the first enhancement was added to a C compiler the language was more or less defined by the compilers. After all, there weren't many compilers and it all took place at Murray Hill in any case. A language issue could be resolved by electronic mail.

This had its problems. A large number of C compilers even to this day are based on Steve Johnson's PCC [PCC], with its bugs, deficiencies and inconsistencies. (It is by no means the only offender.) This gives rise to the "Oh, your compiler has THAT bug" syndrome, characterised by the query "Why can't I have a pointer to a void function?" The answer is "You can but the compiler coughs on it." Porting a program degenerates to appeasing the compiler rather than fixing incorrect code.

This paper presents an overview of the ANSI–C standard and its uses and abusers.

Overview

A good place to start examining the ANSI–C project is with its definition of itself, Section 1.2 – Scope.[2] From this we discover that the standard specifies the representation of C programs, their syntax and constraints, semantics of execution, and the representation of input/output data. It does not specify how C programs are compiled and run, conversion between C I/O data with other data. Neither does it specify various implementation limitations, e.g. limitations on program complexity. Furthermore the standard specifies translation and execution environments and a standard (compulsory) library.

The *base* documents are [K&R] and the 1984 /usr/group standard [UGS]. The first forms the basis for the description of the language proper – the second that of the library. References are made to various other standards including the IEEE Floating Point Standard [IFPS].

A C language standard would not be of much use if it described a language significantly incompatible with accepted C practice. Well written C code can be ANSI–fied painlessly. Unportable and morally corrupt code may prove to be of little use in an ANSI world. Many constructs that raise warning diagnostics or invisible grimaces from traditional compilers would now trigger error diagnostics.

[1] Sparc document number 83-079.

[2] All references are to the Oct 31, 88 draft [ACD].

Curiously, some constructs are specified to produce undefined behaviour – no diagnosis is required. Some of these cases are understandable (it is difficult to detect where code is taking advantage of integer overflow properties). Others (e.g. undefined escape codes in strings) are easily detectable and should be diagnosed. The "I don't know what to do with this so I'll do what I like" attitude has no place in what is meant to be a rigorous technical standard.

Safe C

A programming language standard must be pedantic. On the other hand a programmer gains little from an obsession with language pedantics. He is not out to bring a warm glow to the standards committee, rather to just get the job done, and done well. This point escapes many who write about programming rather than write programs.

C is wide open to abuse. In fact it has a somewhat deserved reputation as an abusable and abused language. Consider these unattributed observations:

"After all, C does give you that feeling of freedom. Programming unconstrained by the limits of reality, understanding, or functionality."

"C gives you all the power of assembly language, with all the convenience of assembly language."

However, the programmer is not compelled to drive unsafely any more than a skateboard rider is, though there is something to be said for the thrill derived from such a practice.

It follows that C programmers often utilise only their favourite subset of C, just as a board rider chooses his tack. Reasons for this include:

1) Preference. Some programmers prefer `#define`'s, some `enum`erators. Many people distrust bit fields. Some riders prefer car parks to shopping malls, many distrust Volvos.

2) Portability. Compilers often choke on some apparently legal constructs (like pointers to void functions). Avoiding such constructs and commonly accepted apparently illegal constructs can avoid portability tears.

3) Restricting a program to a small, well chosen, but consistent, subset of the language helps make programs simpler and more easily maintainable.

Luckily the subset of C that I choose is contained in ANSI–C (well almost). Others may not be so lucky.

Some Changes From [K&R]

So what is different and what do I have to learn and unlearn?

A few keywords have been added, the most annoying being `const`. `const` is such a good variable name that there will be fighting between old programs and ANSI compilers for quite a while over this one.

`const` (constant) is a *declaration qualifier* specifying that a variable's value will not be modified. Its primary role seems to be in the creation of extremely ugly declarations and the introduction of confusing conversion and conformability rules. It seems that the standards committee never did work out what they wanted `const` to do.

The `signed` keyword has been introduced to enable the declaration of a `signed char`, thwarting the compiler's whim to slip you an `unsigned char` while you're not looking. You can also say `signed` or `signed int` which both mean `int`. Alas `long float`, which used to mean `double` but which no-one ever used, is gone. `long double` now means `double` or maybe bigger. `long long` would have been more useful, now that disk sizes can't be held in a `long`.

Declarations can also be qualified with `volatile` (e.g. `volatile int petrol;`) which may or may not do anything. `noalias` was meant to do the same thing. i.e something or nothing but was dropped as things were getting a bit out of hand. Just use volatile or signed or both.

Now for the good news.

- Hex constants in strings and character constants e.g. '\xFF'. Furthermore 8 and 9 are no longer considered octal digits (both in octal escapes and integral constants). Curiously, a hex escape sequence can have as many hex digits as you like but octal escape sequences are still limited to three octal digits. Hence "\xdeadbeef" is a string of length one (plus the terminating zero valued character) but don't ask what it's value is. "\1234", on the other hand, is of length two and is equivalent to "S4" in the ASCII world.
- void * means generic pointer replacing traditional white lies (char *, caddr_t, unsigned long). Library functions that deal with anonymous slabs of memory such as malloc return such things.
- Old-fashioned assignment operators are truly gone, i.e. no =+. Also += is now one token instead of two.
- Unary +. Handy for programs which generate code, though the interaction with the assignment operator change will really annoy someone, someday.
- sizeof is of (implementation dependent) type size_t. size_t is an unsigned integral type declared in <stddef.h>.

Accepted Practice Legitimized and/or Defined.

These come as no suprise.

- enumerated types.
- void type.
- structure assignment and return.
- A precise definition of different name spaces (variables, aggregate tags, aggregate members, labels etc.). Also, formal parameters are in the same scope as their function's body.
- switch on any integral type.
- Initialization of unions. Many compilers have some mechanism for this. ANSI–C choose to initialize the first element. Sigh.
- Function prototypes such as:

```
extern double   fmod(double x, double y);
```

are now part of the language though their use is not mandatory. A prototype also causes arithmetic conversions between conformable types. You either like them or you don't. Also functions with an unspecified number of arguments come into their own. e.g.

```
extern int   printf(const char *, ...);
```

along with a clarification of the accepted varargs practise.

Preprocessor

The ANSI–C standard radically redefines the C Preprocessor, though one could argue that it was never well defined. A C program is now composed of preprocessing tokens. This has grave consequences for the implementer. Eight distinct translation phases are defined. Compiler performance and rational program surface structure are sacrificed in an attempt to codify the quirks of existing implementations.

There are new operators for constructing preprocessing tokens. Previously an empty comment was used to paste together tokens. e.g.

```
#define v(n) gensym/**/n

v(27)
```

would expand to gensym27, which would then be subject to further macro expansion. It never was clear whether this was an intentional feature. To achieve the same in ANSI–C you would use:

```
#define v(n) gensym ## n
```

The ## operator concatenates two adjacent tokens. The result must be a valid preprocessing token. More interesting (perhaps more bizarre) is the # operator which constructs a string from an actual parameter. e.g.

```
#define str(s)   # s

str(x > y)
```

yields "x > y". This was previously achieved with

```
#define str(s)   "s"
```

as strings within replacement text were scanned for parameter replacement. In some versions of the preprocessor interaction with escapes gave humorous results.

```
#define ha(n)    printf("%d\n", n)

ha(f);
```

yielded

```
printf("%d\f", f);
```

perhaps not what was intended. Furthermore if an actual parameter contains string tokens they are modified so that the result is a valid string token (escapes are introduced for double quote and backslash characters).

The semantics of these new operators are strange enough to warrant an almost incomprehensible description in the standard plus some very confusing examples. They are not a strong point of ANSI–C.

To add to this confusion the scanning phase also contains a provision for representing the C character set in non-ASCII character sets using *trigraphs*. e.g. ??< means {. Here is a familiar example.

```
??=include <stdio.h>

main()
??<
    printf("hello world??/n");
??>
```

Once again a feature of questionable merit.

Also, *backslash newline* is deleted everywhere (previously only in strings) and adjacent string constants are concatenated. This is fortunate if you wish to put ?? in a string without running foul of trigraphs. And there's more.

Library

More than half of the 200+ page reference manual is concerned with the standard header files and libraries. It has been said that this section is an unmitigated disaster thrown in to mask the absurdities of the preprocessor definition. Perhaps this is a bit harsh.

ANSI–C requires a large number of standard header files. These may be included in any order and included more than once. The last requirement is of dubious value as on second and subsequent inclusions the headers need to be reparsed on the off-chance that it matters.

It is left as an exercise to the reader to explore the library definition more. You will find many of your favourite section 3 functions with subtle twists plus some true excesses (e.g. `strftime`).

Direction

What does this mean to the man in the street? Disregarding the preprocessor, the library description and the odd wart, the definition codifies and clarifies what we now have. It is telling, however, that compilers that are adopting some or all of the ANSI enhancements feel free to choose semantics the authors like to use or implement. The GNU C compiler is gaining popularity and reports to be ANSI–ish. It doesn't even get close (particularly the preprocessor). It does have a very large number of flags, including a *traditional* flag which reportedly turns it a bit towards accepted [K&R] practise. It is disturbing that it also contains a large number of extensions including more keyword abuse. Combinations of the keywords `inline`, `static`, `volatile` and `asm` indicate all manner of intentions. At least it is free.

Some people advocate *Ken-C*, a language much like [K&R] but with useful ANSI–like extensions – no standard libraries or headers, frugal preprocessing.

I'll stick to my favourite subset, about a quarter of ANSI–C (none of the standard library, half of the language) and the UNIX* C Library. But do I choose `strrchr` or `rindex`?

Conclusion

This paper is structured somewhat like the ANSI–C standard – the initial intention of a scholarly work fading into somewhat a disjoint tirade. It contains worthwhile sections and some almost incoherent examples and asides.

The standard provides some joy to the implementer but also provides him with an almost impossible task. I hope the reader of this paper fares better. Finally another unattributed quote that could have easily been said about ANSI–C:

"I have had to break the news to it that *System V* is a marketing campaign, not an implementation."

References

[ACD] *Draft Proposed American National Standard for Information Systems – Programming Language C.* October, 1988.

[IFPS] *IEEE Standard for Binary Floating–Point Arithmetic.* ANSI/IEEE Std 754-1985.

[K&R] Kernighan, B.W. and Ritchie, D.M. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 1978.

[PCC] Johnson, S.C. *A Tour Through The Portable C Compiler. UNIX Programmer's Manual. Seventh Edition, Volume 2B.* Bell Laboratories, Murray Hill, NJ, 1979.

[UGS] *1984 /usr/group/Standard.* /usr/group Standards Committee, Santa Clara, CA, 1984.

* UNIX is a trademark of AT&T.

No one ever got fired for delivering on time

Stephen Young
NEC Information Systems Australia
stevey@neccan.necisa.oz
Ph: (062) 514611

Introduction

Developing large software systems for profit presents challenges that are not faced by hobbyists, students or academics. Managers want to know:

- When will it be finished?
- How much will it cost?
- How can we raise productivity?
- How can we reduce the number of defects?
- Is the project on schedule?

Answering these questions requires a way of thinking about size, progress and effort.

NEC has a Software Development Centre in Australia. The team working on the UNIX ImageWare project has developed a conceptual framework and a corresponding tool to assist estimating:

- the amount of functionality in software
- the cost, in person/days, of building software;

and tracking:

- progress towards target functionality
- effort, in person/days, spent on progress.

The challenge

Successfully completed commercial software projects are hard to find. Success means correctly scoped functionality delivered on time with Quality. Projects fail for many reasons: political/managerial bungling, external dependencies not met, conflicts within the team and sometimes technical incompetence. Fred Brooks [1], in his famous Mythical Man-Month article, says that "more projects have gone awry for lack of calendar time than for all other reasons combined". An important challenge facing software project managers is how to reliably estimate delivery dates and how to convince people to wait that long for a good product.

A further challenge is how to accurately track progress towards completion, without confusing effort with progress and without swamping the developers in paperwork and meetings.

This paper discusses methods of estimating the size of a software system. In particular, it describes the use of function points to estimate the size of a UNIX software system for processing images. It illustrates the use of "function points completed" as an objective measure of project progress. The design and use of a software tool, ESTRACK, is described. ESTRACK automates many of the clerical steps required for estimation and tracking. Finally, the paper reports statistics on size (measured by function points and source lines of code) and development productivities for several languages used on the project.

As UNIX comes of age in the commercial DP world, we should try to build on the work of others by learning from their failures and picking the best tools from their successes. This applies equally well to project management as to programming. Despite initial scepticism by team members from UNIX backgrounds, our experience demonstrates that good management tools from the non-UNIX world can be successfully used on UNIX projects.

Estimating

Methods of estimating

Traditional methods of estimating include:

- use a manager's guess
- ask a technical expert
- predict source lines of code (SLOC), and a productivity figure
- buy an estimating package
- perform a critical path analysis
- use a mixture of the above.

Each of these methods has its advantages and disadvantages. A good estimate should not only be an accurate prediction, but also should attract commitment from the developers and managers involved.

A manager's guess takes little effort to obtain and can be expected to have a high level of management commitment. But managers' guesses can be influenced more by external constraints than the size of the task. Too easily the initial guess becomes an impossible deadline for developers.

A technical expert's estimate also takes little effort to obtain, and has the advantage that at least one other person in addition to the manager has been consulted. However technical expertise is not the same as estimating expertise. The expert is likely to estimate based on their experience level, is likely to only estimate coding effort and is likely to ignore the effects of non-task effort such as interruptions, sick leave, and working on other things.

Predicting SLOC has the advantage that productivity figures are available for various programming languages, introducing a level of objectivity. Developers can be committed to the estimate, though there is incentive to produce large quantities of SLOC rather than simple elegant solutions with appropriate tools. Managers can understand the basis for the estimate. The main weakness is that SLOC is often hard to predict early in a project, and has a number of shortcomings as a metric. These shortcomings are discussed below.

Buying an estimating software package can appear an attractive option. The software will enforce a degree of standardisation among projects. Estimates appear to be derived in a more objective manner. Commitment is high because of the perception that information from a computer system is trustworthy. However, estimating software packages are only a tool based on a generalised model. They cannot answer the difficult questions of estimating, and their results are often contrived to fit a manual estimate. anyway.

Critical path analysis is not really an estimating method. It presupposes that tasks have been identified and estimated. It is a helpful tool for translating estimates into elapsed time schedules.

Factors in estimating

In estimating it it important to differentiate between the impact of size, productivity and elapsed time issues.

Size is affected by the amount of functionality. The amount of functionality varies with users' requirements. The more data held in a system, the more interfaces with other systems, the more user transactions, the more functionality to be delivered, and the more work involved.

Productivity is affected by the experience of staff, programming tools and the development environment. Experienced developers working with good tools in an environment they enjoy will deliver the same amount of functionality in less time than inexperienced staff with poor tools in a bad environment.

Elapsed time is affected by the size of the team, the proportion of a day actually spent on the project and external dependencies. The same team developing the same amount of functionality but with other commitments, and dependent on outside support, will deliver later than those who have no outside interruptions and no external dependencies.

This paper focuses on the measurement of size.

Inadequacy of SLOC as a metric for size

The most common metric for size is source lines of code (SLOC). This has a number of disadvantages, including lack of uniformity from one language to another, widely varying standards for counting, being hard to estimate early in a project and not being directly proportional with added value.

A line of source code is not uniform from one language to another. A few lines of SQL source code in a 4GL, may be equivalent to hundreds of lines of C. How may lines of C would it take to implement the following SQL ?:

```
select faculty_name, ave (distinct salary)
from faculty_table F, person_table P
where P.faculty_number = F.faculty_number
group by faculty_name
order by faculty_name;
```

The number of SLOC counted can vary by a factor of 2 depending on:

- whether logical or physical statements are counted
- whether comments are counted
- whether blank lines are counted
- whether include files are recounted with every use
- whether makefiles and shell script tools are counted.

The process of measuring affects the thing measured. Whatever you measure you tend to get more of. Measuring size with SLOC subtly encourages more lines of code, which may not mean more functionality delivered. Quite often an elegant programming solution uses less code than a cumbersome solution. Productivity can be *inversely* related to SLOC.

The concept of *function points* is proposed as a better alternative to SLOC.

Function points

Origin of function point concept

The concept of function points was invented by A. J. Albrecht of IBM in 1979 [2]. Function points are essentially a weighted sum of the number of business functions provided to a user. Albrecht's weightings are based on the number of external inputs and outputs, the number of logical files, the number of external interfaces and inquiries. The items are categorised into three levels of complexity.

References [3], [4] and [5] contain additional material on the original function point ideas.

Adaption of function points to local needs

On the ImageWare project we have adapted Albrecht's function point counting method for our environment.

The main changes we have made are:

- Count each module (first level partitioning of the product) as one system. This provides more accurate tracking of progress per module but results in some double counting when we aggregate function points per module for the whole of the product.
- Replace the *External input* category with *Service provided* because we mainly deal with software that provides new library calls, involving both input and output.
- Replace the *External output* category with *Internal interface* because we feel that interfaces to other ImageWare modules are an important factor in measuring functionality.
- Increase the relative weightings for Services provided because they are the primary reflection of functionality.
- Increase the relative weightings for External interfaces because for ImageWare they provide much functionality, e.g. direct access to an optical disk.
- Ignore the processing complexity adjustment because most of Albrecht's factors are either constant or irrelevant for us.

Details of our guidelines for counting function points are presented as an Appendix.

These changes mean that our function points cannot be compared to anyone else's.

Measuring documentation size with function points

Documentation is as important as the software for our product. All the project management issues for software development apply to documentation as well. We decided to use function points to measure documentation as well.

Our guidelines for documentation were designed so that each page of a manual delivered to a customer counts on average as one function point. For our system this results in documentation functions points being approximately the same as the corresponding software. The Appendix details our guidelines for counting documentation function points.

Tracking

It is important to reconcile estimates against what actually happened. In particular, the ratio of function points to person/days is a measure of productivity. This provides feedback into the estimating process. There are two separate parts to tracking - effort and progress. Effort is not the same as progress. A week may be spent finding a solution to a problem without making any progress. Progress can be negative. A code inspection may reveal a design flaw requiring extensive re-work.

Tracking effort

Traditional methods of tracking effort include:

- not tracking it at all
- asking "what did you do last week?"
- filling in time sheets.

