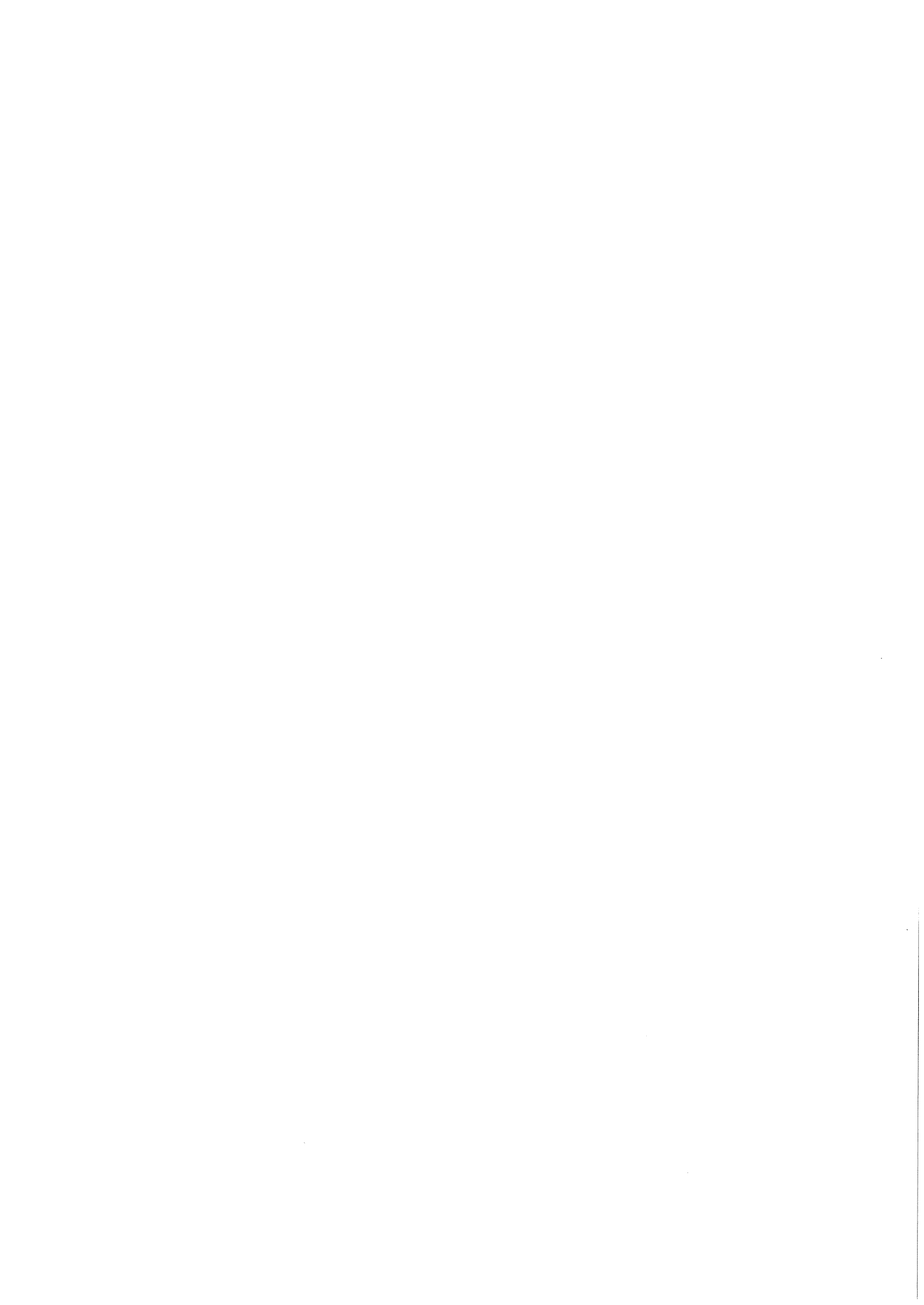


AUUGN

**Australian Unix systems
User Group Newsletter**

Volume 7

Number 6



The Australian UNIX* systems User Group Newsletter

Volume 7 Number 6

April 1987

CONTENTS

AUUG General Information	2
Editorial	3
AUUG Winter 1987 Meeting Announcement	4
Adelaide UNIX Users Group Information	5
Rock C Music	6
ASCII vs UNIX	8
Taking Performance Evaluation out of the "Stone" Age	12
Cake: a fifth generation version of make	22
UNIX Menu System	32
From the <i>login</i> : Newsletter - Volume 12 Number 2	44
MINIX: A UNIX Clone with Source Code for the IBM PC	45
The DASH Project: Design Issues for Very Large Distributed Systems	52
Book Review - The Nutshell Handbooks	54
General Meeting Minutes	58
Letters to the Editor	60
AUUG Membership Categories	79
AUUG Forms	81

Copyright © 1987. AUUGN is the journal of the Australian UNIX* systems User Group. Copying without fee is permitted provided that copies are not made or distributed for commercial advantage and credit to the source must be given. Abstracting with credit is permitted. No other reproduction is permitted without prior permission of the Australian UNIX systems User Group.

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

AUUG General Information

Memberships and Subscriptions

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

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

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

General Correspondence

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

The AUUG Secretary,
Department of Computer Science,
Melbourne University,
Parkville, Victoria 3052.
AUSTRALIA

ACSnet: auug@munnari.oz

AUUG Executive

Ken McDonell, *President*

kenj@moncsbruce.oz
Monash University, Victoria

(Temporary address is kjmcdonell@er.waterloo.cdn)
(University of Waterloo, Canada)

Robert Elz, *Secretary*

kre@munnari.oz
University of Melbourne, Victoria

Chris Maltby, *Treasurer*

chris@gris.oz
Softway Pty. Ltd., N.S.W.

Chris Campbell, *Committee Member*

chris@olisyd.oz
Olivetti Australia, N.S.W.

John Lions, *Committee Member*

johnl@elecvox.oz
University of New South Wales, N.S.W.

Tim Roper, *Committee Member*

timr@labtam.oz
Labtam Limited, Victoria

Lionel Singer, *Committee Member*

lionel@pta.oz
Lionel Singer Group, N.S.W.

(This does not work)

Next AUUG Meeting

The next meeting will be held at NSWIT on the 27th and 28th of August.
Further details are provided in this issue.

AUUG General Information

Editorial

It has become obvious to me that many people in the UNIX community are NOT aware of the User Group. This is especially true amongst commercial vendors and users of UNIX. I think the Group should try to improve the situation. On a personal level, tell people you know who use UNIX, but are not members, about the AUUG. Please show them a copy of this newsletter, and encourage them to join. The Group as a whole could promote itself more at wider forums such as the A.C.S., in the electronic news, and at commercial UNIX seminars.

People that do not know that we exist cannot possibly become members !!

Special thanks to those who contributed to this issue, without them this Newsletter would be very short on AUSTRALIAN content.

Please help the Newsletter by sending me a contribution.

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

AUUGN Correspondence

All correspondence regarding the AUUGN should be addressed to:-

John Carey
AUUGN Editor
Computer Centre
Monash University
Clayton, Victoria 3168
AUSTRALIA

ACSnet: augn@monu1.oz

Phone: +61 3 565 4754

Contributions

The Newsletter is published approximately every two months. The deadline for contributions for the next issue is Friday the 12th of June 1987.

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

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

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

Advertising

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

Mailing Lists

For the purchase of the AUUGN mailing list, please contact Chris Maltby.

Disclaimer

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

AUUG

Winter Meeting 1987

Sydney, August 27 and 28.

The Winter 1987 AUUG meeting will be held in Sydney, August 27 and 28 (Thursday/Friday) 1987.

The meeting is being hosted by the New South Wales Institute of Technology. Local conference organisation is under the direction of Greg Webb, gregw@nswitgould.oz.

More detailed information about this conference will be in the next issue of AUUGN, and posted to the newsgroup *aus.auug*.

We are now actively seeking papers for this conference. The programme committee chairman is Bob Kummerfeld from the University of Sydney.

Please send abstracts of papers to him at bob@basser.oz. Paper abstracts can be sent to

Dr R.J. Kummerfeld,
Basser Department of Computer Science,
University of Sydney,
NSW 2006,
Australia.

The deadline for abstracts is Jul 10 1987. Authors will be notified of acceptance by July 31.

Authors of papers given at the conference will receive complimentary admission to the conference dinner. Authors who provide a written version of their paper by August 21 will have the conference registration fee waived.

In addition, AUUG has decided to hold a competition for the best paper by a full time student at an Australian educational institution. The prize for this competition will be an expenses paid trip to the AUUG meeting to present the winning paper. Students should indicate with their abstract that they wish to enter the competition, and then should provide the full written paper to the programme committee (which will be the sole judge) by August 14. AUUG reserves the right to not award the prize if no entries of a suitable standard are forthcoming.

It is hoped that this will become a regular feature of AUUG conferences.

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

Rock C Music

Bruce Ellis
AT&T Bell Laboratories

There are some frightening parallels between the Rock Music industry and the Computer industry. A quick look at the local newsstand reveals startling similarities. Unix Review is a lot like Rolling Stone. A software review in the former reads much like a record review in the latter.

I was sitting in a bar on 8th Street, slowly sipping a Long Island Iced Tea. To the left of me were three members of a Heavy Metal band, to the right two guys talking about the financial modelling package they were working on. The conversations were very similar. The guys to the left were priming the jukebox with Hendrix and grunting bass lines to each other. The guys on the right were playing the Doors on the box and sketching screens and menus on the coasters. The most apparent difference between the two groups was the money. The guys on the left were not short of smart leather jackets but they were scrounging to pay for the next round of Bacardi and Cokes. The guys on the right did not have this problem. They would nod at the barman mid sentence for their refills. They lived in Manhattan. I'm sure the guys on the left lived in Queens or Brooklyn, or even (God forbid it) New Jersey.

When we move on to the field of electronic musical instruments the parallel lines meet. Fortunately a standard asynchronous information transfer protocol (MIDI) is well established and more or less demanded for every product. MIDI interfaces are available for many types of PCs and lots of software, both good and bad, is available and heavily pirated. This field is a micro hackers dream and the musical instrument shows have turned into engineering shows. Don't try and use any musical term at these unless you are willing to back it up with a distributors name. Where do you buy your leading notes?

Lets look at the end users. Last Tuesday night I was wiling away my time at a bar in the Village, as is my custom. I sipped my Beer as I watched the next band set up. "Do you have a cable with an XLR female on one end and a quarter inch male on the other?" Yes, but can I plug it into my Unibus? Finally the band is ready. The guy "doing" the PA hasn't a clue what he's doing, but he's merely support staff and has a long tradition of performing busy meaningless tasks to uphold. The users have brought their own peripherals, so there is little he can do about security. He can't really call a meeting in the middle of a song to discuss use of the midrange horns nor is he likely to rewire the stage mid set so I guess the analogy falls down.