A major reason for inaccurate estimates is that only effort spent directly on the task is considered. It is important to track effort spent on support activities (such as standards, development environment tools), and non-task activities (such as work on other projects). As a minimum, this provides data to explain "Whatever have you been doing for the last 2 months?".

Tracking progress

Traditional methods of tracking progress include:

- periodic requests for verbal and written reports from team members
- lists of tasks with expected completion dates, reconciled against actual dates
- use of project management software to keep lists of tasks and dependencies.

The concept of function points provides a new method of tracking. Each deliverable (e.g. a function in a C library) in ImageWare has an associated number of function points. A deliverable is tracked through the phases of the project by using a percentage complete figure. Total progress is the sum of function points times percent complete for each deliverable. To avoid the "90 per cent complete" syndrome, guidelines are used to specify maximum percentage complete values within each phase. For example, the completion of coding of a deliverable permits it to be marked as 25 percent complete. Due to the guidelines, individual deliverables tend to experience quantised changes in percentage complete values. When aggregated over a number of deliverables, these step changes are barely perceptible.

Progress over time can be graphed easily. Figure 1 is an example graph including both software and documentation.

The ESTRACK software tool

The ImageWare team used INFORMIX relational database software to build a tool called ESTRACK to automate:

- the calculation of function points
- estimating person days
- tracking progress and effort.

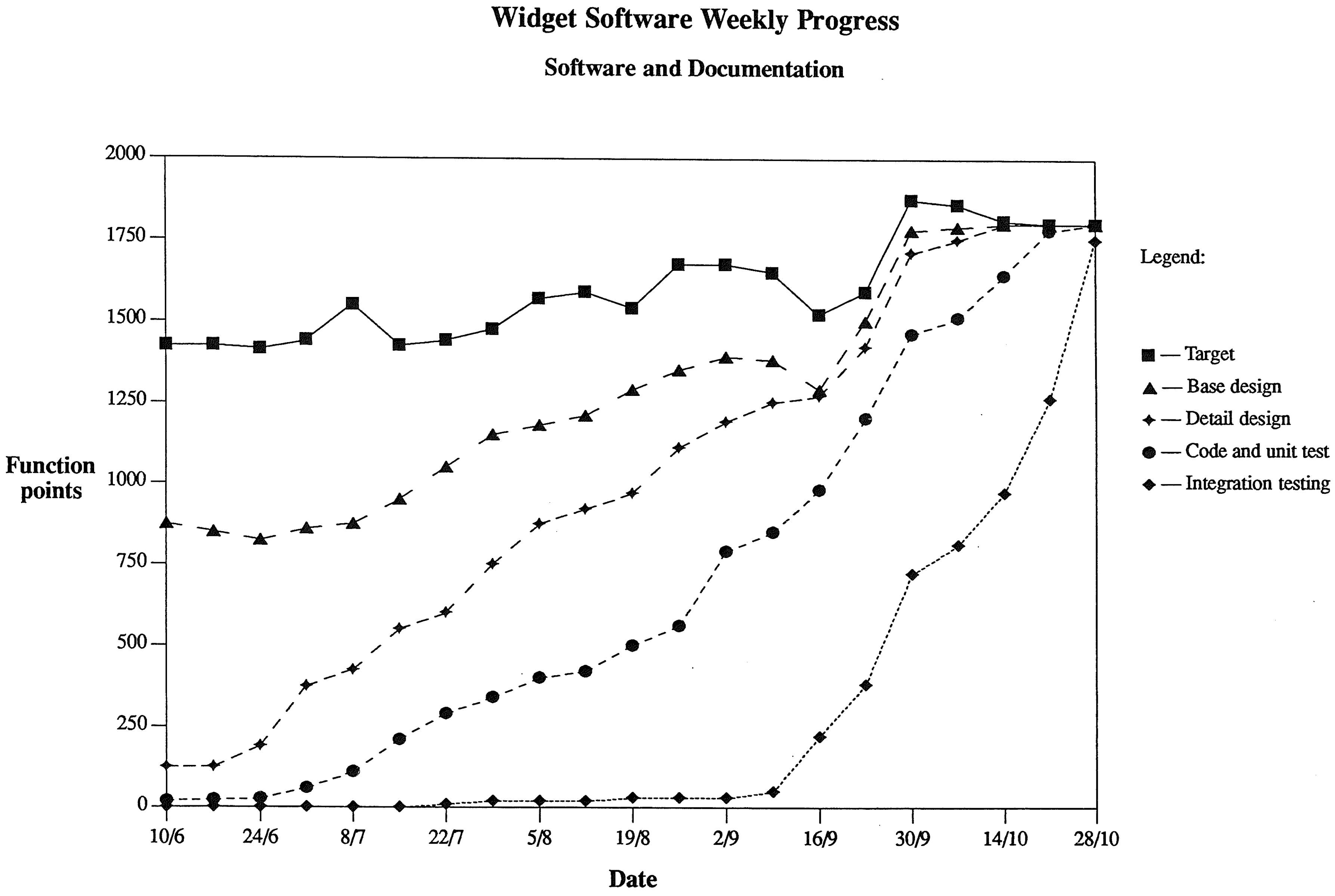


Figure 1: Graph of weekly progress by functions points complete.

The ESTRACK data model

Figure 2 shows the entities that ESTRACK models and their relationship to each other.

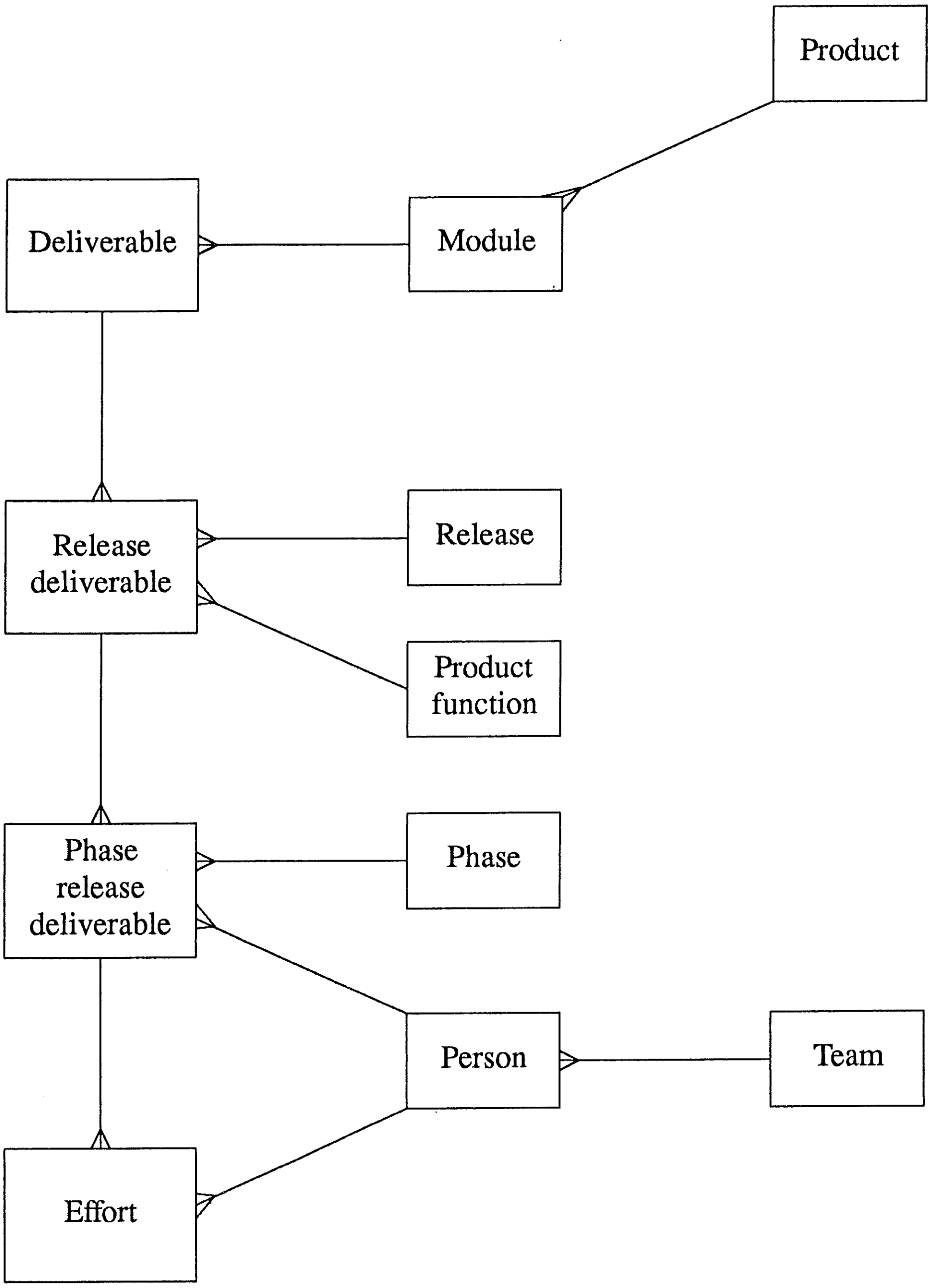


Figure 2: ESTRACK Data Model

The entities on the data model are:

Products are NEC software products.

Modules are layers in the product architecture.

Releases are target amounts of functionality delivered at one time.

Product functions are optional features that can be costed for marketing judgements about their value.

Phases are system development phases such as requirements definition, detailed design, code and inspect.

Persons are people on the project team.

Teams are groups of people with a common purpose.

Deliverables are any software or documentation or support function that the team delivers in any release.

Release deliverables are those deliverables which form a release. A particular deliverable may appear in a number of releases. Function points are calculated at this level.

Phase releases deliverables reflect work done in each phase for a release deliverable. Estimates are calculated at this level using productivity per function point or an absolute amount of person-days.

Effort is actual person-days spent each week on phase release deliverables.

Using ESTRACK

Figure 3 illustrates the data flows in ESTRACK.

Team estimators enter deliverables and function point arguments into ESTRACK. An SQL view calculates function points for each release deliverable.

For each module actual productivity in terms of days per function point is examined from previous releases. Good reasons have to be raised for estimating with a different productivity figure from the actual figure achieved last time.

Reports from ESTRACK show estimated person-days per module and product function. Based on the estimated person-days per product function, decisions are made about which product functions will be in the release.

Once development begins, progress in terms of percentage complete is maintained against each phase release deliverable. Each week ESTRACK calculates function points complete for each phase and stores the values in a history table. The values in the history table are extracted and passed to *grap* for formatting. Figure 1 above is an example of a typical graph.

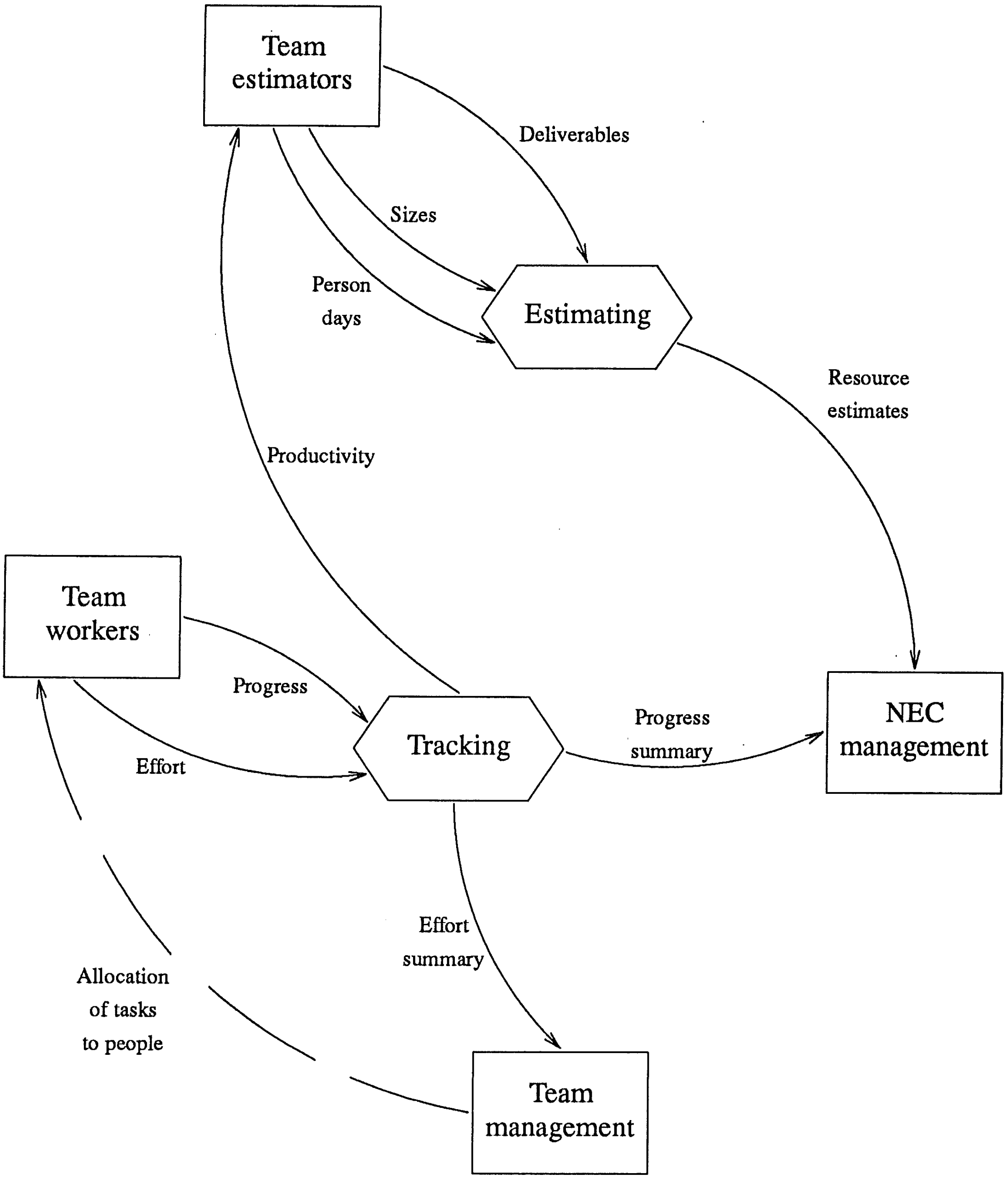


Figure 3: ESTRACK Dataflows

Implementation of ESTRACK

The first version of ESTRACK was implemented using the INFORMIX SQL relational database software. The screen generator was used for data entry, and the report generator for output. The "user menus" of the SQL package were used to provide a menu based interface to the data entry screens and reports.

Recent additions to ESTRACK have been implemented using INFORMIX 4GL. The 4GL language has allowed the addition of functions that could not be built using the basic SQL system. For example reporting criteria can now be specified using the "query by example" technique.

Statistics on size and productivity

The first release of ImageWare was completed in November 1988, by a team of up to 8 people working from January 1988. Some statistics on the development process are shown in Table 1.

Deliverable	Size		(SLOC/fp)	Effort	Productivity
	(fp)	(SLOC)		(person days)	(person days/fp)
4GL software	214	3132	14.6	87	0.41
ESQL/C software	189	2190	11.6	63	0.33
C software	869	12260	14.1	416	0.48
Documentation	1247	21322	-	145	0.12
Support activities	-	-	-	990	-

Table 1: Development statistics for ImageWare Release 1.

The values for SLOC reported are for non-comment physical source lines. Include files were counted once. Values for documentation are 'non-comment troff source lines'. The *software* and *documentation* effort covers requirements definition, detailed design, code and inspect, integration test, and developing installation procedures. The effort expended on *support activities* includes 1 person full time as project manager. It also includes a significant 'start-up' effort in establishing our hardware and software environment.

There are two noteworthy features of these statistics: the high proportion of time on support activities and the lack of correlation between language and productivity.

Support activities

Support activities represented 58% of total effort. This fits with conventional wisdom that great amounts of time just "disappear" in overheads. Table 2 shows the major categories of support effort.

Category	person days
Project management	244
New staff	168
Set up development environment	160
Standards	139
Absent	95
Other	184

Table 2: Effort for major support activities for ImageWare Release 1.

Project management does not include effort by NEC upper management. *New staff* includes time spent recruiting as well as orientation time. *Absent* includes all leave, public holidays and time on external courses.

Language and productivity

SLOC and productivity were not related to programming language. Various reasons can be suggested for the absence of productivity gains from the 4GL over C. The function point counting method does not reflect some functionality of the 4GL which comes as part of the language but would take effort to reproduce in C (e.g. Query by example). The programming language can only affect the coding phase which represented 24% of Release 1 effort. It may be that once the appropriate language has been chosen, productivity is much more related to non-technical factors such as the ethos of the team, experience of staff and the physical work environment.

Lessons

Function point counting

The function point calculation method needs more standardisation in terms of the categories it uses and the weighting values. Albrecht's original categories and adjustment for processing complexity reflects batch oriented accounting and administrative systems. More general concepts are needed for operating system software, graphics interfaces and real-time computing.

The ImageWare project may be the first to apply function points to documentation. This idea needs to stand the test of experience. So far it has been very convenient estimate and track documentation in the same framework as software.

Function points are still dependent on method of implementation. The use of the *Internal interfaces* category presupposes implementation language to some extent.

Use of ESTRACK

Some valuable lessons have been gleaned from the experience of using function points and ESTRACK. Graphing progress works well for upper management and the team. Tracking functionality instead of SLOC encourages programmers to focus on real added value. Productivity figures provide valuable feedback for the next round of estimating. Those with management responsibilities can access the data without interrupting other people. Counting function points encourages a degree of discipline in defining requirements.

Effort tracking lessons

- Effort categories for overheads need more refinement.
- Some categories are too loose and are hard to interpret. For example, the category for *Project management* catches a lot of time from a few people, and includes a mixture of things such as staff reviews, estimating, liaison with NEC head office, progress tracking and task allocations. The effect of this is that we cannot easily identify any of these items individually.

Estimating lessons

- Having a history of actual productivities is very helpful for estimating.
- Support activities need to be estimated as carefully as direct development.
- Function points can only assist estimating direct development effort.
- It is dangerous to assume large productivity gains from 4GLs.

Conclusion

The time invested in adapting function points for ImageWare and building ESTRACK has paid back with savings in team time, in justifying estimates to upper management, in providing a graphical measure of progress (or lack of it) for the team and in credibility gained from delivering the first release on time.

References

[1] Frederick J Brooks, Jr. *The Mythical Man-month*, in Brill *Techniques of EDP Project Management: a book of readings*, Yourdon Press, New Jersey, 1984, p 181.

[2] A. J. Albrecht. *Measuring Application Development Productivity*, in Proceedings IBM Application Development symposium, Monterey, CA, 1979; GUIDE Int. and SHARE Inc, IBM Corp., p 83.

[3] A. J. Albrecht and J. E. Gaffney. *Software function, Source Lines of Code, and Development Effort Prediction*, IEEE Transactions on Software Engineering vol SE-9, No 6, November 1983, pp 639-647.

[4] J. Brian Dreger. *Function Point Analysis*, Prentice Hall, New Jersey, 1989, 174 pages.

[5] Capers Jones. *Programming Productivity*, McGraw-Hill Book company, New York, 1986, 280 pages.

Appendix

Count function points

We count each software module or manual as a system. A module is the first level partitioning of the product.

For each module we identify the deliverables, called services provided. Deliverables are whatever the user of that module can do with that module. E.g. functions in a library, shell scripts, chapter in a manual.

For each module we identify internal and external interfaces. Internal interfaces are those we have control over. External interfaces are those we do not have control over.

For each module we identify logical tables (e.g. files).

For modules providing interactive functions we identify inquiries and reports.

Each of these items is categorised as simple, average or complex. The information is entered into an INFORMIX database, and function points calculated using the multipliers shown in Table 3.

Service type	Simple	Average	Complex
Services provided	x 5	x 7	x 9
Internal interfaces	x 4	x 5	x 7
Logical tables	x 5	x 7	x 10
External interfaces	x 7	x 10	x 15
Inquiry/Report types	x 3	x 4	x 6

Table 3: Multipliers for function point calculation.

Data item units are the basis for complexity categorisation

Where a base type is passed/returned, it is a data item unit.

Where a structure is passed/returned, and the service does not look inside the structure, the structure counts as one data item unit.

Where a structure is passed/returned, and the service looks inside the structure, count one data item unit for each data item unit referenced.

An environment variable counts as 1 data item unit.

Examples of services provided

NB. Inquiries and reports are counted separately (see below).

Examples of a single service provided:

- one library function
- one UNIX tool (e.g. a shell script)
- one X Windows widget (where other widgets are added to a parent widget, the children should be counted as well as the parent.)
- one chapter in a manual
- one INFORMIX query/add/update/delete form.

Simple: less than 10 data item units passed & returned, none optional, or one optional argument.

6 pages of a manual

X widget types: button, core, box, form, label.

Average: neither simple nor complex

one diagram in a manual

X widget types: scrollbar, viewport, vpaned, dialog.

Complex: more than 20 data item units passed & returned, or more than 2 optional, or lists passed,

or complex algorithms involved, where an algorithm is considered complex if a significant portion of the development effort will be devoted to solving the logical process involved,

or input that must be parsed,

or X widget types: text.

Examples of internal interfaces

Interface means a call to something outside the module. Internal means inside ImageWare.

Examples of a single internal interface:

- use of one library function by one or mores release deliverables in a module (e.g. ICreateBuf)
- A pipe to another process is an internal interface.

Simple: less than 10 data item units passed & returned, none optional.

Average: neither simple nor complex.

Complex: more than 20 data item units passed & returned,
or more than 2 optional,
or lists passed.

Examples of external interfaces

External means outside the control of ImageWare development project.

NB. Interfaces to facilities in a programming language are not counted. (e.g. query by form in I-4GL) or interface to Informix from I-4GL

NB. Interfaces to UNIX in Programmers Reference manual are not counted. They are considered part of the C language.

Examples of a single external interface:

- use of one library function in X Windows toolkit
- use of one library function to access network
- a single operation on a Graphics Processor Board
- a single command to a printer/scanner
- a single command to store an image on an optical drive.
- use of INFORMIX-ESQL/C.

Simple: less than 10 data item units passed & returned, none optional
use of SQL SELECT statement.

Average: neither simple nor complex.
use of SQL UPDATE/DELETE/INSERT facilities counts as one average external interface.

Complex: more than 20 data item units passed & returned,
or more than 2 optional
or lists passed;
any use of SQL statements other than SELECT/INSERT/DELETE/UPDATE counts as one complex external interface.

Examples of logical tables

Logical means at an abstract level of functionality, independent of implementation.

A logical table may end up as an INFORMIX table, a UNIX file, etc.

Examples of a single logical table:

- each entity in a data model
- UNIX etc/group file
- a log file.

Simple: less than 10 data item units referenced on the table.
and table lookup only.

Average: neither simple nor complex.

Complex: more than 20 data item units passed & returned,
or more than 2 optional,
or table updated by this release deliverable.

Examples of inquiry/reports

Examples of a single inquiry/report:

- An INFORMIX query by example
- A single SQL view.

Simple: less than 10 data item units passed as selection criteria & shown to user,
no transformations,
no aggregations.

Average: neither simple nor complex.

Complex: more than 20 data item units passed as selection criteria & shown to user,
more than 5 transformations,
more than 5 aggregations.

Problems Facing Corporate Distributed Database Systems

Russell Burstow

ICL Australia Ltd (Brisbane)

Shaun Travers

Main Roads Department (Queensland)

Abstract

Standards and procedures are difficult to implement after the initial installation of a networked system. Organisations embarking on such a task should be aware of the issues, costs and decisions which may arise in the future.

This paper discusses the standards addressed, problems and decisions encountered during the implementation of a network encompassing mainframes, unix workstations, micro computers and remote distributed database servers at Main Roads Department (Qld) with emphasis on the particular issues surrounding the remote MIPS servers.

Introduction

Before the implementation of a Road Management Network for Main Roads Department Queensland, a number of considerations and decisions were necessary to provide a stable standard environment for the network to grow. These subjects could provide a guideline to assist potential network creators with a list of topics for review. The majority of these topics can be grouped into three areas of concern.

The products selected for this network included:-

MIPS M/120 (supplied by ICL)

Computer Protocol CS410 (supplied by Datacraft)

IBM PC clones (supplied by IBM, DataMini & Toshiba)

ORACLE (supplied by ORACLE)

NFS (supplied by ICL)

Existing equipment included:-

ICL Series39 (supplied by ICL)

SUN 3/50 & 3/60 (supplied by SUN)

The decisions and standards development commenced to bring these products from this spread of suppliers together in harmony, to create a network with hosts throughout Queensland.

Network Organisation Issues

Decisions were made to employ TCP/IP over Ethernet to create a LAN within each remote district office. Most district offices and branches would use a MIPS M/120 to serve most of their processing requirements; while access to this machine and the Network would be initiated from up to 100 PCs and SUN Workstations in each district. The wide area networking is to use the synchronous protocol CPNX via the CS410 communications processor.

With an estimated minimum of 30 subnetworks each encompassing up to 100 addressable hosts, CITEC (the Centre for Information TEchnology and Communications) was persuaded to enlarge their original allocation of 1024 host addresses to 65,536 host addresses from their two B class licences to our Network. This is partly attributable to the incapability of the CS410s to split the four byte Internet address on non-byte boundaries for subnet routing. (It was necessary to increase the more appropriate seven bit subnet mask to the entire last byte.) This also simplified the visual diagnosis of any addressing problems which arose. To further obviate address assignments, each type of host (PC, SUN, Terminal Server) is allocated a range within the subnet address. Sequential address allocation for all hosts, while economical in its use of addresses, will not function using the Internet concept of subnet routing.

After evaluation of several Ethernet analysis tools, a "Sniffer" board and software which is used in an IBM PC clone was purchased from Datacraft for LAN diagnosis and has proved invaluable for not only fault detection, but to clarify responsibility for any problems which arise. As the CS410s are passive gateways and use their own routing tables to connect each host across the WAN, UNIX routing utilities are of little use.

User Login Id's are also used to provide some organisational processes. Each district has a range of numerical login and group identifiers. A user may have a different numerical login identifier in each district they access, but the user's login name is identical on any host and unique to that user throughout the network. This is accomplished by maintaining a global user file on one machine which is updated by the administration menu utilities as each local /etc/passwd file is modified. The GCOS (user's real name) field is used to include the user's job title and their home district or branch. This assists housekeeping procedures and the identification of a large user group. This configuration was chosen because similar information on a commercial data base may not always be accessable and Yellow Pages does not provide the required level of control.

Diverse Connection/Application Issues

The majority of our concerns stemmed from the diversity of equipment which the Network would support. Not only would it have to connect Mainframes, Unix departmental servers, SUN Workstations, Terminal Servers and PCs, but some types of equipment must access the network using more than one method.

Several IBM PC Ethernet products were tested with similar performance results. MICOM's NI5210 and NI9210 were selected to be the standard boards. It should be noted that some networking packages use direct hardware calls, such as ORACLENET, which will require customising to the Ethernet board selected.

The choice between suppliers of NFS software for the PCs became complex as their differences became more obvious. The final choice between SUN's comparatively small PC/NFS with application and screen printing and FTP's tape archiving facility and ORACLE support, fell in FTP's favour. Other than these issues, both give reasonable support of remote facilities and virtual-drives.

VT220 became the agreed standard and terminal configurations are changed to give identical user interface and keyboard functionality across all terminals or terminal emulators.

Diversity of applications also made it necessary to create an environment capable of supporting all users needs fairly. Dividing disk storage on the UNIX servers must combine several considerations. PC-users and virtual discs can cause up to 30% usage of Ethernet and fill large amounts of storage very quickly with products such as PC-ARC. Some ORACLE data-bases' growth must be allowed for, but initial estimates of requirements or growth are unavailable, while other databases such as "training" will not grow to the same extent but will need to be cleared and reloaded simply and quickly. Access speed of these databases can understandably be greatly affected by dividing the data needed across separate devices. Accommodating the wishes of potentially dangerous PC-users and a vague group of ORACLE users becomes slightly more complicated while trying to tune access of ORACLE data-bases the size and growth of which is unknown. By splitting before and after image ORACLE files across different disks and restricting PC virtual-drive areas to dedicated partitions encompassing individual areas for shared executable code, development tools, application conversion (to UNIX) and users' data, a standard three disk, one gigabyte storage configuration is defined for all machines.

MIPS M/120 Disc Media Configuration

Disc One (ISCOD0)

/	/usr	/oracle	swap
22 Mb	170 Mb	85 Mb	41 Mb
Unix, Tmp	Unix : Files, Logs, Tmp Oracle : User Source (.INP) PC : System Executables (G:)	Before Image Kernel Executable	Unix Swap Space

Disc Two (ISCOD1)

/dbsdev	/dbs	/dbase3
96 Mb	96 Mb	126 Mb
Oracle Development Databases	Oracle Production Databases	PC Based Databases for conversion to Oracle (J:) Oracle Training Database

Disc Three (ISCOD2)

/users	/orap	/pcdevel
192 Mb	85 Mb	41 Mb
Unix : home directories, users files PC : User area (H:)	Oracle User Executables(.FRM) After Image	(I:) PC Develop

The tuning of the kernel is made difficult by the lack of information available concerning the nature of ORACLE access. Rough tuning on the MIPS M/120 was possible using batches of between 5 and 40 processes creating, modifying and deleting large databases simultaneously, but no information is available on which parameters affect which characteristics of ORACLE. Trial and error is not an economic method of tuning a kernel.

Security concerning ORACLE proved to be more difficult than was expected. A number of groups exist with differing read permissions on each database. Applications such as SQL*Plus (interactive SQL) could potentially corrupt a valuable production database. Therefore a user must only have read permissions when using these tools. The released updating procedures, however must have write permissions, while retaining the user's identity.

The first attempt at achieving this was simply to use the Switch User bit within UNIX to give the released code the ability to update ORACLE as necessary, while the users' permissions would only allow read access. This failed due to the shell exits included within ORACLE which would allow an ORACLE user to use their new effective ID to the databases' detriment.

The final solution includes an encrypted C program to call, with a username-password pair, a privileged user within ORACLE. The ps program is modified to restrict the -f option to super-user to prevent the display of this pair. In this way a shell exit continues to use the correct effective user id and write access is denied.

An advantage of connecting the network to an existing commercial mainframe environment is the inheritance of a well documented formal code releasing procedure. RCS was found the best utility on which to base the UNIX release management system. The result is a process which follows code through development, testing and implementation with a minimum of disruption and effort.

Using the CS410's protocol conversion to the ICL mainframe proved relatively simple. The CS410's support of remote printers and asynchronous terminals did not. Thanks to some code from the Sydney University, TCP pseudo devices are created and used to funnel data to and from printers and other asynchronous devices. To support NFS functions for remote or Ethernet incapable PCs, SLIP is necessary. The serial to TCP/IP conversion (SLIP) should take place at the interface between the two, in the CS410. This was not to be possible and it would be optimistic to imagine that if, when MIPS produced SLIP, it would be possible to use pseudo devices to direct the asynchronous connection of SLIP down the Ethernet to a port on a CS410.

The decision was made to restrict remote and non-Ethernet access of PCs to terminal emulation and file transfer. Printing, remote job execution, tape archive, mail and virtual drives are only supported on Ethernet connected PCs. An existing mainframe access PC program was modified to give file transfer and interactive video VT220 emulation to the network. It is foreseen that some of the facilities not offered by this program may be included in future enhancements.

A second issue which is yet to be finalised is the choice of a mail and office automation package. The Goldilocks approach to identifying the correct offering has seen UNIX mail, Mush and Elm as too unfriendly, ORACLE*Mail unsupported on all machines, Q-office and UNIPLEX as too complex and awkward. The search continues for a package which is "just right". In hindsight, a formal comparison of all available products conducted early in the decision process would have simplified the implementation.

Remote Machine Issues

The districts in which these machines reside are not only geographically remote, but contain very little, if any UNIX expertise. This difference from academic institutions is the source of the third group of considerations. A complete package of setup routines were written to allow a non-computer literate user to complete the initial operating system install, insert a tape and completely customise the machine for their district. As this could be run at any time and included ORACLE installation and customisation, it proved to be a sizable task. Thus far the system has proved to be a success and should form a firm foundation for standard systems throughout the network.

A second suite of procedures was created to give a friendly menu interface for all the necessary functions to the remote system administrator. Great care was taken to include a customised interface for all desired processes, as it is foreseen that the remote system administrators will never leave this menu system to run shell commands. In this way, typical super-user access may be restricted to this menu system. Security breaches are made easy to trace and responsibility given to the remote system administrator is reduced. Basic training for usage of this menu is given and should allow the remote system administrator to overcome all common tasks and difficulties encountered. The remote system administrator will not be expected to diagnose unusual faults in the network which may occur. A central administrator will have super-user access to remote machines for the diagnosis and correction of these problems.

The remote system administator will also have the responsibility for state changes in the UNIX server. The MIPS M/120 employs a key, similar to a PC AT with lock, unlock and reset positions. With the lock switch left in the lock position, the machine state cannot be changed. This avoids an unauthorised user taking the machine to Single User Mode and bypassing the machine's security. To avoid remote PC and UNIX users tape usage contention, the system administrator is also responsible for ensuring the correct tape media is waiting in the UNIX server.

The third shortfall in the standards set involves the postponement of anticipated hardware release dates. All procedures evolved under the expectation of a tape drive which could archive the entire configuration of media without user intervention. An EXABYTE drive capable of storing 2.2 gigabytes on one tape was finally released, but was unavailable as the implementations commenced. Temporary methods for the inbuilt tape drive had to be employed, creating disruption for the administrators. It did however, provide a taste of the problems which could befall a network implemented without prior standards.

SQL*Menu produced by ORACLE will be used to develop an extensive layer of user menus. The choice between menu drivers became a simple matter of which would run on all machines throughout the network.

Conspectus

Network Organisation

Communication Protocols
Address Allocation
Subnet Routing
Communication Analysis Tools
User Id Organisation

Diverse Connection/Application

Ethernet Adaption Products
NFS software
Terminal Emulation Standard
Disc Storage Division
Kernel Tuning
Database Security
Code Release Procedures
Remote Access
Mail

Remote Machines

Setup
Administrator Interface
Physical Security
Archive Media
User Interface

Conclusion

The greater the effort placed in setting standards and the more stable the environment, the smoother the implementation. Intelligent users combined with geographically confined networks will always be able to grow and evolve without such effort. Without the luxury of intense training or experienced users, the decisions made before the network is in place carry a great importance. Mistakes will still be made and changes necessary as unforeseen problems are discovered, but putting a standard stable environment in place can mean the difference between small simple changes and major re-organisation of every system throughout the network.

OLTP Performance – What's Behind the Smoke and Mirrors

Ken J. McDonell
Pyramid Technology Corporation
Melbourne
kenj@yarra.oz

Abstract

As the speed and capacity of UNIX* systems approaches that of conventional main frames, the database vendors and UNIX vendors are embroiled in a battle to establish reputations as serious, high-performance, on-line transaction processing (OLTP) platforms.

These efforts are characterized by "leap-frog" performance in which no one hardware-software combination can expect to claim the "fastest" title for very long, utilization of particularly un-scientific marketing warfare in attempting to justify performance claims, and emerging better marriages between the DBMS products and the UNIX platforms on which they execute.

The paper will explore these issues based upon experience gained using the major UNIX database products for synthetic benchmarks, customer benchmarks and in-depth performance analysis. Some consideration will be given to pending developments that may help "level the playing field" for competing performance claims.

1. Emergence of OLTP Performance

UNIX systems have been, and will increasingly become, important as database servers in on-line transaction processing (OLTP) environments. This assertion is based upon an emerging trend that will be accelerated by the following facts.

1. In terms of raw CPU power and disk subsystem capacity and performance, several UNIX vendors already have products that complete in the traditional main frame and proprietary operating system market place, where OLTP is central to corporate DP strategies.
2. Despite the disproportionate fraction of current DP budgets going to PC equipment, large shared database will remain under centralized control and administration, with database servers being asked to support very high transaction rates on behalf of networked client processes executing on PC's. Note that "personal computing" often involves sophisticated manipulation of "corporate data", and replication of this data across many small machines is an operational nightmare that potentially compromises data integrity and extracts a hugh premium in storage costs.

   UNIX provides the best possible platform for vendor-independent implementations of these heterogeneous networks in which the smaller machines rely on servers to support efficient high-level access to centralized databases.
3. As UNIX provided software portability in the early 1980's, protecting software investments, so the UNIX-based DBMS products now are beginning to provide application and database portability across a wide variety of hardware platforms.

   Applications developed for any of Ingres, Informix or Oracle may be ported at the source-code level to many different platforms (both hardware and operating system) with little effort and preservation of the application semantics and functionality.