Anyway. Back to the band. They're doing a plausible job of actually playing music and the crowd is content. There are bugs, a bit of feedback now and then, but the system seems to be holding up. I ponder a while at the little box sitting on top of the sythesizer, the one connected in-to-out and out-to-in to the synth. Another drink,

another XTC clone and I realise. It is a MIDI-controlled synth module slaved to the keeb. Only one of the MIDI chords is actually carrying data! What the hell? It works so why bother telling the user he doesn't need all that hardware? Sell him some more. Speaking of hardware the more-than-competent drummer has two bass drums! A backup? Surely not. At the end of the second set the band did the usual "introduce the band and see how long they can solo for" song. The second bass drum was for the "Flashy Demo". Sign him up to sell laser printers.

Let's look at the industry watchers. After the Slayer concert at the Ritz the best I could get out of the Black-Shirts was "They're really good!", hardly a compelling product endorsement. What does it matter? They picked a good name. A couple of thousand Black-Shirts belting across town to the PATH, chanting "Slayer!" and breaking Budweiser bottles is as good a promo as they'll ever need. And they put on a great show. It is a bit sad that the best thing that can be said about a lot of bands is "What a nice guitar Jimmy has". But this is what the industry watchers do. This is done all the time in the Computer industry, particularly with words like UNIX and C (I guess you were wondering what this has to do with AUUGN). Who cares about the prestige names: the Marshall's the Strats the 386s? What's is being done with the gear, what's coming out the other end? Don't tell me about your new machine, I'm quite happy using a VAX and never want to have to port another program in all my life. Anyway, I like those bright red angular guitars that sound like banjos and don't stay in tune.

So what can we conclude from all this? Buy Bon Jovi Computers I guess, and give SUN a bad name.

ASCII vs UNIX†

Bob Buckley

School of Mathematics, Physics, Computing and Electronics,
Macquarie University,
Sydney, NSW 2109.
bob@mqcomp.oz

Introduction.

The UNIX System has been with us for a while now and some of its initial advantages seem to be fraying with time. This is an attempt to indicate a few places where repairs are needed.

The motivation for this comes from a persistent problem of inexperienced users and their difficulties with printers and printer support software. However, the problem spills over into a number of related areas. We seem to have a number of models for ASCII files coexisting (and undocumented) in the UNIX environment.

I'm worried about file types. This is a naughty thing to say since there are no file types in this context. We all know that a UNIX file is just a string of 8-bit quantities. This means that we can easily copy, concatenate or do other simple things to files. Though this seems like a good idea, it turns out to be a too simple to be completely practical.

This article is more a warning than a report. It was presented (before being written) at the Adelaide AUUG meeting. It is less ready for printing than I would like.

ASCII interpretation.

Most of us feel we understand use of ASCII on UNIX. However, there are several issues which are not generally addressed.

UNIX likes to remove CR characters from its input. This simplifies recognition of line terminators. It gives an excuse for pretending the CR character doesn't exist. Unfortunately, this ostrich attitude doesn't always work. More and more, files and software is being imported to the UNIX context. Some data and software prefers to use CR to achieve overprinting while most UNIX software uses BS. Typically, CR isn't correctly managed by UNIX utilities. How many programs reset their column counters when they see CR? *Pr* doesn't tackle the problem (it is tricky - in multicolumn output, does CR mean CR or 'start-of-column').

Using CR for overprinting raises another question. Should we interpret spaces as being destructive or non-destructive? At the moment, for physical reasons, we have a mixture: terminals tend to assume destructive spaces while printers have non-destructive spaces. We seem to get by, but sometimes it really matters. It would be nice to have a consistent model.

Another problem character is FF. Mostly, this is ignored by UNIX but sometimes it causes trouble. It didn't used to be there but later versions of *troff* seem to be aware of page boundaries. Also, BSD *pr* will insert FFs (but seems to not like them in the input). The problem here is that the model doesn't determine whether FF moves to column zero (output devices vary on this). Again, *pr* has a problem with this character. Should FF start a new column or a new page?

† UNIX is a trademark of Bell Laboratories.

At the moment, vertical motions are not consistent. NL is generally assumed to move to column zero (except by *col*) while other motions (ESC 7, 8 or 9 and VT don't). Much of the rest of the ASCII control codes are ignored except by the tty drivers (and that's a whole new ball-park). UNIX seems to manage TABs pretty well.

UNIX file types.

There have always been a few special UNIX file types. The first obvious example is a directory. There are many restrictions on what can be done to a directory by a user program. There are some special programs which deal with directories, eg. *ls*, *mv*, *cp*, ... There are a few restrictions, like not *write(2)*ing.

Other special file types are the *special files*. These allow different operations (via *ioctl(2)*) and are really special. Pipes, sockets, etc. are different again.

Another group of file types known to the kernel is the executable files. An executable file can be *exec(2)*ed - others can't. The kernel knows they are executable because they contain type information.

UNIX is full of file types when you consider the utilities. Look at the manual description of file formats (it used to be section 5); as well as *a.out(5)*, you will find things like *ar(5)*, *tar(5)*, *dump(5)*, *wtmp(5)* and *utmp(5)*. All of these are file types.

This is OK. The UNIX kernel doesn't know (or care) about these. It is all up to the programmers responsible for the system software. We just need to be a little more honest: UNIX does have lots of file types but the kernel doesn't care. The utilities care.

Implicit File Types - the Problem.

UNIX has many implicit file types. Further, these types aren't always documented and people aren't conscious of their existence. Many programs are unexpressive of their limitations.

Several programs (eg. *ed*, *vi* and *pr*) expect ASCII files ie. files containing spaces, printable characters and NL (provided NLs aren't far apart). A few other characters are acceptable, in particular TAB and BS. There are common ASCII characters that are not generally acceptable (eg. CR and FF). Such files present few problems provided lines don't get too long.

Actually, such files have different types. C source, Pascal source and English text are different types. Of course, the UNIX kernel doesn't care but other software does, eg. the C compiler. The *file(1)* utility attempts to recognise the difference.

There are other anonymous file types managed by UNIX utilities. *Nroff* is good at producing special file types - by default it produces files intended to be sent, in raw mode, to a TTY37 (whatever that was). Control characters have different meanings (particularly LF). Extra characters may be used, particularly CR, SI, SO, VT and ESC (with 7,8 or 9). The interesting thing is that *nroff*'s TTY37 output can be used as input to other programs, particularly *col* and *pr* (*ul* and *more* on BSD systems) Then, with luck, *col*'s output may be suitable input for *pr*!

Nroff -Txxx produces a different file type and its output should be sent directly to a *xxx* device. Though such a file contains only ASCII characters, it is not UNIX's idea of an ASCII file (despite what *file(1)* says). A related problem arises with *graph(1)* or *plot(3)* output. Due to poor support and limited/infrequent use this presents fewer problems, in practise.

We pretend there aren't file types. For some utilities this is true. Programs like *cp*, *od*, *tr*, etc. genuinely don't care about their input.

The real problem is that there are lots of different file types and there are lots of ways of describing them. What determines a file type? Is it determined by the programs or devices which can process it? Is it determined by its contents or its intended interpretation? Each approach has problems.

TTY37 files are a sort of intermediate form and have a file type which is partially documented and supported. Output for other printer types is not generally supported and file types are not defined. The problem is generally regarded as 'too difficult'. As a result, users are expected to know a file's type and to process it accordingly. Many utilities could object to nonsense input.

What to do.

There are several things that need fixing now. These are minor but need to be fixed consistently. *File(1)* utility is a rarely used and usually out-of-date program. It needs updating and a bit of promotion. Several programs need to improve their image. *Pr(1)* should complain about CR and FF with multi-column output. (What do CR and FF mean in multi-column output?)

The TTY37 model of ASCII is broader than we generally need. It is broader than we can easily handle (few printers handle reverse motions or special characters, some don't backspace and a few can't even overprint). There are probably three major sub-types. The simplest is a basic file for editing. Such a file contains only printing characters (040-0176), TABs and NL. Most source code, data and *nroff*/*troff* input is like this. This material can be easily manipulated - edited, printed, etc.

After this comes files directed to simple printers. As well as the above, they may include overprinting via BS (perhaps FF should also be allowed). A program like *pr* should be able to handle this as input. Character widths are fixed. Font handling for normal, bold, italics and underline should be included (note: bold and underline are currently expressed as overprinting but it should be explicit, especially now that continuous underlining is supported). Programs which understand BS should allow overprinting using CR (this is pretty easy and available on other systems).

The TTY37 model allows SI/SO special character handling. This can be merged with the bold/italic font handling above.

The TTY37 model supports reverse motions and half-line motions. Note: reverse motions can be eliminated using *col(1)* so only half-line motions are necessary.

Other *nroff* output is aimed at specific devices. This should be explicitly indicated so *lpr*, etc. can check that output is going to appropriate devices. Other checks are also needed - eg. no printing of executable (non-ASCII) files unless these are some sort of download file for the device. *Col* and other utilities should complain about unsuitable input. Files requiring particular hardware (fancy fonts, fine motions, proportional spacing, etc.) should be made explicit. With so many (nearly) compatible printers this gets complicated. Checking to see that output is suitable for a particular printer is a real headache - but we need to face this problem. There are problems ensuring fonts are properly downloaded and that the printer is in the right mode, etc.

If we consider files intended for bit-mapped screens or 'intelligent' terminals, the problem gets worse.

This boils down to a need for explicit file types in the UNIX environment. Simple ASCII (editable files) are probably a basic type. It is usually easy to see what these are about. Even so, it doesn't make much sense to *cat* C source code and *nroff* input together (just as it isn't sensible to append a file to an archive). Merging an ASCII file with a printer file may be sensible but it could require significant work (is this what desk-top publishing is about). Like TTY37 files, PostScript and *plot(5)* files are intermediate forms. Should we expect to be able to combine these file types? This is the stuff that dreams are made of.

More needs to be said about file types. Documentation should be fixed up and most references to TTY37 should be deleted. People tend to ignore stuff about devices they don't have.

A rather specific problem is that output programs should handle file types automatically. Conversions for italics, bold, underline, etc. should be standard for all devices. Ideally, this is done as late as possible - perhaps in the device driver (after all, tty drivers do just about everything else) or in the very last program. When this isn't possible, warnings could be posted. A major contender for improvement is the *lpr/lpd* software (consider */usr/games/worms* piped into *lpr*). Surely, Berkeley's *printcap* (and *termcap*, for that matter) needs conversions for TTY37 oddities (reverse motions, half-motions, etc.).

As a final remark, obtaining device specifications from the environment doesn't quite work. It fails for most redirected output - output piped to *lpr* or directed to another terminal. Networks can make the problem even worse. This is not an easy problem to solve.