* UNIX is a trademark of AT&T

Unfortunately, despite the standardization (ISO, 1986) and widespread acceptance of SQL, applications are not likely to be portable between database products due to,

a. non-standard extensions to the SQL language,

b. necessary physical storage parameters that are outside the standardized CREATE TABLE and CREATE INDEX statements,

c. variations in the programming interface for applications with embedded SQL statements, and

d. the absence of standards for presentation layers and the tight coupling between forms packages and an underlying DBMS product.

To make matters worse, application source code compatibility between major releases of the same DBMS product has been a considerable problem (e.g. Informix to Informix Turbo, Oracle 5.1 to Oracle 6.0, Ingres 5.0 to Ingres 6.1).

However, on balance, the UNIX-based DBMS products provide better insurance against major application code rework and a better range of price/performance platforms than any of the alternative DBMS products with a tight marriage to a proprietary operating system.

4. Demands for throughput at the top end of the marketplace will continue to outstrip the capacity of uniprocessor architectures – those systems best prepared to deliver serious multiprocessor performance are predominantly UNIX-based (in part the hard work of building symmetric multiprocessor operating systems has already been done for several UNIX implementations).

As OLTP performance becomes an increasingly important evaluation criteria in the acquisition process, UNIX vendors and DBMS vendors are involved in a ''leap frog'' battle to establish themselves as the ''fastest OLTP platform''. However the rate of change (driven by new hardware, new operating systems releases and major DBMS product re-engineering) is so rapid that no product will retain the ''fastest'' title for long. This may be irrelevant, since to date, the criteria by which ''fastest'' is measured will be seriously questioned later in this paper, and hence the comparisons may have little or no bearing on deliverable performance in common operational environments.

2. DBMS Evolution

Historically the DBMS vendors have been comparatively slow to implement performance enhancements into the basic architecture, operational model and algorithmic implementation of their products. Results from the research community and pragmatic operational considerations have taken some time to become visible in shippable products, for example

1. **Storing the database outside the UNIX file system.**
   If done properly, there is a significant boost to both performance and data integrity that comes from using the raw disk interface rather than the conventional UNIX file system.

2. **Disk striping.**
   Large databases typically have very non-uniform access patterns, and so it is important to spread disk bandwidth demands across multiple devices by ''striping'' logical objects (e.g. relations or indices) across multiple physical devices.

3. **Batched commit.**
   Under heavy update transaction loads, writes to the transaction log may be batched to commit several transactions with a single physical (and synchronous) log write.

4. **Finer granularity of locking.**
   Concurrency control implementations usually rely on locking. Finer locks (e.g. at the page or tuple or attribute level, as opposed to the table or database level) allow (but do not guarantee) higher concurrency in exchange for more computation to manage the locks.

   The use of intentional lock modes extends the simple minded ''share-exclusive'' paradigm to reduce deadlock probability and improve overall throughput (Gray, 1978).

5. **Recovery schemes.**
   It is inevitably expensive to implement the necessary transaction logging and physical database page logging to support transaction integrity and database recovery in the event of a system failure. As pressure for this level of DBMS service grows, the vendors have been forced to consider implementation schemes that have a lesser performance impact than the earlier approaches.

6. **Memory utilization.**
   As systems are being engineered to support many hundred to several thousands of concurrent on-line users, any DBMS architecture that carries a significant memory overhead per user will not be cost competitive. The emergence of "server" architectures (e.g. Sybase and Ingres 6.1) is motivated largely by memory considerations, as is the almost universal use of shared memory for database buffer caches and concurrency control data structures.

7. **Buffer management.**
   The advent of large buffer caches (e.g. more than 40 Mbytes) has challenged the underlying assumptions of many buffer management strategies. Effective utilization of a large buffer cache requires adaptive algorithms that are sensitive to the logical access pattern (e.g. random retrieval suggests LRU, serial scanning suggests MRU), and to the different usage characteristics of pages of different types (e.g. root page of an index, or leaf page of an index, or data page, or free space/control page).

8. **Scalability.**
   At about the time Edition 7 UNIX was released, there was widespread acknowledgement that the simple data structures and linear search algorithms in the kernel would need to be replaced as the number of objects (processes, files, buffers, etc.) started to increase dramatically.

   Twelve years later, the DBMS developers are starting to become painfully aware of the same problems, but static sizing of critical hash tables, context free linear bit-map searches, linked lists and control parameters with absurd granularity are all still evident in the current DBMS products. The resultant performance degradation is startling as we move into multi-Gigabyte databases, buffer caches with hundreds of Megabytes and thousands of concurrent transactions.

For the most part, these evolutionary changes have been implemented by the DBMS vendors on an abstraction of the operating system interface that maximizes DBMS code portability – this is often the lowest common denominator of all UNIX variants and DEC's VMS operating system.

Performance suffers because the abstracted operating system interface is functionally weak, and many critical services are implemented in modules of a "compatibility" library using the most elementary C and UNIX system calls. For example, the standard C string routines tend to be re-implemented using loops and pointers to characters, because "the routines were broken at one time in vendor X's UNIX port". Whilst this approach maximizes DBMS source portability, it also insulates the DBMS from any improvements in the performance of UNIX vendor's library routines.

Better performance demands a better uniform level of service from the operating system **and** utilization of these services by the DBMS code.

3. Operating System Evolution

The vanilla UNIX kernel was engineered for time-sharing, and is seriously deficient in some respects for high performance OLTP. Consequently, UNIX vendors are developing kernel extensions that do not impact System V and POSIX standards conformance, but provide additional services oriented towards the requirements of the DBMS products.

The following list is by no means exhaustive, but suggests the directions that these extensions are taking.

1. **Fast mutual exclusion.**
   The DBMS concurrency control implementations are typically built on some mutually exclusive lock mechanism to protect the critical sections of lock management code. These short-term internal locks (or latches) are also required to protect DBMS internal objects such as page buffers, indices and transaction log buffers.

Conventional UNIX mutual exclusion primitives (semaphore operations, pseudo device lock drivers and lock managers) are just too slow because a system call and/or IPC operation is required and the implemented functionality is a gross "overkill". Fast mutual exclusion usually requires hardware support (e.g. bit set and test) and kernel co-operation (short-term timer waits and a sleep-wakeup mechanism that is cheap and flexible).

2. **Non-vanilla disk I/O.**
   A high-performance DBMS must be able to perform *guaranteed* and *ordered* disk writes to preserve database integrity and to implement the "write-ahead" transaction logging protocol (Gray, 1978).

   A DBMS server responding to requests from multiple client or front-end processes must be able to perform *asynchronous* disk reads and writes.

   Meeting these apparently conflicting requirements demands a new I/O interface (quite unlike that found in conventional UNIX systems) with asynchronous low-level interrupt delivery and/or non-blocking completion notification.

3. **CPU scheduling.**
   The conventional UNIX scheduler has several features that would adversely effect the performance of user-level processes implementing components of a DBMS. Degraded priority for long running processes is not a good idea for the permanently resident "heavy weight" processes of a DBMS server. CPU preemption by the scheduler whilst a high contention latch as held may also have a serious effect on performance.

   In multiprocessor architectures, the scheduler's "fair and even-handed" approach may cause significant process migration between CPUs, with a consequent performance degradation due to poorer utilization of the hardware caches.

   To support the various CPU scheduling strategies appropriate for a DBMS, user-level processes must be able to exert direct and adaptive control over the kernel's CPU scheduling algorithms. In addition, the default scheduling strategies must be guided by significantly more sophisticated heuristics in multiprocessor systems.

4. **Virtual disks.**
   Under certain circumstances, simplicity of DBMS implementation and effective disk utilization can be provided by a generalized virtual disk facility within the kernel. A single implementation layer can support.

   a. striping, for effective spindle load balancing

   b. mirroring, to provide guaranteed up-to-date backup and load-balanced reading,

   c. concatenation, to present a homogeneous address space spanning multiple spindles, and

   d. memory (or RAM) disks.

   Combinations of the above should also be possible, leading to several interesting operational configurations. For example mirroring a RAM disk with a real disk gives memory speed retrievals with guaranteed non-volatile storage, and dynamic reconfiguration of mirrored disks can be used to implement novel back up schemes.

5. **Light weight processes.**
   Within an OLTP system there are many concurrent transactions. Execution of the associated DBMS code may be achieved by dedicating one (or more) UNIX processes to a single transaction and then using the kernel scheduler to multiplex the available CPUs. The attendant process context switches tend to be expensive, and so an alternative scheme favoured in the server DBMS model is to multiplex many concurrent transactions within a single UNIX process (i.e. the DBMS server). The resultant transaction context switches occur in user rather than kernel mode, are comparatively cheaper (a process has much more context to save/restore than does a transaction) and preserve hardware instruction and data cache context.

   The switching of transaction or light weight process contexts can be assisted by the provision of special optimized library routines to save/restore registers, stacks, etc. In the case of the experimental

Camelot DBMS (Spector, Pausch and Bruell, 1988), the light weight transaction support from the underlying Mach operating system (Rashid, 1986) is very extensive.

At least one UNIX kernel[1] already supports these capabilities, however considerable joint software engineering is required to ensure that the the various DBMS products are modified to use and optimally exploit the additional functionality the kernel has to offer.

4. CPU Evolution

The demands of DBMS implementation are also evident in CPU architectures, for example.

1. Instructions that efficiently guarantee mutual exclusion, even across multiple processors.
2. Extended instructions, e.g. direct implementation of the operations and semantics of the null-byte terminated C "string" data type, aligned block move, generalized block move, hardware clock access and control, etc. These specialized instructions are visible in both RISC and CISC architectures.
3. DBMS code is typically large (bigger than a UNIX kernel) with long convoluted execution traces – this often has disastrous effects on the performance of processors with small instruction caches. Data access tend also to have a large working set, which motivates CPU architects to incorporate large data caches.
4. Since scalability is important for DBMS performance, the emphasis in hardware design has been on either families of uniprocessors with radically different price/performance, or symmetric multiprocessor architectures. Successful symmetric multiprocessor designs must address issues of interrupt delivery, data cache coherence and memory bus bandwidth in order that near linear performance is delivered as the number of CPUs is increased.

5. Measuring OLTP Performance

Despite all the efforts that have been made to develop fast platforms for high performance DBMS implementation, there remains very little in the way of reliable and quantifiable methods that can be used to measure performance, either between DBMS products or between different versions of the same product.

Outside IBM and its proprietary Ramp-C benchmark, most emphasis (in marketing "hype", the trade press and product engineering) has been given to the "debit-credit" benchmark (Anon. et al., 1985), or more accurately various perversions of "debit-credit" under the single misleading banner of "TP1" (or "ET1", or "[A-Z][A-Z]1" for *grep*(1) devotees).

The original debit-credit benchmark specification grew out of customer application requirements in a banking environment. The single transaction updates 4 tables in the database. The objective is to maximize the number of transactions that can be processed per second (TPS).

At the time it was published, debit-credit represented the first attempt to tightly specify a synthetic benchmark that could be used for OLTP performance comparisons across heterogeneous DBMS products.

Unfortunately, the widespread use of debit-credit as a general benchmark has resulted in a whole series of DBMS products that have been specially optimized for debit-credit performance. However, most database applications (especially in the UNIX market place) present a transaction profile that is very different to debit-credit in the following ways.

1. Applications typically involve a significant component of keyed retrieval (e.g. customer look up or account retrieval, or a relational join involving an indexed attribute), in which **one** table is accessed and **no** transaction logging occurs.
2. Update transactions often effect a small number of tables (most often one!), thereby experiencing reduced lock durations and higher potential concurrency.

[1] Pyramid Technology's OSx, version 5.0

3. Some applications require much more complicated queries which are sensitive to query optimizer effectiveness (an area not tested at all in debit-credit transactions). Examples: financials packages and database modules for expert systems.

   Query optimizers are notoriously weak and unstable in several of the UNIX database products, relying on weak heuristics and pattern matching rather than true optimization based upon estimated query execution cost.

Consequently real applications may not see a great deal of benefit from the performance enhancements to DBMS products that have been driven by the debit-credit benchmark, and comparative performance based on debit-credit transactions per second may have little predictive accuracy for other transaction and application profiles.

To make matters worse, TP1 implementations have been subject to flagrant violations of the debit-credit specifications that have dramatic effects on performance. The following examples (drawn from analysis of real TP1 performance claims and TP1 implementations) serve to substantiate the assertion that quoted TP1 figures are **irrelevant** to rational considerations of DBMS OLTP performance.

1. The original debit-credit specification included X.25 communications and a simulated network of ATMs. By ignoring this, TP1 implementations tend to avoid
   i. CPU cycles to drive the communications network.
   ii. Management of a large number of terminal sessions and contexts.
   iii. All perturbations and queuing problems associated with non-constant think times and transaction inter-arrival times.
2. Although the original specifications clearly state rules for scaling the database size according to the absolute transaction throughput, many implementations ignore this in favour of smaller database sizes – this has an operational and pragmatic justification (big databases take a long time to set up and require lots of disk storage), however the resultant throughput numbers are invariably inflated.
3. For each transaction, three tables are accessed randomly and one is appended to (i.e. serial writing). By using creative (but operationally nonsensical) storage structures it is possible to reduce or even remove lock and page access contention.
4. Usually in conjunction with an under-sized database, the use of very large buffer caches can avoid many I/O operations, however the resultant performance will not scale with database size (other than with prohibitive system cost).
5. Avoiding all transaction logging and database recovery (e.g. don't let a checkpoint happen during the timing interval) will make a dramatic difference to the throughput. Along similar lines, avoiding all concurrency control and locking will also make a big difference. Unfortunately issues such as lost transactions, data integrity and failure vulnerability are never mentioned alongside the claimed TPS numbers.
6. Either disabling or circumventing the access controls and authorization mechanisms will generally inflate performance.
7. Artificial multiplexing or replication of the database and/or transaction log may be used to avoid ''hot spots'' found in conforming implementations.
8. The original specifications included a response time criterions (90% of transactions less than 2 seconds) – by ignoring this, throughput at 100% resource utilization is often reported.
9. Some TP1 numbers have been produced with specially crafted ''non-products''. These tend to be either experimental or prototype systems with substantially reduced functionality – should a deliverable product evolve from this (and in some cases even this weak intention is not present), the performance is likely to be substantially inferior.
10. Up to a point, big TPS numbers can be obtained by throwing more hardware at the problem. The original debit-credit specifications also called for the ''cost per TPS'' to be reported to help establish how expensive it might be to achieve a specified absolute TPS rate. Many published results do not

report the cost per TPS, or (worse) do not follow the original costing specifications (which include a very large amount of disk storage and on-going maintenance charges).

In an attempt to address the flagrant liberties being taken in the "TP1–sleaze wars", more than thirty companies (hardware vendors and DBMS vendors) are collaborating in the Transaction Processing Performance Council[2] to develop the TPC-A benchmark. TPC-A is based upon debit-credit, but requires a level of functionality, transaction management, operational sanity, costing procedures and testing disclosure that will greatly reduce the opportunity for misleading or deceptive performance claims.

The first draft of the TPC-A benchmark specifications is likely to be available for public review later in 1989.

More importantly, TPC-A is the first of a planned series of benchmarks. Much of the hard negotiating and drafting in the first twelve months of TPC meetings has resolved the difficult issues of benchmarking practice, DBMS functionality and how you test for it, disclosure of testing procedure, system configuration and results, and system costing procedure. Once agreed upon, these clauses may be incorporated with little change in any subsequent DBMS benchmark specifications. The addition of new applications and transaction specifications should be comparatively straight forward, and lead to a set of robust, reproducible and reliable benchmarks for a variety of processing profiles.

Several other groups may also develop better standard performance measures for DBMS applications. The Systems Performance Evaluation Cooperative (SPEC)[3] has been developing a suite of CPU performance benchmarks, but has indicated that their work will extend into multi-user environments and possibly DBMS application environments. The /usr/group Performance Measurements Working Group[4] is beginning to address the issue of workload characterization, including transaction processing applications.

6. Conclusions

UNIX database products have evolved dramatically over a relatively short period from the early educational and research systems – it is not that long ago that Ingres (Held, Stonebraker and Wong, 1975) was first developed at the University of Berkeley. In the interim, the improvements in hardware speed, storage capacity, operating system sophistication and DBMS research have generally outstripped the software engineering of the DBMS products.

Major performance enhancements will only come from a symbiotic evolution of the base hardware architecture, a UNIX kernel with specific performance enhancements to support the DBMS requirements, and database software that exploits the platform functionality.

In the are of performance measurement, much needs to be done. In the meantime, application-specific benchmarking is one of the few reliable techniques available to prospective purchasers to help unravel the competing claims of UNIX and DBMS vendors.

References

Anon. et al. (1985): A measure of transaction processing power. *Datamation*, vol. 31, no. 7, Apr. 1, pp. 112-118.

Gray, J. (1978): "Notes on Data Base Operating Systems". In *Operating Systems: An Advanced Course*, G. Hartmanis (ed.), Springer-Verlag, New York. also IBM Report RJ 2188

Held, G. D., Stonebraker, M. R. and Wong, E. (1975): INGRES - A Relational Data Base Management System. *Proc. AFIPS National Comp. Conf.*, AFIPS Press, Montvale, New Jersey, vol. 44, May, pp. 409-416.

[2] Further information of the membership and activities of the TPC are available from the administrator, Mr. Omri Serlin, ITOM International Co., PO Box 1450, Los Altos, California 94022, (e-mail to {uunet,sun}!uworld!omris).

[3] More information is available from SPEC, 65 Washington Street #138, Santa Clara, California 95050.

[4] Contact David Hinnant of Bell Northern Research at {decvax,akgua}!mcnc!ecsvax!dfh for more details.

ISO (1986): Information processing systems – Database language SQL. Draft International Standard 9075, Jun., 114p.

Rashid, R. F. (1986): Threads of a New System. *Unix Review*, vol. 4, no. 8, Aug., pp. 37-49.

Spector, A. Z., Pausch, R. F. and Bruell, G. (1988): Camelot: A flexible, distributed transaction processing system. *Proc. IEEE Comp. Society International Conf.*, San Francisco, California, Mar., pp. 432-436.

WHAT'S GOLD LOTTO ON-LINE ? UNIX

By David Moles and Mark Pickering
D M P Software Pty Ltd.

ABSTRACT

The customer problem seemed straight forward :- Provide a software solution that harnessed four loosely coupled processors into an online transaction processing system for Lotto game coupons. Software would run in a background mode to process logged transactions from all four machines. The Lotto coupons were to be processed online, in real time, in agencies throughout Queensland by up to one thousand point of sale terminals connected to the central machines by the TELECOM DDS network. The customer demanded a fault tolerant, highly secure product.

The hardware, 6/32 FT machines manufactured by CCI, was supplied with the PERPOS operating system, a BSD UNIX derivative with System V UNIX features incorporated. PERPOS supports mirrored disks, low level file control and inter-processor communication. These features were used along with locally developed software to implement the high speed, secure transaction logging component. In addition log files are periodically written to WORM Laser Disks to comply with Audit Department requirements. Fault Tolerance was provided by decoupling applications running in each processor in such a way that failure in a machine did not affect the other machine environments. Automatic recovery is incorporated in the event of a failure and network software is designed such that agency communication lines may connect to any processing unit. This provides hardware and network fault tolerance and allows load balancing to take place. The network and terminal software was also designed to be AS2805 compliant. The 2805 Australian Standard is used in all banking networks and provides encryption and message authentication facilities.

The Golden Casket Art Union in Queensland issued tenders for automation of it's lotto processing operation. The tender was won by Alcatel-STC and Bitwise was engaged to provide a software team. The Gold Lotto Project was officially started on 11th April 1988 and was commissioned online with a pilot of 20 agents on 17th April 1989.

INTRODUCTION

All software developers will admit, there is an element of risk in every large software project. The challenge for software companies is to minimise the risk, maximise the development output while maintaining product quality, thus providing software value for development dollar. So it was when Alcatel-STC together with a software team from DMP Software commenced the GCAU Lotto project.

Early in 1988 Alcatel-STC was awarded the contract to computerise the Golden Casket Art Union Lotto operation. The solution accepted consisted of four central computers connected to a large network of point of sale terminals via a matrix switch. Lotto coupons are fed into the terminals where they are imaged to determine the data, transmitted to the central computer where the coupon is processed and recorded on disk and then a reply to the terminal and final receipt printing completes the transaction.

THE BUILDING BLOCKS

The central computers are manufactured and supplied by CCI through Alcatel-STC. The 6/32 FT systems offer a UNIX environment in a multi machine, fault tolerant configuration. The 6/32 FT Systems are connected together via an Ethernet. The disk units are shared amongst all processing units and access is coordinated by a central resource manager. Each 6/32 FT system provides five, sixteen line communication processors for communication with the network of terminals.

The operating system, PERPOS, is based on Berkeley BSD 4.1 with System V features. PERPOS offers features to manage and control multiple processing units and provides a low volume transaction processing environment for asynchronous terminal operation. Tools provided include mirrored disk management, file system co-ordination, inter-processor communications and hardware failure notification via invocation of shell scripts.

The Terminal is manufactured by the Swedish company Esselte and supplied through Alcatel-STC. The terminal uses imaging technology to evaluate an electronic image of each coupon to extract the games and numbers marked by the customer. A dual head printer provides receipt printing. Each terminal has 1 megabyte of memory and is down loaded with software from the central computer system.

Up to six terminals connect to the central computer using a network line in a multi-drop configuration. The bit synchronous HDLC protocol running in NRM mode is used to control each line. Over 800 terminals are equally distributed across the four processing units using 160 lines. Complete network redundancy is provided through an additional 160 lines normally idle but available for switching via an AS-100 matrix switch.

The host system software was based on the Esselte Lottery Software written in Fortran for VAX systems. Some of the operational characteristics have been retained in the conversion with development done to supply system software support.

REQUIREMENTS

A medium volume transaction processing system was required to run across multiple machines providing :-

- 40 Lotto sales transactions per second.
- Transaction response times of less than 2 seconds.
- Single point hardware fault tolerance.
- Software reconfiguration in the event of failure.
- Network implementing the Australian Standard for encryption/network security, AS2805.
- Dynamic network reconfiguration through automatic recognition of terminals and allocation of HDLC poll codes.

PROJECT STRATEGY

The project was divided into two phases. The initial phase of three months was a conversion of the software from the VAX/VMS fortran system to C and UNIX system.

A System V UNIX development machine and environment was used for the conversion phase. Standards covering C coding style and naming conventions were enforced, the software library control written using simple shell scripts and programmer tools were drawn from standard UNIX tools.

At all times emphasis was placed on achieving a working system, so initial development used existing UNIX features with recognition that replacement would be necessary because of performance considerations. For example asynchronous communication links to the point of sale terminals were used in place of the more sophisticated HDLC software being developed in parallel and UNIX Pipes were used for inter-process communications.

Owing to the lack of familarity with PERPOS features, an evolutionary prototyping methodology was employed for the development of the application interface to system facilities. This allowed parallel development of application and system software.

Following the successful demonstration of the converted software, development commenced of the Golden Casket Art Union Office system. This required rework of system support software to provide the performance required, AS2805 security and encryption, HDLC network software integration and tailoring of existing features and development of new features for the Golden Casket Art Union.

TRANSACTION PROCESSING ON UNIX

The transaction processing component of the system offered the greatest challenge. Specifically this sub system accepts transactions from the point of sale terminals, processes the transaction either locally or by routing to a server process, allocates a unique receipt number, secures the transaction to mirrored disks, and returns a reply to the terminal after the transaction is on disk.

Generally the transaction system also controls sub-ordinate server processes within a processing unit required for processing time consuming transactions and processes that exist to interface with the network communications or inter machine communications. Control of transaction systems in each processing unit is also built in and is used to provide orderly recovery in the event of hardware failures.

In order to implement this sub system several layers of system software were designed and developed :-

- An inter-process message passing facility.
- Sub second timer control.
- Multiple asynchronous event handling.
- Securing of transactions on disk.
- Inter processor communications.
- Process control within a single processing unit
- Process control and recovery across all processing units.
- A high speed network interface incorporating network management software for the network of 800 terminals.

PERPOS was relied on to provide operating system support for :-

- Transportation of messages between processing units.
- Management of hardware units.

- Notification of hardware failures and invocation of application recovery programs.

A message passing service was required for process communication within a system and communication over the ethernet between systems. Each machine would have a central transaction logging process that must be capable of handling several real time events. Timing functions with low resolutions would be required by this process. Typically UNIX is not good at multiple event handling. The UNIX System V features that did exist were evaluated but discarded either for performance reasons or un-availability on the target fault tolerant machine.

The concept of mailboxes and mailbox buffers was developed to provide the message passing facility. This was first done using UNIX pipes and later enhanced to use shared memory buffers. Low level machine code was used in performance critical areas of this module.

Timing functions were developed that allowed a process to run several timers concurrently. The computer system manufacturer, CCI, provided a 1/60 second resolution timebase within the system.

All these functions were integrated with a central event dispatch loop. This uses a combination of UNIX signal catching and timely polling to ensure real time operation.

A disk system that supports cache bypass was mandatory to allow the application to control the disk writes. Transaction blocking in units matching the operating system buffer size was needed to achieve the throughput figures.

The interface to the network communication processors and the communications software for the network processors was designed with high speed operation in mind. Where possible UNIX device driver design concepts were adhered to. A special device driver was written to support the communication processors. A control and data channel both using a message passing interface allowed separation of the time critical data handling operation and efficient operation of the host computer communication process by polling for the availability of data.

Communication between machines was implemented using features inherent in the CCI 6/32 FT operating system. Where possible the operation of the application software was kept within the bounds of a single processor, with features allowing switching of one or more terminals between processors to provide fault tolerance.

SECURITY

The development group were in charge of two main aspects of security.

Firstly the security of data sent to and from terminals over the wide area network was to be protected from intrusion through the use of AS2805 message authentication codes (MAC). The MACs are generated using the message data and highly secure and unique 'message keys' using certified enciphering hardware in the Host and in the terminal. Each 'message key' is unique to a terminal so each MAC may be verified at the receiving end for validity. An intruder wishing to insert a message into the network must acquire the correct message key. This technique transmits the message data in the clear, however sensitive data is encrypted using private encryption keys.

The AS2805 implementation was underestimated. The estimates neglected the complexity of the host and terminal interface. Each relied on each other's correct operation before progress could be made. In general the secure nature of the encryption and message authentication precludes open discussion and makes the design and testing time significantly longer. This needs to be considered in project schedules.

Secondly the security and integrity of the UNIX operating environment was to be preserved. Correct adminstration of the system directories and protecting against known security holes was implemented. To protect and check the system integrity an integrity checking program is run periodically. This program checks file permissions and sizes against a known specification. This program also perfoms software releases to the production machine thus reducing risk of incorrect operation.

As a protection against fraud the audit department required that transaction log files be secured to optical WORM drives at regular periods during the day.

A significant deficiency in implementing a secure system was the lack of a system log to record security violations and record process activity in the system.

CONCLUSION

"What's Gold Lotto Online? - Simple".

The answer is yes!

As far as the application program is concerned the UNIX implementation is no more difficult than similar systems based on other Operating Systems.

The solution lies in supplying an application interface to the systems software which isolates the implementation specifics. This provides application flexibility but restricts the implementation specifics to a limited number of modules within the system.

As the Gold Lotto project has shown UNIX can be used as a platform for real time transaction processing. The system software that must be developed for these applications is primarily associated with real time event handling, message passing, subsecond timers, process control and monitoring and transaction logging. These can be implemented in a UNIX environment using standard tools whereas some computer systems provide these features as standard in the operating system.

Some limitations have been forced upon the system design due to the lack of integration with the operating system (communications interface to network) but these have been overcome to achieve the performance and functional goals of the system.

Having a partially working host system and terminal is much better than either end being in advanced stages without ever being connected. If there's one thing the C language and UNIX environment excel at, it's getting a design prototype working quickly. The prototype and progressive enhancement cycle provide excellent returns, the project management being more complicated.

1. INTRODUCTION

1.1 MAIN ROADS

Main Roads is a large and decentralised organisation responsible for the management of the Queensland declared road system. Main Roads provides advice to the Minister on the provision of Public Roads in Queensland.

Main Roads is decentralised into five geographical Divisions, each of which is composed of a number of Districts. There are fifteen Districts in total. Each District has a very high degree of autonomy providing fast and effective response to local needs.

Main Roads is responsible for a network of around 34,000km of roads throughout the State. Main Roads employs approximately 4,500 field staff and has an annual budget of around $500-$600 million.

1.2 REASON FOR TENDER

Since early 1987, Main Roads has undertaken a number of studies reviewing the information needs of the organisation.

The first study (MARIS) was undertaken in conjunction with IBM. The study recognised that the primary functions of Main Roads were road network management and road engineering. The Districts were identified as being the major generators and users of road management and engineering data.

It was highlighted that these core functions and major users were also the least supported by the existing information systems.

The second study (ARMIS) undertaken in conjunction with QCOM, identified a number of major engineering systems which were required to support the Districts. The study proposed a strategy for implementing a distributed computing environment and developing a comprehensive Road Management Information System.

These recommendations were adopted by Senior Management and led to the calling of public tenders for the supply of hardware, system and development software, and application packages.

2. METHODOLOGY

2.1 Overview

With the assistance of Coopers and Lybrand, a methodology was developed to evaluate the tenders and ensure the evaluation was fair and consistent, but as fast as reasonable.

The objectives of the evaluation were defined to be to:

(a) select the best possible systems for Main Roads within acceptable cost parameters.

(b) select organisations which could maintain a high level of support.

(c) maximise Australian context where possible.

The weighted scoring methodology for the detailed technical evaluation had a hierarchical structure. For each major aspect of the tender a set of issues were defined. In most instances, issues were subdivided into sub-issues.

At the bottom of the hierarchy of issues were features. A feature was defined as an issue which could be scored. Each feature has associated with it one or more clauses of the Tender.

Weightings for issues were developed by holding workshops with user representatives. Issues were ranked in a pairwise fashion and the weightings then calculated using the Analytic Hierarchy Process (Saaty, 1980). At each level in the hierarchy, the weightings for all elements linked to a common higher level element, add up to 100.

2.2 SCORING METHOD

Scoring was undertaken at feature level and recorded in a database. The raw score for each feature is determined based on a rating defined as follows:

0 - the tender does not comply

1 to 2 - the tender partly complies

3 to 4 - the tender complies but at a marginal level

5 to 7 - the tender fully complies

8 - the tender complies at a standard higher than the base requirement

9 - the tender complies at a standard substantially higher than the base requirement

Where a raw score was greater than 7 or less than 4, the scoring for the feature was checked by another member of the team serving as an auditor. The basis of the assessment was documented in the database.

Scoring at each level in the hierarchy was achieved using the database and a weighted summation of all sub-issues or feature scores.

To cater for the more impossible claims made by suppliers, which required further validation, two scores were kept, a maximum and a minimum score. The maximum score represented the claim made by the supplier while the minimum score related to what the evaluator had been convinced actually existed.

Maintaining a measure of the uncertainty in the scores assisted in identifying lower scoring tenders earlier in the evaluation than was otherwise possible. The two scores were progressively and eventually reconciled.

The evaluation methodology was divided into five phases.

2.3 Phase 1 Classification of Tenders

The team classified tenders into conforming and non-conforming categories. Each conforming tender was then divided into “responses”. A response is a section of a tender dealing with one of the major aspects and each response could be mixed with responses from other suppliers to form a total solution.

2.4 Phase 2 - Short-List Selection

The objectives of this phase were the identification of potential total solutions from the tendered complete environments and the selection from these of a short list of approximately five total solutions for further evaluation. Where applications were identified as being of strategic importance, they were evaluated during this phase and their weighted score added to that of any compatible total solutions. During this phase, mandatory features and some highly weighted desirable features were scored.

2.5 Phase 3 - Costing

Accurate assessment of the cost implications was to be achieved through development of a comprehensive cost model for each of the short-listed total solutions. Rules for excluding tenders on the basis of cost were developed in advance.

For the purposes of this phase, the costing details used were the best available at that stage in the analysis. It was expected that this would closely approximate final costings but would differ to some degree.

The costs of the environments which make up the short- listed total solutions were normalised to a minimum common standard (having regard to the mandatory and desirable items offered) by reducing or adding to tendered configurations. This process needed to take account of level of service issues such as response time and system availability and to ensure that the final configurations supported consistent operational performance.

The costs of the supporting services also needed to be normalised.

2.6 Phase 4 - Final Selection Phase

The objective of this phase was to complete the evaluation process and to recommend a final contractor and top-ranking total solutions.

This stage involved completing the short list technical evaluation including evaluation of all remaining features and consideration of the responses/objections to the proposed Conditions of Contract, and company viability issues.

Visits were to be made to a selection of user sites at which at least a substantial portion of the combinations of equipment and software was represented. The purpose of these visits was to obtain a quick double check that the results of the evaluation were valid and that there were no undetected viability issues.

2.7 Phase 5 - Trial Period

Subsequent to endorsement of a selection, the proposed contractor was to enter into a trial involving two configurations from the tendered range. This trial was intended to provide further confirmation of the results of the evaluation. Provisions were made to allow rejection of the selected contractor during the trial.

3. EVALUATION

The evaluation of the tender responses proceeded in accordance with the pre-defined procedures (as defined in the methodology) and concentrated primarily on scoring the technical features of the responses.

3.1 HARDWARE

Sixteen hardware responses were received. Out of these, four were rejected as failing the critical mandatory requirements. Of the remaining twelve, the scores dictated that five (four running UNIX and one running a proprietary operating system) would make it to the short-list. At this stage, we had also engaged a consultant to investigate whether a UNIX based configuration was suitable for the Main Roads environment.

It must be noted that there was little difference between any of the tendered computers and that all are in the acceptable range (which meant any machine chosen would be a viable option, although the difference in scores reflect real differences in the quality of the computers).

3.2 NETWORKS

Twelve network responses were received. Out of these, two proprietary networks were rejected as failing the critical mandatory requirements. Two proprietary networks were rejected because the corresponding hardware was rejected. Of the remaining eight, the scores dictated that three (two vendor independent and one proprietary) would make it to the short-list.

Again, most networks are in the acceptable range, but there were greater differences between the networks than in the case with hardware.

3.3 FOURTH GENERATION ENVIRONMENTS

Eleven 4GE responses were received. Out of these, four were rejected as failing the critical mandatory requirements. Of the remaining seven, the scores dictated that three (two vendor independent and one proprietary) would make it to the short-list.

3.4 TOTAL SOLUTION

The ARMIS Request for Tender stated that Main Roads would consider options for combining tenders for hardware, network and 4GE in any way that would optimise the advantage to the organisation. This compilation of total solutions must take account of technical compatibility, overall capability and viability from a commercial and management point of view.

The total solution considered for the short-list consisted of those utilising a proprietary network and those utilising a vendor independent network (at this stage about 60% of hardware and/or network features for each response had been scored).

Each of the vendor independent networks was considered in conjunction with each of the hardware options. In addition when there were doubts over the capability of proprietary networks, then the hardware also was considered in conjunction with the vendor independent networks.

Consideration of such a range of options was made possible through the use of the database tool which enabled projection of total scores for each configuration to be compared against the corresponding measure of uncertainty in that score. Poorly documented evasive tenders which left many questions unanswered, resulted in high uncertainty measures.

Based on this hardware and network assessment, five hardware, three network and two 4GE would progress to the short-list selection.

4. NORMALISATION

The objective of normalisation is to reduce the difference between each suppliers' response if comparable products can be found, to allow a more reasonable comparison of products. Not all products can be normalised because of some basic differences in each suppliers response. These differences will be reflected in the technical scoring.

Some products which can be normalised include, central processing unit memory, disk storage capacity, system printer capacity and the central processing unit itself.

4.1 COMPUTER SYSTEM PERFORMANCE ISSUES

Defining and understanding computer system performance is complex and inherent difficulties exist in making comparisons between different systems and alternative suppliers. Despite this, a choice has to be made, and this choice will be influenced by the perceived system performance.

The most widely-quoted and best-known measurement of system performance in the industry is MIPS. As the acronym implies, a MIPS rating measures how fast a particular processor can execute its raw processing instructions. It is a legacy from the earlier days of computing when all hardware performed instructions serially. Currently, whenever a new model is released, if MIPS are not directly provided by the vendor concerned, they will almost inevitably be extrapolated by the press releases attempt to highlight competitive advantages or weaknesses based on reported or estimated MIPS.