Conclusion

The UNIX model of ASCII files was based around the TTY37. This terminal has departed (or was never here) but its memory is enshrined in the UNIX environment. It served us well but the advent of 'advanced' printers and displays show the need for change. If this is mismanaged, we may never recover.

In most cases, the utilities are the problem. The above recommendations hardly effect the UNIX kernel. The utilities have been derived from a large user base. It is difficult to maintain a consistent model over the time they've been developing and such a large development team. There might be fewer problems if the model were more explicit.

You may feel that this approach is totally (or partially) wrong. This article will have served it's purpose if you think about it.

TAKING PERFORMANCE EVALUATION OUT OF THE "STONE" AGE *

Ken J. McDonell[†]
Department of Computer Science
Monash University
AUSTRALIA
kenj@moncskermit.oz

ABSTRACT

Predicting the performance of any computer system is critical to rational equipment acquisition by purchasers and successful product delivery by vendors. This paper takes a brief but critical look at "figure of merit" performance metrics, especially with respect to their reliability and predictive usefulness.

Necessary prerequisites for serious performance evaluation and prediction are identified. The architecture of the MUSBUS benchmark suite is described with particular emphasis on the technical considerations that allow MUSBUS to be tailored to accurately predict multi-user system behaviour in specific operational environments.

1. Performance Evaluation Goals

All computer system performance evaluation is directed to one or more of the following specific goals,

- (G-1) Compare the **measured** performance of heterogeneous systems executing specific identical tasks.
- (G-2) Compare the **anticipated** performance of heterogeneous systems executing the same "typical" tasks.
- (G-3) Collect diagnostic evidence to substantiate hypotheses about anomalous performance (either very good or very bad) for one or more tasks executed on a particular stable system.

Irrespective of the specific tests employed most performance metrics are based upon resource consumption (e.g. cpu time), throughput (e.g. as measured by elapsed time) or some related measure (e.g. number of users supported or mega-grunts per second).

Goal G-2 typically implies performance *prediction* based upon performance *measurement*, and this is by far the most useful application of performance evaluation. Purchasers want reliable estimates of expected system performance in *their* anticipated operational environment. This would be useful not only when new systems are being acquired, but also when upgrades are considered, e.g. "what demonstrable improvement can we expect from adding 2Mbytes of memory or a second disk controller?". On the other hand, vendors need reliable estimates of system performance across a range of designated application environments to guide marketing and system tuning activities.

The central thesis of the first half of this paper is that G-2 is the most widely sought, but seldom realized, goal of current performance evaluation activity in the UNIX‡ community. An approach to achieving this goal in any operational environment is described in the later sections.

2. Some Performance Tests and Measures

The UNIX system and the C programming language combine to provide a stable software execution environment on machines across the full price-performance spectrum. The consequent ease with which portable software can be developed has fostered a large class of tests purporting to measure system performance by a single "figure of merit" metric. These tests fall into two basic classes,

* This paper will be presented by Ken at the next Usenix meeting, Phoenix, Arizona, June 1987 and will be printed in the proceedings - AUUGN Editor.

† Currently on leave at the Department of Computer Science, University of Waterloo, Ontario, CANADA, kjmcdonell@er.waterloo.cdn or kjmcdonell@waterloo.csnet

‡ UNIX is a Registered Trademark of AT&T.

- (a) The “one liners” that are easy to type in, use standard UNIX programs and measure cpu time using either the shell built-in time function or /bin/time. Some common examples are shown in Figures 1 and 2.
- (b) Synthetic tests such as the “stone” family (whet[3], dhry[9, 12], dhamp[5], ...).

Irrespective of which class they come from, these tests may be characterized as follows,

- (a) The test is easy to run, and is guaranteed to produce a value for the performance metric.
- (b) The value obtained is statistically unreliable due to an unknown and often large measurement error; worse still, the relative error may vary between different machines.
- (c) The performance metric typically measures some combination of
 - cpu arithmetic speed, and
 - C compiler quality.

Unfortunately, other important factors (as identified below) are totally ignored.

- (d) System performance under *real* operating loads may be, but most often is not, well correlated with the test and performance metric[6].

Clearly these tests potentially satisfy the performance evaluation goal G-1, and with careful use could assist with goal G-3. However the majority of people running and interpreting these tests appear to believe the results have some predictive value as per goal G-2. The compilation and publication of tabulated results from these tests under the heading of “UNIX benchmark results” is highly misleading because what has been measured is influenced by only a few of the many factors contributing to system performance – this is the fundamental weakness common to all the single “figure of merit” approaches. These test programs and performance evaluation suites are essentially of no more use than common sense and guesswork in predicting the performance of a *real* system under *actual* load conditions.

To be fair, it is the use of the tests and interpretation of the results that is most commonly at fault. The tests have often been constructed for a specific purpose, and in that role are both accurate and useful. Despite the pleas of their creators (see for example[8, 10, 11]), and caveats in the code, it is the adoption of the test for an unsuited role (i.e. general performance prediction) that is the problem.

Another class of tests has evolved from recognition of specific areas of weakness in some UNIX implementations and/or factors perceived to be important influences on performance, for example

- filesystem throughput
- system call overhead

```
main()
{
    int    i;
    for (i = 0; i < 1000000; i++)
        ;
}
```

Figure 1: The “count to a million” test.

```
$ time dc
99k2vpq
1.414213562373095048801688724209698078569671875376948073176679737990732\
478462107038850387534327641572
          9.2 real          1.3 user          0.2 sys
```

Figure 2: The “square root of 2 to 99 decimal places” test.

- fork() and exec() speed
- pipe throughput
- memory access bandwidth

These tests are clearly aimed at goal G-3, and any wider interpretation of their results cannot be made. Even in this restricted usage, these tests can be tricked by implementors aiming for a competitive edge in commonly used benchmarks (e.g. cache the results from getpid()). In some cases the tests are simply misguided, measuring behaviour that is uncommon in many operational UNIX environments (e.g. random file I/O).

Some attempts have been made to provide test environments in which many G-3 style tests are performed, and the results combined using relative weights of importance [1,4,7]. The disadvantages of this approach are,

- (a) the tests are not statistically independent (interaction between factors is not measured), and
- (b) accurate assignment of the weights of importance is more difficult than constructing a user-level workload profile as suggested below.

Finally, there have been some G-1 type tests developed for comparing heterogeneous systems executing the same set of end-user tasks, for example [1,2]. The predictive value of these tests depends upon the extent to which the supplied end-user tasks are representative of a particular operational environment.

3. Improving the Methodology

In the performance of various UNIX systems running the same task was accurately measured, the differences in the observed results may be attributed to some of the following factors,

- processor performance; includes raw speed, configuration options (e.g. FPU, data cache, co-processor) and interrupt servicing overheads
- disk subsystem performance; device characteristics, bus bandwidth and channel/controller configuration
- C compiler; the quality may vary dramatically with revision level
- UNIX kernel implementation; quality varies between base versions, ports and release levels
- filesystem configuration; allocation of filesystems across spindles and filesystem age
- real memory available to user processes
- configuration parameters; most notably disk buffer cache size and filesystem block size(s)

Reliable performance evaluation tests must be sensitive to all the above factors, because the performance delivered to the end-user can be seriously degraded by any one of these factors. The simplest test meeting this criterion is to measure the time required to perform some mixture of typical processing tasks from the anticipated operational environment. In this way, total system performance is measured directly, rather than attempting to isolate and measure the performance in each of the critical areas.

Whilst there undoubtedly exist classes of UNIX users with similar patterns of system usage, it is unrealistic to expect that one test or one mix of processing tasks will be representative for all operational environments. Rather, we should be aiming for tools that help us create, realistically execute and instrument the running of a representative collection of tasks on a variety of system configurations.

Serious performance evaluation requires,

- (a) Definition of the anticipated workload profile.
- (b) A test environment that will execute randomized tasks, chosen from the desired workload profile, for various levels of system load and record statistically sound measures of resource consumption. This test environment must be portable and extremely robust to maximize its usefulness and to encourage vendors to run user-specific benchmark tests, often at remote locations.
- (c) Careful documentation of the environment in which the test was conducted (hardware configuration, revision levels of the operating system and C compiler, workload profile, sysgen configuration parameters, filesystem partitioning, etc.).

A workload profile may be characterized by a set of independent user-level tasks, each typically corresponding to one or more program executions. For each task, the following information is required,

- the particular programs involved
- representative test data (data files, user input, patterns to search for, directory contents, etc.)
- relative frequency of execution

Identifying and describing the anticipated workload profile is a task of varying difficulty. In some environments, historical records (e.g. process or shell accounting) or known application usage provide accurate data from which the workload profile may be constructed. In other cases, informed guesswork is required.

For the MUSBUS multi-user test described below, a workload profile consists of an annotated shell¹ script with all associated data files. Tasks with high relative frequencies may appear more than once in the script.

4. MUSBUS

The Monash University Suite for Benchmarking UNIX Systems (MUSBUS) is a public-domain benchmark suite developed originally for equipment comparison during acquisition procedures.

The suite supports all three goals of performance evaluation with a simulated multi-user testbed facility and a battery of specific diagnostic tests.

The diagnostic tests have been designed to measure raw speed in very specific areas. Their execution is controlled by a shell script and parameterized so that the default values effecting test selection, size and duration may be overridden by command line options and environment variables. Table 1 provides a brief summary of these tests.

Of all the tests in MUSBUS, the simulated multi-user test is the by far the most complicated, most realistic and most likely to uncover operating system bugs. It is also the test specifically engineered to provide reliable predictions of anticipated performance (goal G-2) since it may be easily configured to perform “typical” tasks for any operational environment and then run on heterogeneous systems.

Once a workload profile has been defined (as described in the previous section), several (typically 4) scripts are automatically created, each comprising a randomized permutation of all the tasks in the workload profile. A control file (*workload*) is also created to describe how each script should be run (refer to Figure 3).

The multi-user test simulates a variable number of users, each executing their own job stream. The job streams are chosen by cyclic selection from the scripts.

Control over the multi-user test rests with the program *makework* (refer to Figure 4) that performs the following functions.

- (a) Read the workload and script files into dynamically allocated buffers.
- (b) Make cloned copies of itself (via `fork()`) to run the job streams for groups of users (necessary due to per process open file limits).
- (c) Start each user shell with its input coming from *makework* via a pipe.
- (d) Send random chunks of input to the job streams, controlled so that the aggregate rate across all simulated users does not exceed a specified rate in characters per second.
- (e) All output from the shells and echoing of all input is directed to one or more real tty devices to ensure that an appropriate number tty output interrupts occur. See Figure 5.
- (f) When all script input has been sent, wait for all user shells and *makework* clones to terminate.

All MUSBUS tests are run under the control of a large Bourne shell procedure charged with.