In recent years, the usefulness of MIPS has been criticised. Upon research of the benchmark, it becomes clear that a "10 MIPS" system from one supplier does not necessarily have anything in common with the "10 MIPS" system marketed by another vendor. It follows that, for buyers, MIPS figures will not yield an adequate understanding of how the two systems will perform in their business environment.

In the computer systems marketplace of today, MIPS as a measure of processor speed alone is largely irrelevant. This is because vendor architectures use differing design features such as multiple processing units, specialised microcode, instruction sets used by different models. Some processors can perform a limited number of instructions, whereas others are equipped to process many complex instructions, the execution of which may be equivalent to dozens of the simpler instructions.

Furthermore, MIPS as a performance measure takes no account of the processing load outside of the CPU such as work done by intelligent input/output sub-systems or intelligent workstations. As a result, with the development of peripheral technologies, MIPS have become progressively less relevant as a means of comparing two systems with different architectures. Consequently, as a measure of overall systems performance it says virtually nothing about the amount of productive work that can be accomplished by the system users.

While MIPS is still the most widely quoted and extrapolated performance measure available, they fail to provide a conclusive comparison of the commercial processing capability of different systems. Given the shortcomings of MIPS, other performance benchmarks are now coming to the fore. These are the LINPACK and Dhrystone benchmarks. They can be classified as typical single-thread performance measures. This means that they measure how fast a single defined task or process is performed by the system processing unit. These single-thread benchmarks do not include any wider consideration of factors such as intelligent I/O processors and workstations, nor the use of multiple processors to improve throughput. Consequently, whole single-thread benchmarks can be useful, they are limited by only reporting the performance of a distinct part of the overall systems architecture.

The earlier discussion of MIPS concluded that they did not provide any meaningful indication of overall system performance, nor did they offer any direct way to compare one vendor's processor to another.

Many major vendors now refuse to release MIPS figures for their machines because of their widely stated shortcomings. Despite this, these figures are almost always estimated in the trade press.

As MIPS is the most readily accessible measure available to purchasers, they have been chosen for analysis of their relativity to other benchmark data to assess their usefulness as a method of comparison.

The following analysis of the benchmark data listed will show that in the absence of any other benchmark data, "adjusted MIPS" can be used to provide a more meaningful approximation of comparative performance between two or more systems. This method was used by Touche Ross and is documented in their paper “Critical Factors in Evaluating and Selecting Mid-Range Computer Systems”.

4.2 CALCULATING "ADJUSTED MIPS"

In analysing the tabulated benchmark data, we chose in the first instance to compare MIPS against other processor performance indicators that are more inherently accurate, such as LINPACK and Dhrystone. As a start to the following analysis, two major assumptions need to be understood:

> As LINPACK and Dhrystone benchmarks report the results of actual recorded calculations, (MIPS does not) they are assumed to be more meaningful performance measures. (MIPS is a measure of internal activity, LINPACK and Dhrystone are measures of external activity.)

> If MIPS was a realistic comparative measure of performance then the ratio derived from dividing a recorded benchmark such as LINPACK by the MIPS rating would be approximately equivalent across all vendors. The clear implication from the variance in these ratios is that if the independently tested benchmark figures are valid as comparative performance measures, then then the MIPS rating is not.

While these ratios clearly vary considerably from vendor to vendor, there is a trend that indicates these ratios are relatively consistent within any one particular vendor's series.

More detailed investigation of the ratios shows that the relativity of the measures in various benchmarks could provide some useful calculable comparative measures.

Based on this observation some "MIPS Adjustment Factors" will be derived, in the first instance from the published figures from LINPACK and Dhrystone benchmarks we also asked suppliers for these figures. It should be noted here that these factors are not intended to provide any sort of definitive statement of system performance. What they are is an approximate assessment of processor performance that can be used to gain quick relativity across a variety of vendors in the absence of any published benchmark data other than a MIPS rating. Equally, in deriving the "MIPS Adjustment Factors" we have only referred to the two sets of benchmark data discussed earlier. As other benchmarks become widely quoted and published, the same methodology could be used to derive a broad understanding of comparative vendor system performance.

By using the average benchmark/MIPS ratio for the DEC VAX series as datum set at 1. 00, it can be shown from the independent benchmark data reported in the LINPACK and Dhrystone benchmarks, that the appropriate multiplication factor to adjust the MIPS for the other series listed can be summarised as follows:

The Series Processor Factor in the PROCESSOR FACTOR column of Appendix A, Table A is an overall processor performance ratio that has been arrived at by averaging, by series, the benchmark/MIPS ratios from Dhrystone and LINPACK figures in the previous columns of the table.

What follows in Appendix B, Table B is an illustration of how the previous analysis could be applied to assess comparative processor performance.

Purely on the basis of their quoted MIPS ratings, the processors for the average configuration in Appendix B, Table B appear to be of similar performance.

However, once the series processor factors suggested in Appendix A, Table A are applied, the MIP/120-3 processor rates over 2 time faster than the Pyramid 9810. This would illustrate that despite roughly equal quoted MIPS, the four models do not appear to represent similar processor performance. A more accurate method of selecting "equivalent" processors would be to select them based on adjusted MIPS as shown in Appendix B, Table B and Appendix C, Figure 1.

The normalised configuration for each supplier were audited and the suppliers informed of decisions taken, comparison was then based on these normalised small, average, large and extra large configurations.

5. RECOMMENDATION

Based on the comparative analysis of the strengths and weaknesses of the short-listed solutions, the recommended selection of timeliness for ARMIS was:

MIPS processors to be provided by ICL

Computer Protocol networking equipment provided by Datacraft

ORACLE 4GE

and that a rigourous trial period be undertaken to ensure that this combination is proved.

6. CONCLUSION

This tender evaluation has sought to select a combination of hardware and system software, network and 4GE products which will provide a platform for the provision and operation of a set of systems for road management. The expectation of this environment, both explicit and implicit, have included:

(a) Provision of processing systems at sites distributed throughout the State in order to enable localised control of road management functions.

(b) Provision of local area network facilities at all sites to facilitate connection of terminals, PCs and printers and to enable high speed data transfer to PCs.

(c) Interlinking of all sites through a communications network which enables any terminal or PC to access any host and any host to transfer data to any other host.

(d) Provision of a relational database and 4GL environment on each processor which enables rapid development of new systems and which assists users to undertake ad-hoc enquiries and simple development. Implicit in this is the requirement for implementation of a distributed database whereby any user can access or update any data if permitted.

(e) For each of these products, a high level of vendor independence is required to allow a change in the future to an alternative supplier.

Through the range of products evaluated, requirements a, b, c and e can be met to a high degree. The extent to which expectations of 4GEs can actually be met requires some discussion.

The picture painted of 4GEs today is mainly derived from the glowing reports from vendor representatives in the form of sales talk, glossies or articles in computer publications. Few words are expressed by the user fraternity. This has resulted in the prospective user having high to very high expectations of the product in terms of productivity, software sophistication and ease of use.

We have found that most users are not as far down the development path as they expected to be after the implementation of a 4GE. They have found that the learning curve is steep and that a level of programming competence required to write programs of reasonable complexity still takes some months to attain (figures up 8 months were expressed). The number of systems in production collectively for all sites visited was disturbingly low.

It is still reasonable to assume that productivity gains can be achieved using 4GEs. The management of development teams will be critical to ensure success.

Our evaluation disclosed that the number of companies investing in 4GEs is increasing and that it could be expected that 4GEs will be with us for at least the next 10 years. If anything, the increase in the number of users is straining the support and development resources of the vendor companies in meeting the demands of these users.

All of the 4GEs still require considerable investment in research and development to provide the sophistication within the products that we have come to expect form the existing 3GE products in use today. In the bid to get the products into the market, some areas have not been addressed, while some areas are inefficient or difficult to use. These products are not yet ready to support large scale mainframe transaction processing systems employing non-stop processing and guaranteed response times. None of the products provide tools in the Data Administration/Change Control area displaying the equivalent of the current levels of control available in the existing mainframe environments. These controls progress corporate database structure changes through to production together with the modules affected by those changes.

At the commencement of the project we envisaged a method of using Distributed Processing in Main Roads. Unfortunately during the course of our evaluations, we could not locate a site which has adopted or anticipates adopting a similar method. Some had abandoned such ambitions after some research. ORACLE has distributed retrieval capacity (i. e. a user can query on data without being aware of where in the network the data is located) and is working hard at developing fully distributed databases. Main Roads will not require more than distributed retrieval for some time.

References

Saaty, T.L., *The Analytic Hiererchy Process : Planning, Priority setting, Resource Allocation.* McGraw Hill, New York - London, 1980.

Touche Ross, *Critical Factors in Evaluating and Selecting Mid-range Computer Systems*, Touche Ross Services, 1988.

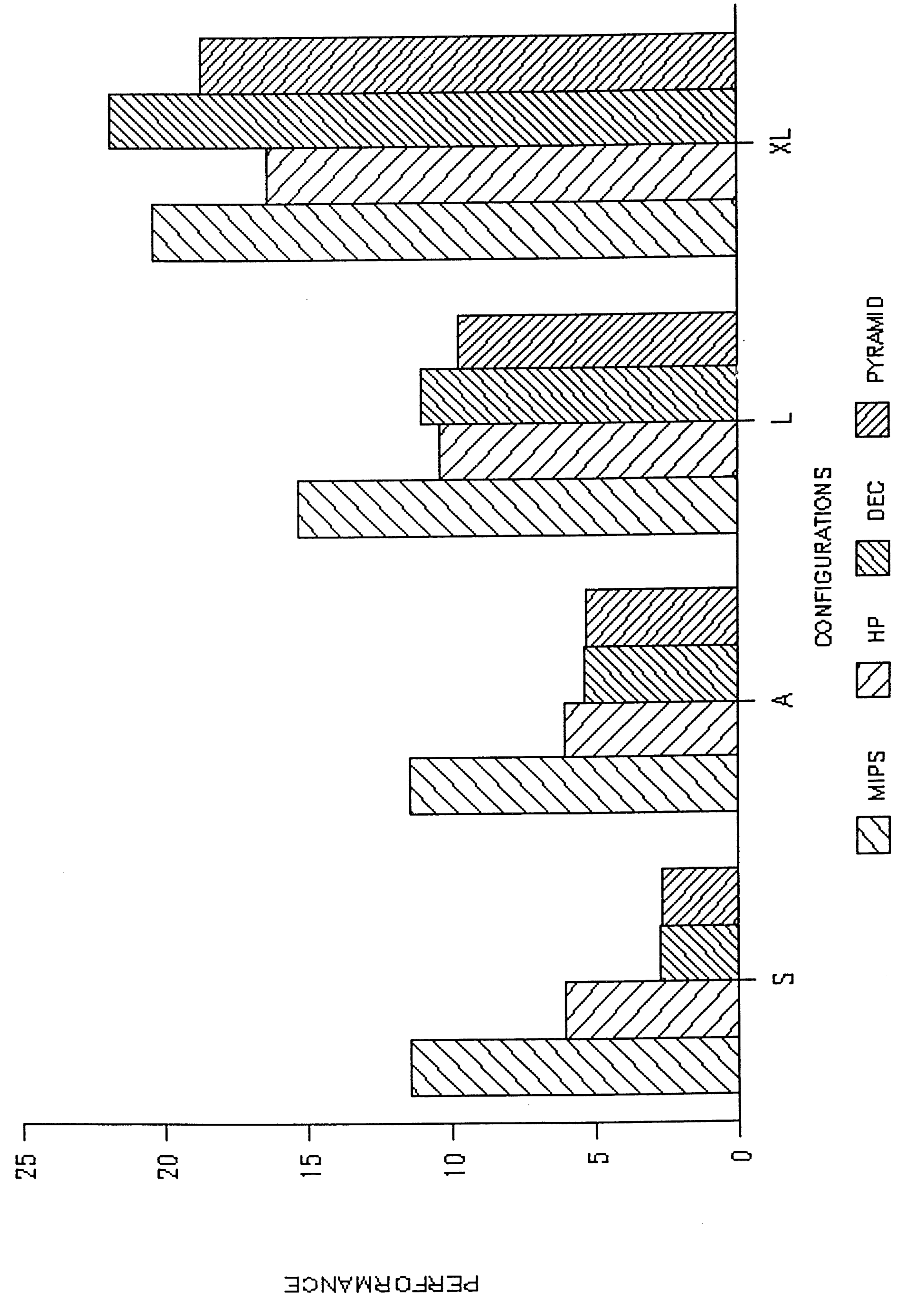
TABLE A

Model Name	Configuration	MIPS	Wetstone	Wetstone	Dhrystone	LINPACK	LINPACK	PROCESSOR	MIPS
			Single	Double		Single	Double	FACTOR	ADJUSTED
MIP/120-3	S	9.0	8600	6800	20500	3.00	1.60	1.27	11.46
MIP/120-5	A	12.0	11400	9100	27400	4.00	2.10		15.28
MIP/2000-6	L	16.0	13120	11040	33600	4.80	2.90		20.37
MIP/2000-8	XL	20.0	16400	13800	42000	6.00	3.60		25.46
HP 9000/810S	S	8.2	3597	2545	15625	0.64	0.50	0.74	6.08
HP 9000/825S	A	8.2	3597	2545	15625	0.64	0.50		6.08
HP 9000/835S		14.0	9000	6600	19400	2.30	1.80		10.37
HP 9000/835SE	L	14.0	9000	6600	19400	2.30	1.80		10.37
HP 9000/840S		8.0	3597	2545	12263	0.64	0.50		5.93
HP 9000/850S	XL	14.0	8100	6100	18735	2.10	1.68		10.37
HP 9000/855S		22.0							16.30
MVAX II	S	0.9	1225	750	1600	0.17	0.14	1.00	0.90
MVAX 3600	A&L	2.7	2975	1925	4600	0.66	0.42		2.70
VAX 6210		2.8	2975	1925	4700	0.67	0.42		2.80
VAX 6220	XL	5.5	5890	3812	9300	1.34	0.84		5.50
VAX 6240		11.0							11.00
PYRAMID 9805	S	3.5	2500	1315	4991	0.54	0.38	0.75	2.63
PYRAMID 9810	A	7.0	5000	2127	10121	1.02	0.50		5.25
PYRAMID 9815TA		7.0	5000	2127	10121	1.02	0.50		5.25
PYRAMID 9820	L	13.0	10000	4254	20242	2.04	1.00		9.75
PYRAMID 9825TA		13.0	10000	4254	20242	2.04	1.00		9.75
PYRAMID 9830		19.0	15000	6381	30363	3.06	1.50		14.25
PYRAMID 9835TA		19.0	15000	6381	30363	3.06	1.50		14.25
PYRAMID 9840	XL	25.0	20000	8508	40484	4.08	2.00		18.75
PYRAMID 9845TA		25.0							18.75

TABLE B

Model Name	Configuration	MIPS	PROCESSOR FACTOR	MIPS ADJUSTED
MIP/120-3	S	9.0	1.27	11.5
MIP/120-3	A	9.0		11.5
MIP/120-5	L	12.0		15.3
MIP/2000-6	XL	16.0		20.4
HP 9000/810S	S	8.2	0.74	6.1
HP 9000/825S	A	8.2		6.1
HP 9000/835SE	L	14.0		10.4
HP 9000/855S	XL	22.0		16.3
MVAX 3600	S	2.7	1.00	2.7
MVAX 3600 x 2	A	5.3		5.3
VAX 6240	L	11.0		11.0
VAX 6240 x 2	XL	21.8		21.8
PYRAMID 9805	S	3.5	0.75	2.6
PYRAMID 9810	A	7.0		5.3
PYRAMID 9820	L	13.0		9.8
PYRAMID 9840	XL	25.0		18.8

Appendix C


PERFORMANCE
0
5
10
15
20
25
S
A
L
XL
CONFIGURATIONS
MIPS
HP
DEC
PYRAMID

Sun, Surf and X in California.

Andrew McRae

Megadata Pty Ltd..
2/37 Waterloo Road
North Ryde
andrew@megadata.mega.oz

ABSTRACT

"It's Heisenberg's Principle of Uncertainty all over again - you can either tell where the mouse is, or that it is moving, but not both"

Xhibition 89, an industry conference focusing on the X window system, was held from June 24 to June 27 in San Jose California. The conference consisted of a number of tutorial sessions, combined with a technical program and an industry trade show.

This paper attempts to summarise the salient features of the conference, and to highlight key issues. Particular focus is given to new developments, and the strategic directions occurring in the X world. The goal is to provide a 'snapshot' (screendump?) of the current state of X and associated developments, especially as it relates to the UNIX† world.

Introduction.

Xhibition 89 comprised of 18 tutorials, and 37 technical sessions, as well as a parallel industry exhibition, held over 4 days. The conference was well attended, with some sessions attracting several hundred participants. Whilst the central focus was on the X Window System, a number of tutorials and papers dealt with related topics in the general area of graphical windowing systems. The tutorial sessions ran for a half day each, except for a two part introductory tutorial which spanned two sessions. The technical program had a number of formats, ranging from panel discussions, to technical presentations, to multiple lectures within the same session. Keynote addresses were given by David Tory of Open Software Foundation (OSF), and Thomas Macy of Unix International (UI).

As an industry conference, it highlighted the strategic importance of the X Window System, and was a good testimony to the acceptance of X as a standard in graphical windowing systems.

Overall Impressions.

I came away from Xhibition with a clear and unmistakable sense that finally the computing industry has come up with a standard that *no one* is arguing about. The other conviction was that there is still a lot of debate about which window manager to use, which look and feel to adopt, which toolkit to base your application on, and which operating system to run on your computer.

One of the most frustrating arrangements of the Xhibition was that up to five sessions were held at the same time, thereby making it impossible to cover more than one in four sessions. The tutorial

† UNIX is a trademark of Bell Laboratories.

sessions overlapped the technical program for a day, and the industry exhibition was held over the final two and a half days of the technical program. At the end of the four days I felt that I had missed out on a number of sessions that may have been extremely worthwhile. I even changed sessions mid stream if I felt that the content was not as good as I had expected. A better balance may have been achieved if the conference had been a least a week long, and perhaps fewer papers.

The tutorials were partially disappointing, in that some tutorial material was covered in the technical program, and some of the tutorials were not presented very well. Other comments I heard rated the material from poor to very good, depending on the tutorial. Some material presented in the technical program belonged in the tutorial schedule, and vice versa. A summary of the tutorials is presented in an appendix.

The conference technical program was generally excellent in its coverage of a wide variety of subject matter, and in providing relevant and up to date information. Key areas covered included Look and Feel issues, X Terminals, Server issues (porting, evolution, improvements), Toolkits, Window managers.

The trade show was on the outside not nearly as large as I expected, but once I scratched the surface I was impressed with the commitment most vendors had in supporting X. There was lack of X based applications. More about this later. Most large vendors such as DEC, IBM, HP etc. did not have large stands or many workstations running X. Software houses selling X orientated tools such as user interface builders and compilers, or tools related to X seemed to make up the bulk of the Exhibition. A number of vendors were selling X terminals, which in themselves are interesting products. Overall the exhibition represented a good proportion of the smaller vendors and software houses specialising in X products, but the turn out of larger vendors was disappointing, especially as most of them are claiming conversion to the X cause.

Most of the systems running at the show were connected via ether ('shownet'), and a T1 line provided Internet connection, which allowed some very nice demonstrations of interconnectivity, and proved that X really did work as a machine independent graphics platform. *Framemaker* was the main application everybody would run to prove that their X box actually worked (the Lotus 1-2-3 of X terminals?), and I suspect that demonstrating it on most vendor's hardware did more for selling *Frame* than it did for selling the platform. It was instructive to watch it running on an X terminal, a PC X server, and a high end 20 Mip workstation. It certainly proved that a floating licence worked.

The State of X.

Since the release of R3, X has stabilised, and generally the attitude was that X is here to stay, and we had better all do something about it. It was encouraging to see that in the midst of double standards such as OSI & TCP/IP, OSF/1 & System V.4, NSF & RFS ad nauseum, X is seen as an accepted standard. It is also generally recognised that the closely associated portions of the X distribution such as the X toolkit are also considered part of the X standard, even though all applications may not use them.

Now that the basic platform is stable, a number of important extensions are forthcoming to address particular needs or future issues. The development of these extensions is not aimed at fixing deficiencies in the basic X standard, but are a natural evolution designed to build on top of a stable foundation, much as protocol layers are built on top of transport mechanisms, and link mechanisms.

One issue is how X will be ported to operate over an OSI based communications link. It highlights the fact that X does not rely on a particular transport protocol, but it does raise the question of where do you put X in the protocol stack? It contains session and presentation information, and may interact at the application level via window managers and inter client communications. It certainly leaves the purists in a dilemma.

PEX (PHIGS Extensions to X) is a maturing extension which enhances the server in providing 3D operations. The goal is to allow PHIGS/PHIGS+ clients access to 3D operations on the server display, without mandating the level of support (colour, hardware assist) the server is required to have. In the same spirit as X, PEX does not define the policy of the operations, but provides a mechanism by which higher layers may provide policy. PEX will be key in the acceptance of X in higher end graphic applications such as imaging, rendering, simulation etc. The provision of PEX will mean greater portability and market opportunities to software written to the standard, and the emergence of hardware dedicated to this area, as well as the use of standard workstations (maybe with accelerated graphics).

XIE (X Imaging Extension) is designed to support digital imaging and rendering under X, and is still undergoing some development in the X consortium. The idea is that XIE will provide server extensions to allow more sophisticated imaging and image data processing to occur in the client and server.

VEX (Video Extension to X) deals with low end video handling and transfer of video data; a nice example of this was a demo program running at the show which had 'Star Trek II' being shown real time in a window, which could be resized, moved or closed. As X is increasingly used as the standard graphics transport, and applications such as video conferencing become accepted, this is an important extension.

At a display level we've seen the development of sophisticated window managers; a new term has been invented to describe a program handling the connection and control of a display to a host (such as logging in and accessing applications), called a **session manager**. The session manager may be the same program as the window manager, but is usually a separate program controlling the access to various host resources. My understanding of a session manager is somewhat sketchy, as the definition overlaps with some window management functions, resource allocation and host accessing.

The adoption of the Inter Client Communication Convention (ICCC - nicknamed 'ice-cubed') has helped to standardise on the method of client interaction by generalising such operations as selection, text cut and paste etc. It is expected that ICCC will evolve to handle graphic object manipulation (picture cut and paste), and to formalise the communication between clients, window managers and session managers.

Release 4 of the MIT X distribution promises to have an even more overwhelming volume of software, and it is clear that X users and developers are getting the benefit of a large amount of software development in the industry. The openness of X is encouraging, with a large number of vendors placing software (as well as fonts) into the public domain via the X distribution. R4 will expand the core distribution by migrating some of the contributed software into it (I can visualise now that getting *your* window manager into the core distribution will bring showers of praise and elevate your standing to heights undreamt of), and start to address some of the gaps that the current distribution has, such as more window managers, useful clients, security measures, optimised servers etc. Older or obsoleted software will be retired to an accompanying 'unsupported software' section.

David Rosenthal and Jim Fulton chaired a discussion on X and Security; to sum it up, there is no security. Yet. R4 will have a start by putting some security hooks in, and it is likely that the general direction is to base it on authentication via *Kerberos*. Apart from the obvious problems to do with workstation security, and network transparency, X servers usually have no restriction on which clients have access to the display, and can even allow clients to 'grab' other windows and read the contents. An X based virus is too nasty to contemplate. Speaking to David afterwards he noted that it is unclear whether Kerberos will be allowed to leave the shores of the USA by the DoD (I guess he meant the DES stuff, not the generic Kerberos code), and he sympathised with developers in the Antipodes ('You could always move to the US').

To summarise, X seems to be a success in its basic goal to provide a platform independent graphics transport mechanism, and the current efforts are directed towards provided extensions to address specific technical needs without clashing with the fundamental nature of X, and to standardise on supporting structures such as toolkits, and look and feel.

Look and Feel.

The standards debate seems to have moved to a higher level away from X (e.g. X vs NeWS etc), and the Look and Feel question is currently the predominant issue that is being discussed; without doubt the two contenders being OSF's Motif, and AT&T's Open Look. Much the same attitudes abound as with the generic OSF/1 and System V.4 debate, with the difference being that you can at least look at and feel the contending products that result.

Open Look from AT&T is a Look and Feel specification, which defines the style of window presentation, and provides guidelines on the window operations. AT&T is selling a Open Look toolkit which implements the Open Look specification. Sun Microsystems have developed XView, an Open Look conforming toolkit, which is going to be included in the X11 R4 distribution, along with the X11/NeWS merged server. The AT&T Open Look implementation is based on the standard Xt intrinsics, whilst XView is not (a fact that may have significance, depending on the evolution and importance of the toolkit).

OSF/Motif, on the other hand (screen?), is also based on the Xt intrinsics, and the Motif specification not only includes the widget descriptions, but defines a Motif Style Guide (similar to Open Look in principle), and also includes MWM, the Motif Window Manager. The Motif Look and Feel is compatible with Presentation Manager. I was impressed with Motif as a Look and Feel, as it defined push buttons with a 3D effect, and the window borders had a similar 3D look to them.

It seems that most vendors are aligning themselves one way or the other depending on their basic loyalties, but the software developers are not revealing their hands until they feel one or the other is pre-eminent. In fact it looks likely that developers will have to support both toolkits to properly cover the X market.

The word **standard** was probably the greatest misnomer used when the various parties were describing their own Look and Feel packages, as it is difficult to claim to have a standard product when no copies have shipped to the real world. The best summary I heard was from Richard Stallman, stating the obvious that it matters not whether look and feel A is easy to use, or that look and feel B is nicer to look at, but that we only have **one** look and feel. I can only agree.

OSF and UI.

The keynote addresses were a disappointment, and I felt that I didn't learn anything new, except a new appreciation for rhetoric and the number of ways you can combine the phrases *open standard, open software, open solutions, open specifications, open systems, open extensions, vendor independence* and *vendor neutral.* The sound of beating of drums, and grinding of axes was unbearable.

I spent some time in discussion with some of the OSF staff, and it was enlightening to hear about the decision making process within OSF. I think that OSF is becoming aware of the growing disenchantment of the end users and software developers, and the general problem of not knowing the specific directions, intentions and timetable of OSF, and perhaps we shall see a more active role in distribution of information.

Tom Mace from UI seemed to come to grips better with the subject of real standards, but then he is in an enviable position of seemingly having a real product to sell.

All parties agree that the key entities that must be considered above all are the application software developers and ultimately the end users; it is useless to develop wonderful and creative standards if the field of play is deserted due to incompatibilities.

X Applications.

Due to the fairly recent acceptance of X, there was a noticeable lack of X applications. It is obvious that writing a sophisticated X application such as a WYSIWYG word processor is not trivial, and probably more complex than an application targeted to a particular hardware platform such as a PC or a Sun workstation, the reason being the application must handle indeterminate and unknown server problems (lack of memory, missing fonts) and other network orientated problems (response and round trip times etc), as well as the very reason that makes X unique; the ability to run the same application on a large number of different hardware graphic platforms.

Several approaches are taken to porting an application to X. Developers who designed their product to be fairly device independent usually placed an intermediate layer between the application and the graphic device handling; it was then reasonably straightforward to develop an X interface to replace this layer. The application would often undergo some basic changes to cope with the existence of a window manager, and to allow the use of the X event handling scheme.

Other developers are currently rewriting their applications from the ground up, placing a degree of trust in the stability of the X standard to provide a secure foundation. This scheme suffers from the (usually large) amount of work needed for software rewriting, but presumably benefits from better performance.

Older applications such as Uniplex II were developed around the tty interface, with graphics added for some smarter screens (e.g. Tektronix terminals). The approach taken to port Uniplex II to X was to develop an X filter to act as a smart terminal, converting the terminal sequences to Xlib calls, and filtering the X events back to the core application.

A number of software developers are releasing X based 'graphic shells', which provide a friendly, Mac like interface to Unix. Combined with the low price of X terminals, it is certainly a extremely cost effective and attractive solution as opposed to building a PC or Mac network. If a large vendor such as DEC or IBM bundled such a package to give Unix a visual interface onto multiple screens for their more powerful hardware, then it may do for Unix what the PC did for personal computing.

X based tools for developing graphical user interfaces provide a means for custom tailoring applications and user interfaces, without a large amount of developing. These meta-tools I envisage being used by large scale software developers to easily build new interfaces. It provides an impetus for the developers to concentrate on providing the solutions, rather than developing their own front ends.

It was also clear in the panel sessions that X applications must provide a greater level of robustness than the less commercially orientated X clients in the public domain, which typically do not handle, or handle poorly, the issue of screen resolution difference, server errors (out of memory etc.), font differences, Inter Client Communication Conventions (ICCC) etc.

To aid in the porting from older style windowing systems, some vendors such as Sun are providing migration tools such as XView, in an attempt to lever the software effort and crank out real applications based on X.

1990 will probably be the year that X based applications will start to hit the market in a much bigger way then they have up to now. Due to the wide acceptance of X, platform dependent software will fade, and X based clients will allow the easy mixing of a broad range of graphics hardware and applications. I heard of a number of packages that are being rewritten for X, or being ported to X, such as CASE tools, word processing software, CAD tools etc.

X Terminals.

X terminals are a fascinating market segment, and the panel discussion on X terminals was in my opinion the highlight of the conference. A number of factors has contributed to the rapid definition and growth of this area, such as the acceptance of X as a standard, the use of Ethernet and TCP/IP as a communications backbone, and the price advantage over more powerful workstations. One quote succinctly summed up the situation as "Ethernet is the RS-232 of the 90's, X is the ASCII of the 90's, and X terminals are the dumb ASCII terminals of the 90's" (The quote went on to say that "NeWS will be the EBCDIC of the 90's").

In many ways the growth of X terminals parallels the early growth of the workstation market, especially when users are starting to request more memory and more power in their display. The difference is that the X terminal will only be a display server, and will not support X clients (except perhaps a local window manager or telnet client). The X terminal vendors saw their market window cutting off at half the price of a workstation; as the price of workstations continue to decrease, the X terminal market may well be severly constrained. One vendor saw that the 3M principle applied i.e. an X terminal will have a 1 Megapixel display (monochrome), a 1 Mip processor, and a Megabyte of memory, usually with some ASICs to handle blitting.

The second half of this year should see the introduction of colour X terminals, but due to the cost of display hardware, it is likely that it will be more cost effective to have a workstation instead of a dedicated X terminal.

Critics of X terminals claim that network performance will suffer unduly when a number of X terminals are placed on a net, but it was clear from the presented network loading statistics that X terminals present a minor loading. A much more obvious imbalance is the CPU performance loading when you have a large number of sophisticated clients. A catalyst for the growth of X terminals will be the provision of applications that effectively use the display characteristics without being server bound.

Another problem for X terminals is the current reliance on TCP/IP for the transport mechanism, which in all reality precludes the X terminal from running over a slow communications channel (e.g. a 9600 baud modem link). SL/IP was not considered to be a fast enough link for X. A novel approach to this problem was shown by GraphON Corp., which runs the bulk of the server in a host, and communicates to a graphics terminal using a proprietary protocol; the solution worked very well from the performance viewpoint.

X terminals may signal a fundamental change away from the distributed workstation concept, to a 'super timesharing' system, which has a powerful central CPU running a number of X terminals over a LAN; this has the advantage of easy system administration, centralised data storage (ease of backup etc.) and appeals to the empire builder in all of us. Indeed the release of an X terminal from DEC may herald a sneak attack on the workstation market, especially if the huge installed base of VAXen is addressed using low cost X terminals, and a friendly and smart user interface.

X on the PC.

It is difficult to see where the PC will fit into the X world. A number of vendors sell X servers for the PC, but the fundamental problems are that a PC needs to have a network connection, and that the server usually has to run under some form of multitasking. The most common approach to running X was to embed the majority of the X server code in a dedicated graphics board, to which a simple PC resident program would feed all the requests that came in via the network. It was obvious that a PC with a memory resident X server, and network connection, left little memory or CPU time for any other processing to occur. The smaller resolution meant that some applications (such as Frame) had some limitations in the display capabilities.

Why is there interest, then, in running X on a PC? Simply because the installed base of PC's is so great, and that a lot of those PC's are on networks, and may wish to directly access Unix based X clients, and also run local PC applications such as Lotus (and Larry).

Window Managers.

Window managers are in the same category as editors; everyone has their favourite, no two are the same, they are usually customised to an unrecognisable degree, and everybody thinks that extra features should be added. It is a testimony to the basic philosophy of X that window managers are many and varied, and you can terminate the current one, and start a new one at any time.

There is a strong move to migrate some window management into the server for speed, and this is evidenced in the X11/NeWS server, which has part of the NeWS window manager in the server.

I attended a presentation at DEC WRL by Colas Nahaboo of Bull Research on GWM (Generic Window Manager). GWM is the Emacs of window managers, able to be extended via a lisp style language, providing a large degree of customisation. Interestingly enough, the project under which the software was written was named KOALA, hopefully not because they were out on a limb.

A number of vendors were selling their own window managers, usually combined with their graphical front ends. Window managers are also used in conjunction with the Look and Feel specifications, such as MWM (Motif Window Manager). Along with all the other window managers such as AWM, GWM, UWM, XWM, TWM and WM, I suspect you need to get in early to name your own WM, as there are only 26 letters in the alphabet.

Server Issues.

Some of the more technical presentations oriented around the X server, and most of them were stimulating and informative. The feedback from the Server Internals tutorial was disappointing; the technical presentations were better. Joel McCormack from DEC explained the great lengths he went to achieve 60K characters per second on the PMAX 3100 dumb frame buffer. Another speaker from DEC described porting the X server to a smart frame buffer (i.e. a display with a dedicated graphics processor). At the figures shown, it was questionable whether a graphics processor is an advantage or not, as often the overhead of accessing and formatting the requests is higher than if you did the operation using the main processor. Certainly it is a loss if your hardware doesn't quite do the 'right thing' e.g. draw lines ending on the wrong pixel.

It was considered that to achieve the goal of a working server, the server must satisfy the X protocol, and it must be reliable and robust. In the example of the DEC server, it was estimated some 15000 lines of maintainable code was developed, and some 2 man years of effort. It'is key to have a fast and reliable server, as performance sells the product.

Richard Stallman prompted a fair degree of brainstorming when he suggested some improvements to X and the X server. One of the major problems is the dependence on fast round trip times for mouse movement; it was suggested that a programmable server would provide extensible functions to handle a portion of window management, and also handle mouse movements locally. Also a novel approach was the concept of pie shaped menus, with the mouse being centred and the the direction of movement selecting the option; thus was born the notion of 'click ahead', allowing a number of mouse selections to take place without the appearance of the associated menu.

Lightweight processes should prove to be a boon for X servers, with a thread for every window providing a more elegant design and perhaps easier porting and scalability of X servers. R4 may well have some server work using LWP, and will be interesting to see.

The encouraging element was that ideas are being considered for the evolution of X and the X server, and whilst some are far fetched, a few will be incorporated into the protocol and the servers, and will continue to improve X.

Sun and X

I had arranged with Sun Australia to visit Sun's Mountain View HQ, but I was told the morning of my departure that all the engineers would be on a holiday, so I went to DEC instead.

With the release of the beta X11/NeWS server, and XView, Sun Microsystems may well have said 'if you can't beat them, join them', but from the demonstrations and time I have had to experiment, NeWS and X work very well together. Whilst the die-hards contest that NeWS is technically superior than X, and provides a better foundation for the future, X is here to stay, and Sun have responded to that.

A number of technical features of the release are worth examining. Without question the use of NeWS as an imaging model is an advantage. It highlights the role that PostScript (Display PostScript or NeWS) has in functioning as an imaging engine on top of X. I'm not sure what future NeWS really has in the X world (will it really be the EBCDIC of the 90's?). NeWS has some fundamental differences to X, and it may just be that both benefit from the marriage i.e. that some cross fertilisation will result.

I have to be honest and say that I do not think it is possible for NeWS to even contemplate overtaking X at this stage, due to the current impetus that X has, but perhaps X may be improved to operate with some of the nice features of NeWS, such as the concept of a programmable server, with lightweight threads for each window, and perhaps some embedded imaging in the server. NeWS may be viewed as a graphics engine running on top of X, giving access to sophisticated imaging, with some form of filtering to allow it to operate on generic X servers.

NeWS will probably fall into the category of some of the extensions to X (XIE, PEX, VEX), which not all users will operate, and not all servers will support. This is not to belittle NeWS, as it is obviously an extremely powerful and elegant system. It is probably a testament to the difficulty involved in merging the two diverse systems, and to the complexities of NeWS, that Sun has slipped the shipping schedule of X11/NeWS back to August.