- (a) Executing each test several times (the default is 6 or 3, depending on the particular test), recording the `/bin/time` results then computing the mean and standard deviation of the total (user plus system) cpu and elapsed times.

¹ The choice of “shell” is truly arbitrary, and may include any interactive application environment, e.g. an SQL database query language interface.

Test Name	Controlling Variables and Default Values	Description
arith	arithloop [1000]	A family of tests that compute the sum of a series of terms such that the arithmetic is unbiased towards operator type. Each major loop in the computation involves summing 100 terms; there are \$arithloop major loops. Repeated for all flavours of ints and floats.
dc		Compute the square root of 2 to 99 decimal places using <i>dc</i> . This test is due to John Lions (University of New South Wales) who has suggested it as a good first order measure of raw processor speed.
hanoi	ndisk [17]	A recursive solution to the classical Tower of Hanoi problem. \$ndisk provides a <i>list</i> of the number of disks for a set of problems.
syscall	ncall [4000]	Sit in a hard loop of \$ncall iterations, making 5 system calls (dup(0), close(i), getpid(), getuid() and umask(i)) per iteration.
pipe	io [2048]	One process (therefore no context switching) that writes and reads a 512 byte block along a pipe \$io times.
spawn	children [100]	Simply repeat \$children times; fork a copy of yourself and wait for the child process to exit.
execl	nexecl [100]	Perform \$nexecl execs using execl(). The program to be exec'd has been artificially expanded to a reasonable size.
context	switch [500]	Perform 2 x \$switch context switches, using pipes for synchronization. The test involves 2 processes connected via 2 pipes. One process writes then reads a 4-byte (descending) sequence number, while the other process reads then writes a sequence number.
C		Measure the time for each of <pre>cc -c cctest.c</pre> and <pre>cc cctest.o</pre> where cctest.c contains 124 lines of uninteresting C code (108 lines of real code after <i>cpp</i>).
seqmem	poke [100000] arrays [8 64 512]	These tests try to measure memory read accesses per real second. \$poke accesses are made into arrays of \$poke x 1024 ints. A cyclic sequential access pattern is used.
randmem		Like seqmem, but uses random access patterns.
fstime	blocks [62 125 250 500] where [.]	Sequential file write time, file read time and file copy time for files of \$blocks Kbytes. Temporary files will be created in the directory \$where. The <i>copy</i> time for the larger files is the best indicator of throughput and reflects the type of disk activity most commonly generated by compilers, editors, assemblers, etc.

Table 1: MUSBUS diagnostic tests.

```
/bin/sh -ie <script.1
/bin/sh -ie <script.2
/bin/sh -ie <script.3
/bin/sh -ie <script.4
```

Figure 3: Typical specifications for executing scripts (workload).

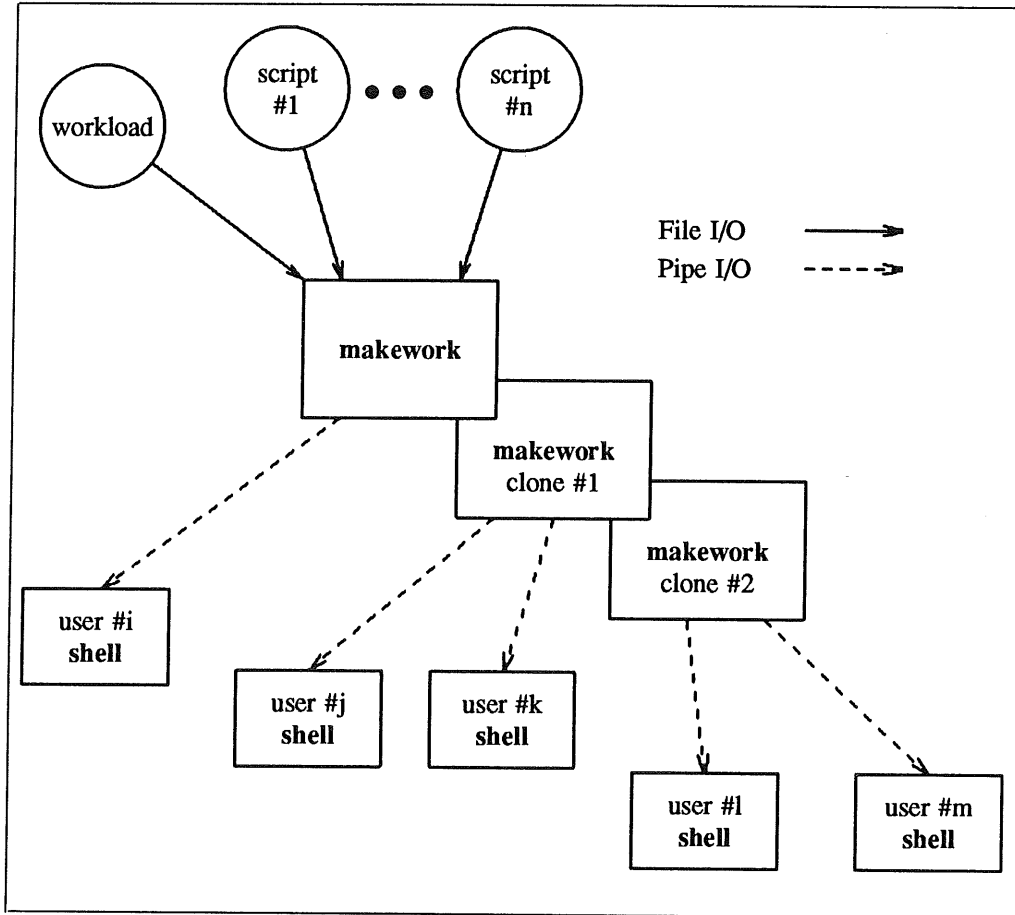


Figure 4: Overall architecture of the MUSBUS multi-user test.

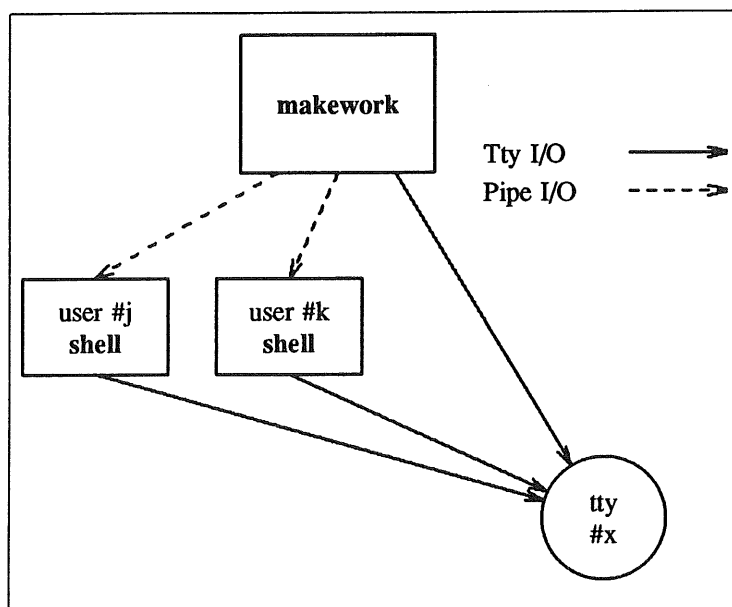


Figure 5: Directing terminal output to a physical device.

- (b) Reconfiguring the multi-user tests to allow tty and filesystem activity to be distributed across an arbitrary number of physical devices.
- (c) Performing tests with different control parameters, e.g. varying the number of job streams in the simulated multi-user test.
- (d) Monitoring completion status and standard error output to detect failed tests.

5. Issues Related to Test Engineering

The development of the MUSBUS multi-user test in particular has highlighted a number of issues related to benchmark test design in the UNIX environment.

Quiescent system configuration. Some tests must be run as super-user to avoid per user limits (e.g. maximum number of processes). However, given the performance prediction goals, the tests should be run on an otherwise unloaded machine in **multi-user** mode. This ensures that mandatory daemon activity will be present during test measurements.

Interactive input rate limitation. Limiting the *rate* at which input is presented to the shell and other interactive programs is an important factor influencing the predictive accuracy of the multi-user test results. Without this constraint, an interactive program's contribution to resource consumption for the job stream may be significantly reduced (e.g. artificially short program residency leads to higher buffer cache hits rates during `exec()` and application file I/O for repeated program executions, and reduced swapping and/or paging activity). Of course the ideal situation would be to simulate demand driven (i.e. no typeahead) and rate limited input. Unfortunately there is no portable and cheap (in terms of resource consumption) software technique² for one process to determine that another process is waiting for input, and so simulating demand driven input is not possible.

Bogus file sharing. If tasks in the job streams require access to the same data file, private copies should be made unless the files are truly shared in the application environment, otherwise buffer cache hits will artificially reduce the cost of file I/O. For example, if all *N* job streams contain an edit task on a sample data file, there should be *N* copies of the file made, one per simulated user.

Multiplexed standard input. When two processes compete in time for the one source of standard input (see Figure 6) serious problems may arise if the input generator (i.e. *makework*) is not response-driven. In general *makework*

² Although external hardware "stimulators" have been used in some cases.

cannot tell whether the current “chunk” of input text is intended for the user’s shell or some program invoked from that shell or a mixture of both. With reference to Figure 6 there are many pathological situations, the worst being program K consumes in one read some input that includes it’s own termination command and some of the following text intended for the shell once program K has finished – the shell never sees that text! The architecture shown in Figure 7 has been used to overcome this; *keyb* includes the rate-limited text generation algorithm from *makework* and allows separation of shell and application input. However to retain control over the shell’s rate of execution, the shell script must be padded with comments by the number of bytes in the input stream to program K.

Testing for failure. *Makework* checks the results of every system call, has a SIGPIPE handler and checks the status returned via wait(). If any error is detected, *makework* kills off all shells and the *makework* master kills off all clones (and their dependent shells). There is a certain degree of paranoia in this error checking, fostered by several bad experiences in which bizarre UNIX implementation bugs resulted in very good, but incorrect, predicted performance (if

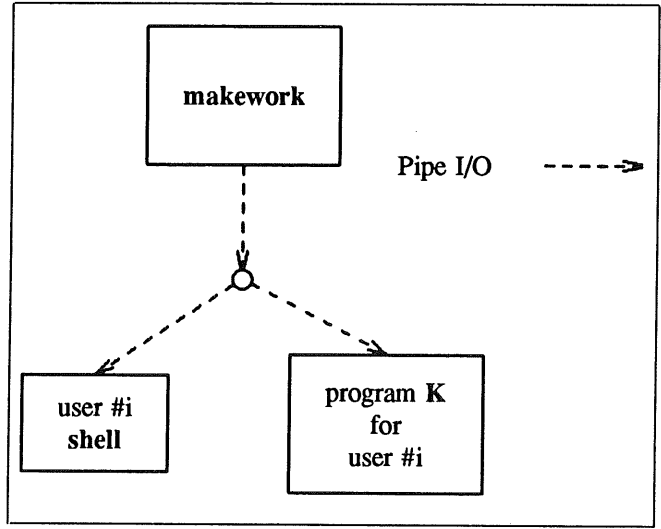


Figure 6: Multiplexing standard input in a job stream.