The performance of the merged server is quite impressive, also showing how much effort has gone into making it work. I have played with the server on a Sun Sparcstation 1 with the GX graphics accelerator, and it is *fast*. I was told the X server would run some 5-7 times faster than the MIT distributed server on a colour Sun 3/60, which was nice to hear for those amongst us who have experienced the latter. My bet is that most people will use the server as an X server (except for existing NeWS users), rather than migrate applications to NeWS based displays.

XView may well be the dark horse of toolkits, as it is one of the only ones not based on the Xt intrinsics (widget based) toolkit. The object orientated programming interface is friendly and powerful, and providing SunView compatibility is a smart move (and very important). I liked the Open Look based Look and Feel of XView, and it was easy to drive; XView will be part of the contributed software to R4, so it will be interesting to see if it will be ported to many more machines. Sun themselves have ported the toolkit to (I believe) a DEC workstation to prove to themselves it is easy to port.

Miscellaneous Issues.

A number of fringe issues were discussed, which I won't go into detail on. They included internationalisation, with emphasis on text encodings, and standard symbol and icon definition (just what is the international symbol for a program?); porting X to VMS and Primos; different language bindings; Open Fonts in X; scientific visualisation.

Integrated Computer Solutions (the conference organisers) has given preliminary permission for reprinting conference papers in AUUGN, and I will try to obtain some of the key papers when the conference proceedings finally arrive.

Summary and Conclusions.

X has reached the level of accepted standard, and it is clear that virtually all graphic or workstation orientated applications will eventually migrate to X and allow independence from vendor hardware and software, similar to the manner in which Ethernet and TCP/IP allows interconnectivity of diverse architectures and vendor equipment.

Since X is considered stable at the protocol level, we are seeing higher layer standards emerging, albeit not without some labour pain. The jury is still deliberating on Look and Feel, and hopefully reason will prevail.

There is a proliferation of toolkits, and window managers, and this will only serve to stimulate ideas and growth in the market, as users and software developers build on the developments.

The acceptance of X has created new markets, such as X terminals, and it is worth observing how the larger vendors will respond to this potentially huge market. The only thing stopping X is a lack of application clients, which are coming in the near future.

X is also an evolving animal, and we must give credit to MIT and the designers of X that allow it to do so without major havoc. It is also a credit to the developers of X, both educational and commercial, that X is an *open* system, in the true sense of the word open.

A final conclusion is that micro-breweries are a redemption to U.S. beer, and you'll regret ever attending a Palo Alto Chili Cook-Off.

Boring Bits at the End.

The opinions expressed herein are mine alone, and I apologise profusely for any misquotes, or information that may be incorrect.

Open Look is a trademark of AT&T.

The X Window System is a trademark of MIT.

OSF/Motif is a trademark of OSF.

DEC is a trademark of Digital Equipment Corp.

PostScript and Display PostScript are trademarks of Adobe Systems.

XView, NeWS and Sparcstation are trademarks of Sun Microsystems.

* is a wildcard trademark of [matching organisation].

Appendix A

Tutorial Summaries

Below are summaries of the tutorials; some are from memory, others from the notes and word of mouth feedback. I have a complete set of tutorial notes if anybody has a particular subject they would like to follow on with.

Programming the X Window System.

Basically an introduction to X, aimed at new X users with little or no experience with X. I attended, but hoping to get a more formal grounding in Xlib and the standard toolkit, but found it too shallow, and too many interruptions from the attendees to maintain a good flow.

Using Widgets.

Describes how to program using the Xt standard toolkit. Widgets are class orientated X objects, used to provide a meta level of X programming.

Object-Oriented Programming with C++.

Nothing to do with X *per se*, but C++ is a natural language for manipulating graphics based objects, so we should see a growing interest in C++ in the context of X programming.

Programming User Interfaces With InterViews.

InterViews is a C++ based toolkit for X, developed originally at Stanford (I believe).

Creating Andrew Applications.

The Andrew Toolkit was developed at Carnigie Mellon University, and is part of the Andrew Development Environment Workbench; it is guaranteed against viruses.

Colour.

A tutorial on the intelligent use of colour in graphic displays, and how colour is used and implemented in X.

Getting In Tune With Motif.

A detailed look at Motif the toolkit, Motif the style guide, Motif the user interface language, and Motif the window manager.

Server Internals.

I was told that this tutorial was not very worthwhile, so I'm glad I didn't go. It basically was a summary of the MIT standard server(s), and the internal structure therein.

Widget Writing.

How to write a Xt toolkit widget.

Writing Portable X Code.

Ralph Swick once said that there is no such thing as portable code, only code to be ported. This tutorial aimed to give guidelines on how to write clients to handle different server and host constraints.

Fundamentals of Interactive Computer Graphics.

Where have I heard that title before? Describes the general concepts behind computer graphics. The tutorial notes has the best set of graphic references of books and papers I have ever seen.

User Interface Designs.

With the growth and sophistication of graphics, it is becoming much more important to have an interface that works with the user rather than against them, and this tutorial aimed at identifying the elements that comprise a good user interface.

XView, An Open Look Toolkit.

An enjoyable tutorial, as it gave insight into the design processes behind XView; aimed at the program interface level.

Tour Of The Xt Intrinsics.

The Xt toolkit is considered almost part of the X standard, as it gives a higher level of programming abstraction than Xlib itself. Basically described the design and available facilities of the toolkit.

Application Development With The AT&T Open Look Environment.

Describes the three major portions; the Window Manager, the Workspace Manager, and the File Manager.

Inter-Client Communications Conventions.

The ICCC describe how clients can cooperate in window and session management, sharing data, and using the X server as a common point of communication. The ICCC are a fertile area for growth, in providing a generic IPC mechanism, and in allowing sophisticated object sharing.

Display PostScript.

Describes how Display PostScript can be used an imaging engine on top of X.

PEX: The 3D Extension To X.

The extensions to X which give PHIGS/PHIGS+ 3D rendering within the X environment. I have the PEX draft documents for anyone who wants a copy.

SWiGS† Report

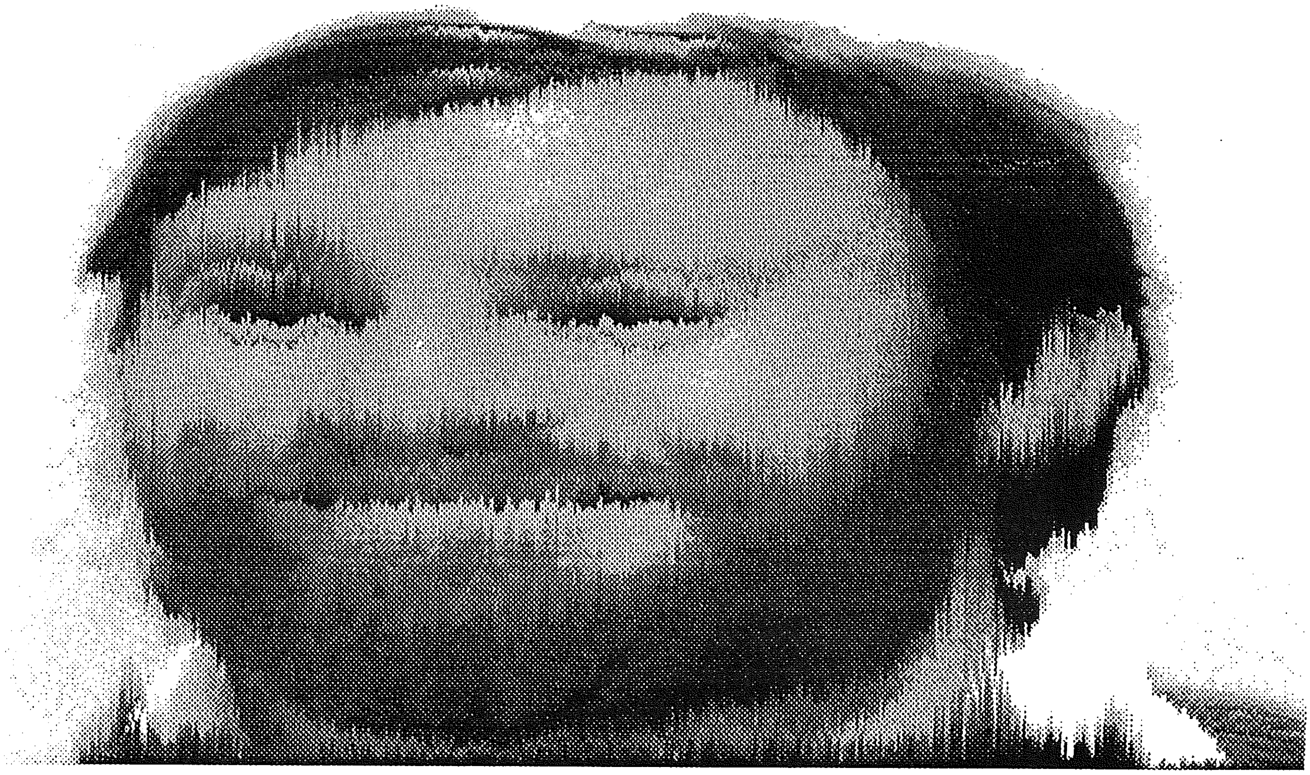
David Purdue

Future Editor of AUUGN

How Greg Rose looked before SWiGS:

† SWiGS is the Softway Wine Guzzlers' Society.

How Greg Rose looked after SWiGS:



And a good time was had by all!

AUUGN Back Issues

Here are the details of back issues of which we still hold copies. All prices are in Australian dollars and include surface mail within Australia. For overseas surface mail add $2 per copy and for overseas airmail add $10 per copy.

pre 1984	Vol 1-4	various	$10 per copy
1984	Vol 5	Nos. 2,3,5,6	$10 per copy
		Nos. 1,4	unavailable
1985	Vol 6	Nos. 2,3,4,6	$10 per copy
		No. 1	unavailable
1986	Vol 7	Nos. 1,4-5,6	$10 per copy
		Nos. 2-3	unavailable
			(Note 2-3 and 4-5 are combined issues)
1987	Vol 8	Nos. 1-2,3-4	unavailable
		Nos. 5,6	$10 per copy
1988	Vol 9	Nos. 1,2,3	$10 per copy
		Nos. 4,5,6	$15 per copy
1989	Vol 10	Nos. 1,2,3,4	$15 per copy

Please note that we do not accept purchase orders for back issues except from Institutional members. Orders enclosing payment in Australian dollars should be sent to:

AUUG Inc.
Back Issues Department
PO Box 366
Kensington NSW
Australia 2033

AUUG

Membership Categories

Once again a reminder for all ``members'' of AUUG to check that you are, in fact, a member, and that you still will be for the next two months.

There are 4 membership types, plus a newsletter subscription, any of which might be just right for you.

The membership categories are:

Institutional Member
Ordinary Member
Student Member
Honorary Life Member

Institutional memberships are primarily intended for university departments, companies, etc. This is a voting membership (one vote), which receives two copies of the newsletter. Institutional members can also delegate 2 representatives to attend AUUG meetings at members rates. AUUG is also keeping track of the licence status of institutional members. If, at some future date, we are able to offer a software tape distribution service, this would be available only to institutional members, whose relevant licences can be verified.

If your institution is not an institutional member, isn't it about time it became one?

Ordinary memberships are for individuals. This is also a voting membership (one vote), which receives a single copy of the newsletter. A primary difference from Institutional Membership is that the benefits of Ordinary Membership apply to the named member only. That is, only the member can obtain discounts on attendance at AUUG meetings, etc, sending a representative isn't permitted.

Are you an AUUG member?

Student Memberships are for full time students at recognised academic institutions. This is a non voting membership which receives a single copy of the newsletter. Otherwise the benefits are as for Ordinary Members.

Honorary Life Memberships are a category that isn't relevant yet. This membership you can't apply for, you must be elected to it. What's more, you must have been a member for at least 5 years before being elected. Since AUUG is only just approaching 3 years old, there is no-one eligible for this membership category yet.

Its also possible to subscribe to the newsletter without being an AUUG member. This saves you nothing financially, that is, the subscription price is the same as the membership dues. However, it might be appropriate for libraries, etc, which simply want copies of AUUGN to help fill their shelves, and have no actual interest in the contents, or the association.

Subscriptions are also available to members who have a need for more copies of AUUGN than their membership provides.

To find out if you are currently really an AUUG member, examine the mailing label of this AUUGN. In the lower right corner you will find information about your current membership status. The first letter is your membership type code, N for regular members, S for students, and I for institutions. Then follows your membership expiration date, in the format exp=MM/YY. The remaining information is for internal use.

Check that your membership isn't about to expire (or worse, hasn't expired already). Ask your colleagues if they received this issue of AUUGN, tell them that if not, it probably means that their membership has lapsed, or perhaps, they were never a member at all! Feel free to copy the membership forms, give one to everyone that you know.

If you want to join AUUG, or renew your membership, you will find forms in this issue of AUUGN. Send the appropriate form (with remittance) to the address indicated on it, and your membership will (re-)commence.

As a service to members, AUUG has arranged to accept payments via credit card. You can use your Bankcard (within Australia only), or your Mastercard by simply completing the authorisation on the application form.

AUUG

Application for Ordinary, or Student, Membership Australian UNIX* systems Users' Group.

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

To apply for membership of the AUUG, complete this form, and return it with payment in Australian Dollars, or credit card authorisation, to:

AUUG Membership Secretary
P O Box 366
Kensington NSW 2033
Australia

- Please don't send purchase orders — perhaps your purchasing department will consider this form to be an invoice.
- Foreign applicants please send a bank draft drawn on an Australian bank, or credit card authorisation, and remember to select either surface or air mail.

I, ... do hereby apply for

□	Renewal/New* Membership of the AUUG	$78.00	
□	Renewal/New* Student Membership	$45.00	(note certification on other side)
□	International Surface Mail	$20.00	
□	International Air Mail	$60.00	(note local zone rate available)

Total remitted AUD$______
(cheque, money order, credit card)

* Delete one.

I agree that this membership will be subject to the rules and by-laws of the AUUG as in force from time to time, and that this membership will run for 12 consecutive months commencing on the first day of the month following that during which this application is processed.

Date: __/__/__ Signed: ____________________

□ Tick this box if you wish your name & address withheld from mailing lists made available to vendors.

For our mailing database - please type or print clearly:

Name: .... Phone: .... (bh)

Address: .... (ah)

.... Net Address: ....

....

.... *Write "Unchanged" if details have not*

.... *altered and this is a renewal.*

Please charge $_____ to my □ Bankcard □ Visa □ Mastercard.

Account number: _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _. Expiry date: __/__.

Name on card: ______________________________ Signed: ____________________

Office use only:

Chq: bank ________ *bsb* ___-___ *a/c* ______________ *#* ______________

Date: __/__/__ *$* *CC type* ___ *V#* ___________

Who: ____________ *Member#* _________

Student Member Certification *(to be completed by a member of the academic staff)*

I, ... certify that

.... *(name)*

is a full time student at .... *(institution)*

and is expected to graduate approximately ___/___/___.

Title: ______________________________ Signature: __________________

AUUG

Application for Institutional Membership
Australian UNIX* systems Users' Group.

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

To apply for institutional membership of the AUUG, complete this form, and return it with payment in Australian Dollars, or credit card authorisation, to:

AUUG Membership Secretary
P O Box 366
Kensington NSW 2033
Australia

- Foreign applicants please send a bank draft drawn on an Australian bank, or credit card authorisation, and remember to select either surface or air mail.

.... does hereby apply for

□ New/Renewal* Institutional Membership of AUUG	$325.00
□ International Surface Mail	$ 40.00
□ International Air Mail	$120.00
Total remitted	AUD$______ (cheque, money order, credit card)

* Delete one.

I/We agree that this membership will be subject to the rules and by-laws of the AUUG as in force from time to time, and that this membership will run for 12 consecutive months commencing on the first day of the month following that during which this application is processed.

I/We understand that I/we will receive two copies of the AUUG newsletter, and may send two representatives to AUUG sponsored events at member rates, though I/we will have only one vote in AUUG elections, and other ballots as required.

Date: __/__/__ Signed: ____________________

Title: ____________________

□ Tick this box if you wish your name & address withheld from mailing lists made available to vendors.

For our mailing database - please type or print clearly:

Administrative contact, and formal representative:

Name: .... Phone: .... (bh)

Address: .... (ah)

....

Net Address: ....

....

.... *Write "Unchanged" if details have not*

.... *altered and this is a renewal.*

Please charge $_____ to my/our □ Bankcard □ Visa □ Mastercard.

Account number: _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _. Expiry date: __/__.

Name on card: ____________________________ Signed: ____________________

Office use only: **Please complete the other side.**

Chq: bank ________ *bsb* ____ - ____ *a/c* ____________ *#* ____________

Date: __/__/__ *$* *CC type* ___ *V#* __________

Who: ____________ *Member#* ________

Please send newsletters to the following addresses:

Name: ....
Address: ....
....
....
....
....

Phone: .... (bh)
.... (ah)
Net Address: ....

Name: ....
Address: ....
....
....
....
....

Phone: .... (bh)
.... (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	$ 90.00
□ International Surface Mail	$ 20.00
□ International Air Mail	$ 60.00

Copies requested (to above address) ________

Total remitted AUD$______

(cheque, money order, credit card)

□ Tick this box if you wish your name & address withheld from mailing lists made available to vendors.

Please charge $_____ to my □ Bankcard □ Visa □ Mastercard.

Account number: _. Expiry date: __/__.

Name on card: ______________________________ Signed: ____________________

Office use only:

Chq: bank ________ *bsb* ____-____ *a/c* ______________ *#* ______________

Date: __/__/__ *$* *CC type* ___ *V#* __________

Who: ____________ *Subscr#* ________

AUUG

Notification of Change of Address
Australian UNIX* systems Users' Group.

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

If you have changed your mailing address, please complete this form, and return it to:

AUUG Membership Secretary
P O Box 366
Kensington NSW 2033
Australia

Please allow at least 4 weeks for the change of address to take effect.

Old address (or attach a mailing label)

Name: .... Phone: .... (bh)

Address: .... (ah)

....

.... Net Address: ....

....

....

New address (leave unaltered details blank)

Name: .... Phone: .... (bh)

Address: .... (ah)

....

.... Net Address: ....

....

....

Office use only:

Date: ___/___/___

Who: ______________ *Memb#* __________