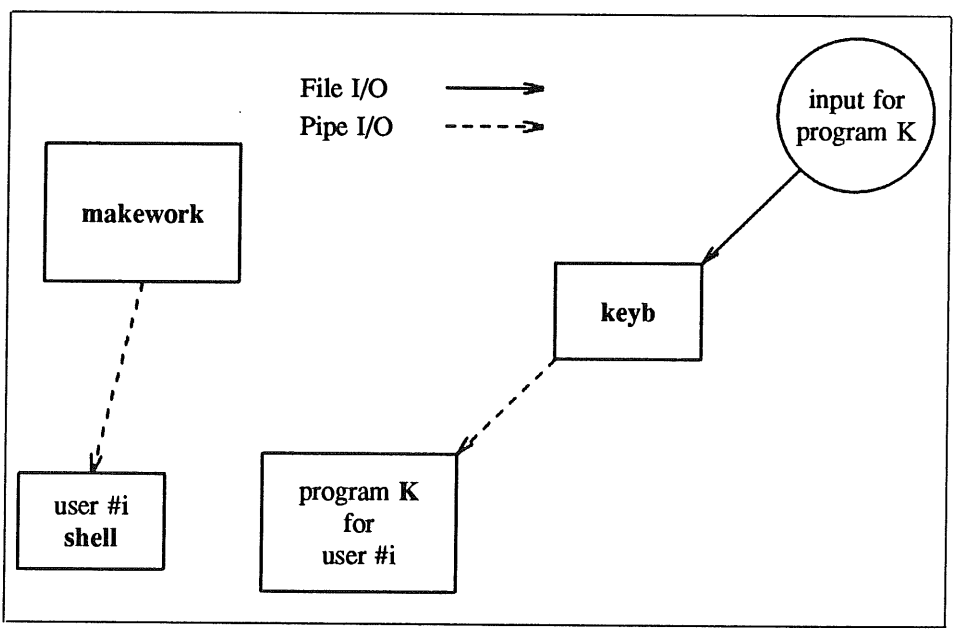


Figure 7: Multiple input sources for a job stream.

only a fraction of each job stream is executed, the work can be completed in a very short time!). No program or system call can be assumed to always execute correctly.

Randomizing the processing. Particularly misleading results are produced when a number of identical job streams are executed in effective synchrony. In some other benchmark suites, this has been used as a cheap way of increasing the "work" performed in a test. MUSBUS randomizes the processing load by using permuted scripts and randomizing the input rates to individual shells.

Using standard tools. MUSBUS uses many standard UNIX tools and utilities, in particular the Bourne shell, *awk*, *sed*, *grep* and a particularly bland vanilla dialect of C that is very portable. Uses include,

- permuting tasks to construct scripts
- checking standard error output for unexpected messages
- producing statistical summaries of repeated test results
- post-processing results to automatically produce *tbl* input for summary tables and tables comparing system performance
- the driving script, controlled by command line arguments and environment variables.

Constructing portable software. Since MUSBUS was specifically designed for comparing performance between heterogeneous UNIX systems, software portability was always an issue of importance. Despite the apparent uniformity of the C and UNIX interfaces, and considerable prior experience in building portable systems, a number of hidden incompatibilities were revealed in early MUSBUS usage. Some of the more notable problems included,

- */bin/time* produces different format output, which means different *awk* scripts are required to produce the statistical summaries.
- The behaviour of *wc* when given a single argument is not consistent.
- Trying to measure short elapsed time intervals varies between, impossible, hopelessly unreliable and grossly obscene code.
- The total lack of standardization in *cpp* predefined macros to reflect environment parameters (cpu and UNIX flavour) led to the creation of yet another redundant set of *cpp* macros that must be checked by hand before the software can be installed.

6. Performance Prediction

All MUSBUS performance predictions are based upon the multi-user test results for a varying number of job streams. Provided sufficient data points have been collected (4 or more) over a range of "number of job streams" that moves out of the linear region of performance behaviour, reasonable extrapolated results can be obtained for each of the measures described below.

System throughput: the elapsed time for a particular number of job streams.

System saturation: can be deduced from the ratio of cpu to elapsed times.

Relative response time degradation: can be determined directly from the ratio of elapsed times for 1 and N job streams.

These predictive measures may be extrapolated to answer the following sorts of questions.

- With 48 simulated users, which system offers best throughput?
- Which system supports most simulated users at 0.85 cpu utilization?
- Which system supports most simulated users at the point where response times have deteriorated by 50% over the single user performance?

Accurate interpretation of measured performance requires considerable skill and awareness of factors such as the following.

- (a) Particular hardware configurations, versions of the same Unix port and C compilers vary with time to such an extent that labelling one set of figures as from Brand X Model Y is misleading to all concerned.

- (b) MUSBUS is intended to be reconfigured in the multi-user simulated workload test to reflect the work profile of a particular user site. Whenever different workloads are used the results cannot be compared.
- (c) Deliberately the MUSBUS tests are in two distinct categories, raw speed and multi-user. The former are useful for diagnostic purposes only and give little useful information for a potential purchaser. The latter test gives good predictions of system performance.
- (d) Changing Unix configuration parameters (e.g. cache size, filesystem architecture, filesystem age, etc.) may have dramatic effects on the observed performance.
- (e) Beware of simulating **too few** users in the multi-user test. Useful information about system throughput and performance under heavy load conditions can usually be obtained by extrapolation of various measures computed from the CPU and elapsed times for the multi-user tests with various numbers of users. However this assumes the machine has been sufficiently loaded to move out of the *linear* part of the performance curves. For very fast machines, this may require emulation of a *large* number of users in the multi-user test.
- (f) Beware of simulating **too many** users in the multi-user test. This can result in unexpected resource depletion (e.g. serial line bandwidth) that does not accurately reflect the likely operating conditions.
- (g) Serious testing has been known to “break” UNIX ports. Causes have been identified as implementation (configuration) limits in the system being tested (e.g. proc slots), real bugs in the port or MUSBUS errors.

7. Concluding Comments

MUSBUS was originally developed to assist in equipment selection decisions. In that role it has proven to be most useful, and by empirical standards, an accurate predictive tool.

However the use has grown to include technical performance criteria to be met in contractual acceptance conditions, system check-out during installation, in-house performance measurement and kernel-exercising by several vendors during product evolution.

References

1. AIM Technology, AIM Benchmark Suite II – Evaluating UNIX Computers , 1984.
2. L. F. Cabrera, Benchmarking Unix – A Comparative Study, in *Experimental Computer Performance Evaluation* , D. Ferrari and M. Spadoni, (eds.), North-Holland, Amsterdam, 1981.
3. H. J. Curnow and B. A. Wichmann, A Synthetic Benchmark, *Comp. J.* 19 , 1, (Feb. 1976), 43-49.
4. G. Dronek, Relating Benchmarks to Performance Projections, *Proc. USENIX*, Salt Lake City, Utah, Jun., 1984.
5. R. J. Eickemeyer and J. H. Patel, Dhampstone, USENET, Newsgroup comp.arch, Article <500001@uicsg>, 14 Feb., 1987.
6. J. Mashey, Re: 01/31/87 Dhrystone Results and Source, USENET, Newsgroup comp.arch, Article <112@winchester.mips.uucp>, 9 Feb., 1987.
7. M. F. Morris and P. F. Roth, *Computer Performance Evaluation – Tools and Techniques for Effective Analysis*, Van Nostrand Reinhold, New York, 1982.
8. J. H. Patel, Re: Dhrystone and Dhampstone, USENET, Newsgroup comp.arch, Article <500002@uicsg>, 26 Feb., 1987.
9. R. Richardson, Dhrystone, USENET, Newsgroup comp.arch, Article <153@homxb.uucp>, 14 Mar., 1987.
10. R. Richardson, Re: 01/31/87 Dhrystone Results and Source, USENET, Newsgroup comp.arch, Article <2366@homxb.uucp>, 7 Feb., 1987.
11. R. Richardson, Re: 01/31/87 Dhrystone Results and Source, USENET, Newsgroup comp.arch, Article <2387@homxb.uucp>, 14 Feb., 1987.
12. R. P. Weicker, Dhrystone: A Synthetic Systems Programming Benchmark, *Comm. ACM* 27 , 10, (Oct. 1984), 1013-1030.

Cake: a fifth generation version of make

Zoltan Somogyi
Department of Computer Science
University of Melbourne
zs@mulga.OZ

Abstract

Make is a standard Unix¹ utility for maintaining computer programs. Cake is a rewrite of make from the ground up. The main difference is one of attitude: cake is considerably more general and flexible, and can be extended and customized to a much greater extent. It is applicable to a wide range of domains, not just program development.

1. Introduction

The Unix utility make (Feldman, 79) was written to automate the compilation and recompilation of C programs. People have found make so successful in this domain that they do not wish to be without its services even when they are working in other domains. Since make was not designed with these domains in mind (some of which, e.g. VLSI design, did not even exist when make was written), this causes problems and complaints. Nevertheless, implied in these complaints is an enormous compliment to the designers of make; one does not hear many grumbles about programs with only a few users.

The version of make described in (Feldman, 79) is the standard utility. AT&T modified it in several respects for distribution with System V under the name augmented make (AT&T, 84). We know of two complete rewrites: enhanced make (Hirgelt, 83) and fourth generation make (Fowler, 85). All these versions remain oriented towards program maintenance.

Here at Melbourne we wanted something we could use for text processing. We had access only to standard make and spent a lot of time wrestling with makefiles that kept on getting bigger and bigger. For a while we thought about modifying the make source, but then decided to write something completely new. The basic problem was the inflexibility of make's search algorithm, and this algorithm is too embedded in the make source to be changed easily.

The name cake is a historical accident. Cake follows two other programs whose names were also puns on make. One was bake, a variant of make with built-in rules for VLSI designs instead of C programs (Gedye, 84). The other was David Morley's shell script fake. Written at a time when disc space on our machine was extremely scarce, and full file systems frequently caused write failures, it copied the contents of a directory to /tmp and

¹ Unix is a trademark of AT&T Bell Laboratories.

invoked `make` there.

The structure of the paper is as follows. Section 2 shows how `cake` solves the main problems with `make`, while section 3 describes the most important new features of `cake`. The topics of section 4 are portability and efficiency.

The paper assumes that you have some knowledge of `make`.

2. The problems with `make`

`Make` has three principal problems. These are:

- (1) It supports only suffix-based rules.
- (2) Its search algorithm is not flexible enough.
- (3) It has no provisions for the sharing of new `make` rules.

These problems are built deep into `make`. To solve them we had to start again from scratch. We had to abandon backward compatibility because the `make` syntax is not rich enough to represent the complex relationships among the components of large systems. Nevertheless, the `cake` user interface is deliberately based on `make`'s; this helps users to transfer their skills from `make` to `cake`. The *functionalities* of the two systems are sufficiently different that the risk of confusion is minimal².

Probably the biggest single difference between `make` and `cake` lies in their general attitudes. `Make` is focused on one domain: the maintenance of compiled programs. It has a lot of code specific to this domain (especially the later versions). And it crams all its functionality into some tight syntax that treats all sorts of special things (e.g. `.SUFFIXES`) as if they were files.

`Cake`, on the other hand, uses different syntax for different things, and keeps the number of its mechanisms to the minimum consistent with generality and flexibility. This attitude throws a lot of the functionality of `make` over the fence into the provinces of other programs. For example, where `make` has its own macro processor, `cake` uses the C preprocessor; and where `make` has special code to handle archives, `cake` has a general mechanism that *just happens* to be able to do the same job.

2.1. Only suffix-based rules

All entries in a `makefile` have the same syntax. They do not, however, have the same semantics. The main division is between entries which describe simple dependencies (how to make file `a` from file `b`), and those which describe rules (how to make files with suffix `.x` from files with suffix `.y`)³. `Make` distinguishes the two cases by treating as a rule any dependency whose target is a concatenation of two suffixes.

For this scheme to work, `make` must assume three things. The first is that all interesting files have suffixes; the second is that suffixes always begin with a period; the third is that prefixes are not important. All three assumptions are violated in fairly common situations. Standard `make` cannot express the relationship between `file` and `file.c` (executable and source) because of assumption 1, between `file` and `file,v` (working file and RCS file) because of assumption 2, and between `file.o` and `../src/file.c` (object and source) because of assumption 3. Enhanced `make` and fourth generation `make` have special forms for some of these cases, but these cannot be considered solutions

² This problem, called cognitive dissonance, is discussed in Weinberg's delightful book (Weinberg, 71).

³ For the moment we ignore entries whose targets are special entities like `.IGNORE` `.PRECIOUS` etc.

because special forms will always lag behind demand for them (they are embedded in the make source, and are therefore harder to change than even the built-in rules).

Cake's solution is to do away with make-style rules altogether and instead to allow ordinary dependencies to function as rules by permitting them to contain variables. For example, a possible rule for compiling C programs is

```
% .o:      %.c
          cc -c %.c
```

where the % is the variable symbol. This rule is actually a *template* for an infinite number of dependencies, each of which is obtained by consistently substituting a string for the variable %.

The way this works is as follows. First, as cake seeks to update a file, it matches the name of that file against all the targets in the description file. This matching process gives values to the variables in the target. These values are then substituted in the rest of the rule⁴. (The matching operation is a form of *unification*, the process at the heart of logic programming; this is the reason for the *fifth generation* bit in the title.)

Cake actually supports 11 variables: % and %0 to %9. A majority of rules in practice have only one variable (canonically called %), and most of the other rules have two (canonically called %1 and %2). These variables are local to their rules. Named variables are therefore not needed, though it would be easy to modify the cake source to allow them.

Example

If cake wanted to update prog.o, it would match prog.o against %.o, substitute prog for % throughout the entry, and then proceed as if the cakefile contained the entry

```
prog.o:   prog.c
          cc -c prog.c
```

This arrangement has a number of advantages. One can write

```
%.o:      RCS/%.c,v
          co -u %.c
          cc -c %.c
```

without worrying about the fact that one of the files in the rule was in a different directory and that its suffix started with a nonstandard character. Another advantage is that rules are not restricted to having one source and one target file. This is useful in VLSI, where one frequently needs rules like

```
%.out:    %.in %.circuit
          simulator %.circuit < %.in > %.out
```

and it can also be useful to describe the full consequences of running yacc

⁴ After this the rule should have no unexpanded variables in it. If it does, cake reports an error, as it has no way of finding out what the values of those variables should be.

```

%.c %.h: %.y
    yacc -d %.y
    mv y.tab.c %.c
    mv y.tab.h %.h

```

2.2. Inflexible search algorithm

In trying to write a makefile for a domain other than program development, the biggest problem one faces is usually make's search algorithm. The basis of this algorithm is a special list of suffixes. When looking for ways to update a target `file.x`, make searches along this list from left to right. It uses the first suffix `.y` for which it has a rule `.y.x` and for which `file.y` exists.

The problem with this algorithm manifests itself when a problem divides naturally into a number of stages. Suppose that you have two rules `.c.b` and `.b.a`, that `file.c` exists and you want to issue the command `make file.a`. Make will tell you that it doesn't know how to make `file.a`. The problem is that for the suffix `.b` make has a rule but no file, while for `.c` it has a file but no rule. Make needs a *transitive rule* `.c.a` to go direct from `file.c` to `file.a`.

The number of transitive rules increases as the square of the number of processing stages. It therefore becomes significant for program development only when one adds processing stages on either side of compilers. Under Unix, these stages are typically the link editor `ld` and program generators like `yacc` and `lex`. Half of standard make's builtin rules are transitive ones, there to take care of these three programs. Even so, the builtin rules do not form a closure: some rare combinations of suffixes are missing (e.g. there is no rule for going from `yacc` source to assembler).

For builtin rules a slop factor of two may be acceptable. For rules supplied by the user it is not. A general-purpose makefile for text processing under Unix needs at least six processing stages to handle `nroff/troff` and their preprocessors `lbl`, `bib`, `pic`, `tbl`, and `eqn`, to mention only the ones in common use at Melbourne University.

Cake's solution is simple: if `file1` can be made from `file2` but `file2` does not exist, cake will try to *create* `file2`. Perhaps `file2` can be made from `file3`, which can be made from `file4`, and so on, until we come to a file which does exist. Cake will give up only when there is *absolutely no way* for it to generate a feasible update path.

Both the standard and later versions of make consider missing files to be out of date. So if `file1` depends on `file2` which depends on `file3`, and `file2` is missing, then make will remake first `file2` and then `file1`, even if `file1` is more recent than `file3`.

When using `yacc`, I frequently remove generated sources to prevent duplicate matches when I run `egrep ... *. [chyl]`. If cake adopted make's approach to missing files, it would do a lot of unnecessary work, running `yacc` and `cc` to generate the same parser object again and again⁵.

Cake solves this problem by associating dates even with missing files. The *theoretical update time* of an existing file is its modify time (as given by `stat(2)`); the theoretical update time of a missing file is the theoretical update time of its youngest ancestor. Suppose the

⁵ In this case make is rescued from this unnecessary work by its built-in transitive rules, but as shown

yacc source `parser.y` is older than the parser object `parser.o`, and `parser.c` is missing. Cake will figure that if it recreated `parser.c` it would get a `parser.c` which *theoretically* was last modified at the same time as `parser.y` was, and since `parser.o` is younger than `parser.y`, theoretically it is younger than `parser.c` as well, and therefore up-to-date.

2.3. No provisions for sharing rules

Imagine that you have just written a program that would normally be invoked from a make rule, such as a compiler for a new language. You want to make both the program and the make rule widely available. With standard `make`, you have two choices. You can hand out copies of the rules and get users to include it in their individual `makefiles`; or you can modify the `make` source, specifically, the file containing the built-in rules. The first way is error-prone and quite inconvenient (all those rules cluttering up your `makefile` when you should never need to even look at them). The second way can be impractical; in the development stage because the rules can change frequently and after that because you want to distribute your program to sites that may lack the `make` source. And of course two such modifications may conflict with one another.

Logically, your rules belong in a place that is less permanent than the `make` source but not as transitory as individual `makefiles`. A library file is such a place. The obvious way to access the contents of library files is with `#include`, so `cake` filters every `cakefile` through the C preprocessor.

Cake relies on this mechanism to the extent of not having *any* built-in rules at all. The standard `cake` rules live in files in a library directory (usually `/usr/lib/cake`). Each of these files contains rules about one tool or group of tools. Most user `cakefiles` `#define` some macros and then include some of these files. Given that the source for program `prog` is distributed among `prog.c`, `aux1.c`, `aux2.c`, and `parser.y`, all of which depend on `def.h`, the following would be a suitable `cakefile`:

```
#define MAIN      prog
#define FILES     prog aux1 aux2 parser
#define HDR       def.h

#include <Yacc>
#include <C>
#include <Main>
```

The standard `cakefiles` `Yacc` and `C`, as might be expected, contain rules that invoke `yacc` and `cc` respectively. They also provide some definitions for the standard `cakefile` `Main`. This file contains rules about programs in general, and is adaptable to all compiled languages (e.g. it can handle NU-Prolog programs). One entry in `Main` links the object files together, another prints out all the sources, a third creates a `tags` file if the language has a command equivalent to `ctags`, and so on.

`Make` needs a specialized macro processor; without one it cannot substitute the proper filenames in rule bodies. Fourth generation `make` has not solved this problem but it still wants the extra functionality of the C preprocessor, so it grinds its `makefiles` through both macro processors ! `Cake` solves the problem in another way, and can thus rely on the C preprocessor exclusively.

above this should not be considered a *general* solution.

The original `make` mechanisms are quite rudimentary, as admitted by (Feldman, 79). Unfortunately, the C preprocessor is not without flaws either. The most annoying is that the bodies of macro definitions may begin with blanks, and will if the body is separated from the macro name and any parameters by more than one blank (whether space or tab). `Cake` is distributed with a fix to this problem in the form of a one-line change to the preprocessor source, but this change probably will not work on all versions of Unix and definitely will not work for binary-only sites.

3. The new features of `cake`

The above solutions to `make`'s problems are useful, but they do not by themselves enable `cake` to handle new domains. For this `cake` employs two important new mechanisms: dynamic dependencies and conditional rules.

3.1. Dynamic dependencies

In some situations it is not convenient to list in advance the names of the files a target depends on. For example, an object file depends not only on the corresponding source file but also on the header files referenced in the source.

Standard `make` requires all these dependencies to be declared explicitly in the `makefile`. Since there can be rather a lot of these, most people either declare that all objects depend on all headers, which is wasteful, or declare a subset of the true dependencies, which is error-prone. A third alternative is to use a program (probably an `awk` script) to derive the dependencies and edit them into the `makefile`. (Walden, 84) describes one program that does both these things; there are others. These systems are usually called `makedepend` or some variation of this name.

The problems with this approach are that it is easy to alter the automatically-derived dependencies by mistake, and that if a new header dependency is added the programmer must remember to run `makedepend` again. The C preprocessor solves the first problem; the second, however, is the more important one. Its solution must involve scanning through the source file, checking if the programmer omitted to declare a header dependency. So why not use this scan to *find* the header dependencies in the first place ?

`Cake` attacks this point directly by allowing parts of rules to be specified at run-time. A command enclosed in double square brackets⁶ may appear in a rule anywhere a filename or a list of filenames may appear. For the example of the C header files, the rule would be

```
%.o:      %.c [[ccincl %.c]]
          cc -c %.c
```

signifying that `x.o` depends on the files whose names are listed in the output of the command `ccincl x.c`⁷, as well as on `x.c`. The matching process would convert this rule to

⁶ Single square brackets (like most special characters) are meaningful to `csh`: they denote character classes. However, we are not aware of any legitimate contexts where two square brackets *must* appear together. The order of members in such classes is irrelevant, so if a bracket must be a member of such a class it can be positioned away from the offending boundary (unless the class is a singleton, in which case there is no need for the class in the first place).

⁷ `Ccincl` prints out the names of the files that are `#included` in the file named by its argument.

```
x.o:      x.c [[ccincl x.c]]
          cc -c x.c
```

which in turn would be *command expanded* to

```
x.o:      x.c hdr.h
          cc -c x.c
```

if `hdr.h` were the only header included in `x.c`.

Command patterns provide replacements for fourth generation make's directory searches and special macros. `[[find <dirs> -name <filename> -print]]` does as good a job as the special-purpose make code in looking up source files scattered among a number of directories. `[[basename <filename> <suffix>]]` can do an even better job: make cannot extract the base from the name of an RCS file.

A number of tools intended to be used in just such contexts are distributed together with cake. `Ccincl` is one. `Sub` is another: its purpose is to perform substitutions. Its arguments are two patterns and some strings: it matches each string against the first pattern, giving values to its variables; then it applies those values to the second pattern and prints out the result of this substitution. For example, in the example of section 2.3 the `cakefile` `Main` would invoke the command `[[sub X X.o FILES]]`⁸, the value of `FILES` being `prog aux1 aux2 parser`, to find that the object files it must link together to create the executable `prog` are `prog.o aux1.o aux2.o parser.o`.

Cake allows commands to be nested inside one another. For example, the command `[[sub X.h X [[ccincl file.c]]]]` would strip the suffix `.h` from the names of the header files included in `file.c`⁹.

3.2. Conditional rules

Sometimes it is natural to say that `file1` depends on `file2` *if* some condition holds. None of the make variants provide for this, but it was not too hard to incorporate conditional rules into cake.

A cake entry may have a condition associated with it. This condition, which is introduced by the reserved word `if`, is a boolean expression built up with the operators `and`, `or` and `not` from primitive conditions.

The most important primitive is a command enclosed in double curly braces. Whenever cake considers applying this rule, it will execute this command after matching, substitution and command expansion. The condition will return true if the command's exit status is zero. This runs counter to the intuition of C programmers, but it conforms to the Unix convention of commands returning zero status when no abnormal conditions arise. For example, `{{grep xyzy file}}` returns zero (i.e. true) if `xyzy` occurs in `file` and nonzero (false) otherwise.

⁸ `Sub` uses `X` as the character denoting variables. It cannot use `%`, as all `%`'s in the command will have been substituted for by cake by the time `sub` is invoked.

⁹ As the outputs of commands are substituted for the commands themselves, cake takes care not to scan the new text, lest it find new double square brackets and go into an infinite loop.

Conceptually, this one primitive is all one needs. However, it has considerable overhead, so `cake` includes other primitives to handle some special cases. These test whether a filename occurs in a list of filenames, whether a pattern matches another, and whether a file with a given name exists. Three others forms test the internal `cake` status of targets. This status is `ok` if the file was up-to-date when `cake` was invoked, `cando` if it wasn't but `cake` knows how to update it, and `noway` if `cake` does not know how to update it.

As an example, consider the rule for RCS.

```
%:      RCS/%,v      if exist RCS/%,v
        co -u %
```

Without the condition the rule would apply to all files, even ones which were not controlled by RCS, and even the RCS files themselves: there would be no way to stop the infinite recursion (`%` depends on `RCS/%,v` which depends on `RCS/RCS/%,v,v ...`).

Note that conditions are command expanded just like other parts of entries, so it is possible to write

```
%:      archive      if % in [[ar t archive]]
        ar x archive %
```

4. The implementation

4.1. Portability

`Cake` was developed on a Pyramid 90x under 4.2bsd. It now runs on a VAX under 4.3bsd, a Perkin-Elmer 3240 and an ELXSI 6400 under 4.2bsd, and on the same ELXSI under System V. It has not been tested on either System III or version 7.

`Cake` is written in standard C, with (hopefully) all machine dependencies isolated in the makefile and a header file. In a number of places it uses `#ifdef` to choose between pieces of code appropriate to the AT&T and Berkeley variants of Unix (e.g. to choose between `time()` and `gettimeofday()`). In fact, the biggest hassle we have encountered in porting `cake` was caused by the standard header files. Some files had different locations on different machines (`/usr/include` vs. `/usr/include/sys`), and the some versions included other header files (typically `types.h`) while others did not.

As distributed `cake` is set up to work with `csh`, but it is a simple matter to specify another shell at installation time. (In any case, users may substitute their preferred shell by specifying a few options.) Some of the auxiliary commands are implemented as `csh` scripts, but these are small and it should be trivial to convert them to another shell if necessary.

4.2. Efficiency

Fourth generation make has a very effective optimization system. First, it forks and execs only once. It creates one shell, and thereafter, it pipes commands to be executed to this shell and gets back status information via another pipe. Second, it compiles its `makefiles` into internal form, avoiding parsing except when the compiled version is out of date with respect to the master.

The first of these optimizations is an absolute winner. `Cake` does not have it for the simple reason that it requires a shell which can transmit status information back to its parent process, and we don't have access to one (this feature is provided by neither of the standard

shells, `sh` and `csh`).

`Cake` could possibly make use of the second optimization. It would involve keeping track of the files the C preprocessor includes, so that the `makefile` can be recompiled if one of them changes; this must be done by fourth generation `make` as well though (Fowler, 85) does not mention it. However, the idea is not as big a win for `cake` as it is for `make`. The reason is as follows.

The basic motivations for using `cake` rather than `make` is that it allows one to express more complex dependencies. This implies a bigger system, with more and slower commands than the ones `make` usually deals with. The times taken by `cake` and the preprocessor are insignificant when compared to the time taken by the programs it most often invokes at Melbourne. These programs, `ditroff` and `nc` (the NU-Prolog compiler that is itself written in NU-Prolog), are notorious CPU hogs.

Here are some statistics to back up this argument. The *overhead ratio* is given by the formula

$$\frac{\text{cake process system time} + \text{children user time} + \text{children system time}}{\text{cake process user time}}$$

This is justifiable given that the `cake` implementor has direct control only over the denominator; the kernel and the user's commands impose a lower limit on the numerator.

We have collected statistics on every `cake` run on two machines at Melbourne¹⁰. These statistics show that the processes and system calls invoked by `cake` take on average about 70-80 times as much CPU time as the `cake` process itself. This suggests that the best way to lower total CPU time is not to tune `cake` itself but to reduce the number of child processes. To this end, `cake` caches the status returned by all condition commands `{{command}}` and the output of all command patterns `[[command]]`. The first cache has a hit ratio of about 50 percent, corresponding to the typical practice in which a condition and its negation select one out of a pair of rules. The second cache has a hit ratio of about 75 percent; these hits are usually the second and later occurrences of macros whose values contain commands.

`Cake` also uses a second optimization. This one is borrowed from standard `make`: when an action contains no constructs requiring a shell, `cake` itself will parse the action and invoke it through `exec`. We have no statistics to show what percentage of actions benefit from this, but a quick examination of the standard `cakefiles` leads us to believe that it is over 50 percent.

Overall, `cake` can do a lot more than `make`, but on things which *can* be handled by `make`, `cake` is slightly slower than standard `make` and a lot slower than fourth generation `make`. Since the main goal of `cake` is generality, not efficiency, this is understandable. If efficiency is important, `make` is always available as a fallback.

4.3. Availability

`Cake` has been fairly stable for about six months now. During this time it has been used without major problems by about twenty people here at Melbourne. It will be posted to the net in the near future, complete with auxiliary programs and manual entries.

¹⁰ On `munmurra` (an EXLSI 6400), the main application is Prolog compilation; on `mulga` (a Perkin-Elmer 3240), the main applications are text processing and the maintenance of a big bibliography (over 36000 references).

5. Acknowledgements

John Shepherd, Paul Maisano, David Morley and Jeff Schultz helped me to locate bugs by being brave enough to use early versions of `cake`. I would like to thank John for his comments on drafts of this paper.

This research was supported by a Commonwealth Postgraduate Research Award, the Australian Computer Research Board, and Pyramid Australia.

6. References

- (AT&T, 84) Augmented version of `make`, in: *Unix System V - release 2.0 support tools guide*, AT&T, April 1984.
- (Feldman, 79) Stuart I. Feldman, `Make` - a program for maintaining computer programs, *Software - Practice and Experience*, 9:4 (April 1979), pp. 255-265.
- (Fowler, 85) Glenn S. Fowler, A fourth generation `make`, *Proceedings of the USENIX Summer Conference*, Portland, Oregon, June 1985, pp. 159-174.
- (Gedye, 84) David Gedye, Cooking with CAD at UNSW, Joint Microelectronics Research Center, University of New South Wales, Sydney, Australia, 1984.
- (Hirgelt, 83) Edward Hirgelt, Enhancing `make` or re-inventing a rounder wheel, *Proceedings of the USENIX Summer Conference*, Toronto, Ontario, Canada, June 1983, pp. 45-58.
- (Walden, 84) Kim Walden, Automatic generation of `make` dependencies, *Software - Practice and Experience*, 14:6 (June 1984), pp. 575-585.
- (Weinberg, 71) Gerald M. Weinberg, *The psychology of computer programming*, Van Nostrand Reinhold, New York, 1971.

UNIXTM MENU SYSTEM

Keith Godfrey
Telecom Australia Educational Fellow
University of Western Australia (keith@trlluna.oz)

Peter Y. F. Hui
Principal Computer Systems Officer
Telecom Australia Research Laboratories (hui@trlsasb.oz)

© Australian Telecommunications Commission 1987
Permission to reprint in the AUUGN is kindly acknowledged.

ABSTRACT

This paper describes the implementation of a "user friendly" interface to the UNIX systems. It allows non-technical users to utilize the UNIX operating system with minimal training.

A powerful menu driven system has resulted which encompasses most of the capabilities required by such users without encountering a shell prompt. On-line help is also available. The system produces compilable C code from a menu script file - a special language. The compiled menu system is fast, flexible and friendly.

1 Introduction

The Telecom Australia Research Laboratories in Melbourne have a number of computers with the UNIX operating system. These machines are used to support research work, to exchange electronic mail with other research organisations, and to provide a programming environment for software development.

Most of the managerial staff and researchers are not familiar with UNIX shell commands and have no desire to learn them. They only want to use the computer as a tool to capture and analyse data, then carry on with their administrative or research work.

The problem is that these users run into difficulties as soon as they need to do anything that is outside their normal operations. Often the required task is very simple, perhaps to create or remove a directory, copy or rename a file, or merely change the password of their account. But they cannot quite remember the command. In vain they try several commands that sound similar, say "password", "pw", "pwd", "newpass" or "pass", then panic and call the system manager for help.

Trying random commands is generally harmless, but it can cause a lot of trouble if the user executes a valid command that means something else. Executing "pwd" instead of "passwd" will merely display the name of the working directory. Trying "cat" for a catalog of files (instead of "ls") will lock the terminal until Control-C or Control-D is typed since it will copy stdin to stdout. Unfamiliar users are not likely to deduce that.

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

The aim of this menu driven interface is to perform all the necessary commands without the user needing to know the commands and parameters.

All the programs have been written in C under UltrixTM Version 1.2 on a MicroVAX II.

2 Menu Operation

A menu system in a computer is rather like a menu in a restaurant. Presentation is very important. Menus should look elegant, with the title and list of choices clearly shown. Although it may be desirable to give a description of each item, it should be kept to an absolute minimum otherwise it will become cluttered and the user will get confused. Menus should be clear and concise.

The selection process must be very flexible; users need to be able to make their choices with ease. Since there are many types of user, the menu system should be responsive to several different styles of selection.

This menu system has been designed with these points in mind.

The screen display can be broken up into several distinct sections; the menu number, title, list of choices, description of selected choice, and instructions on how to choose. A typical menu screen is shown in figure 1. Due to problems of reproduction, text which would be in inverse video is represented in boldface.

The user selects an item by moving the "highlight" up or down to the desired choice then pressing the RETURN key to confirm it.

The highlight can be moved in several ways:

- By pressing the UP or DOWN arrows. (The most logical way.)
- By pressing SPACE to move down or BACKSPACE to move up. (This method was included to suit the people who are familiar with the Wang Menu System.)
- By entering the FIRST LETTER of the item's name. (This is a quick way to skip down the menu.)
- By entering the NUMBER of the item. (It is also possible to go directly to any other place in the menu system using this method.)

The highlight (shown in boldface in figure 1) will rotate back to the beginning of the list if it is moved off the last line and vice-versa. If any intervening items are missing from the menu, the highlight will automatically skip across the gap.

Each time the highlight is moved, a brief description of the highlighted choice is displayed near the bottom of the screen. This provides the user with additional information about the list of choices without cluttering the screen.

TMUltrix is a trademark of Digital Equipment Corporation.

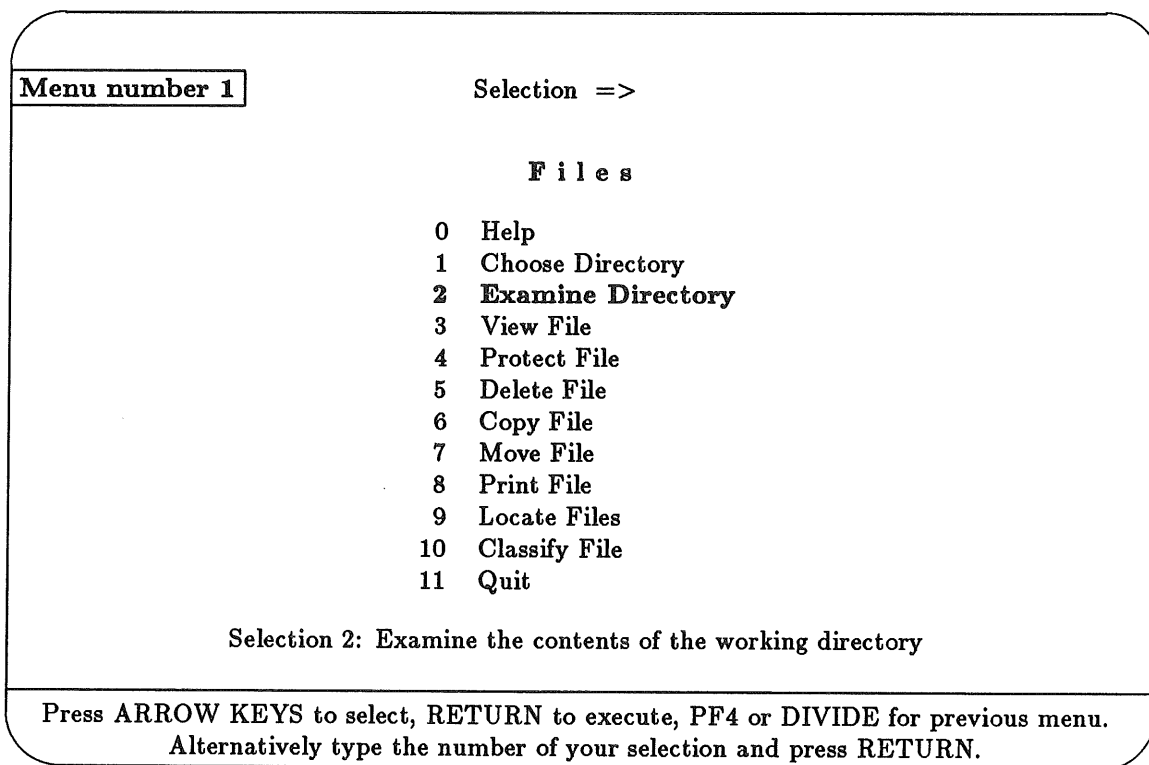


Figure 1: Sample Menu Screen

3 Menu Tree

The UNIX operating system has a large number of commands, which cannot all fit in a single menu. It is therefore necessary to have menus of sub-menus, arranged in a tree structure.

This system allows the menus to be linked together in many ways. A logical way is to separate the UNIX commands into functional groups:

- File Handling Commands (ls [-alR], cat files, rm files, ...)
- Directory Handling Commands (mkdir dir, rmdir dir, rm dir/*, ...)
- Personal Account Details (whoami, ps -x, quota, groups, ...)
- System Information (uptime, w -h, ps -augx, vmstat -f, ...)
- Special Applications (man, sh -c command, csh)
- Printer Commands (lpr file, lpq, lprm)
- Mail System Commands (mail: send, read, delete, ...)
- News System Commands (vnews)
- Editor Commands (vi file)

The menu tree is then constructed accordingly, figure 2 shows an abbreviated representation of such a tree.

Menus or items are identified uniquely by their number. Number 0 is the top of the tree


```

Main Selections
0  Help Information
1  File Commands
    1.0  Help Information
    1.1  Choose Directory
    1.2  Examine Directory
        1.2.0  Help
        1.2.1  List Directory
            :
            :
        1.2.6  Subdirectories (y/n)
        1.2.7  Quit
    1.3  View File
        :
        :
    1.9  Locate Files
    1.11 Quit
2  Directory Commands
    :
    :
4  System Information
etc.

```

Figure 3: Numbering of The Menu Tree

5 Implementation

Generally there are two ways to implement a menu system:

- Compiler Reads a file of menu definitions and creates an executable menu system.
- Interpreter Loads menu definition files while running.

Each has several advantages and disadvantages. It was decided to build a compiler because:

- The resulting menus operate much faster.
- All menu definitions can be contained in a single file.
- Compilation time is small (about 5 minutes on a MicroVAX for a complicated system).

6 The Programs

The menu system uses a two-stage compiler:

- | | | |
|----------|----------|---|
| Stage 1. | menucom1 | Condenses the menu script file <i>menu.in</i> and produces a temporary file <i>menu.out</i> |
| Stage 2. | menucom2 | Processes the condensed script file <i>menu.out</i> and the runtime function library file <i>menu.include</i> and reads all specified message files <i>menu.help.*</i> then produces C source code file <i>menu.c</i> |

The C compiler *cc* is then used to compile the source code *menu.c* into an executable program *menu*. Figure 4 shows the compilation process.

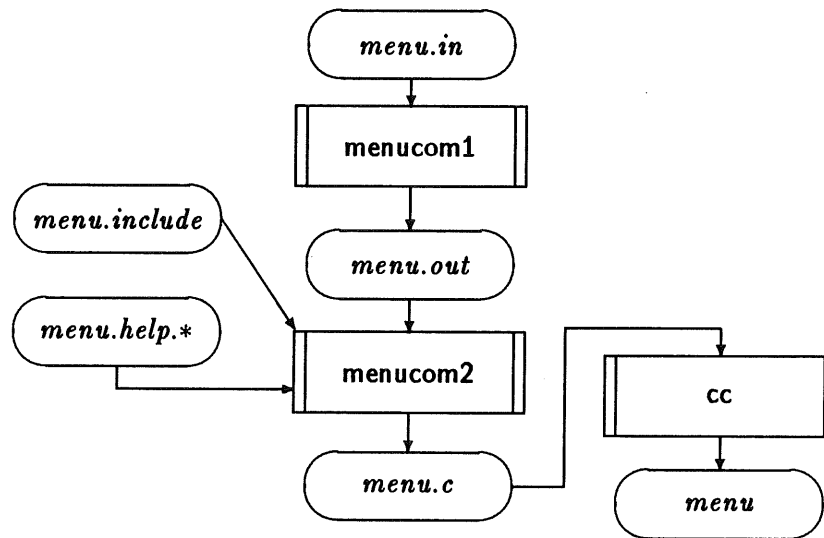


Figure 4: Flow Chart of the Menu Compilation Process

Two satellite programs have been included to perform specialised tasks while conforming to the screen format of menu:

- ct* Operates in a similar manner to *cat* but includes paging and filtering of escape sequences.
- chmode* Performs the same task as *chmod* but displays the current file mode and visually adjusts it via a menu, instead of asking for a four digit octal number.

These can be executed in place of the normal UNIX commands.

7 Menu Script Language

The menu layout is specified in a file called *menu.in*. It contains the definitions of all the items in all menus, listed one after another. They do not need to be in numeric order.

Each item definition will require at least two lines of the file. Most items, especially UNIX commands, will require more.

The first line contains the number, name, and description of the item. This has the form:

```
number name: description
```

Example:

2.4 Empty Directory: Remove all the files within a directory

This causes menu number 2 to display "4 Empty Directory" as the fourth choice. Selecting this option (by moving the highlight on to it) produces the explanation

Selection 4: Remove all the files within a directory

at the bottom of the screen.

The second and subsequent lines of the item definition determine what the item will do when it is chosen. Some items will call up another menu (for example, the 1st choice of the Main Selections menu will display the Files menu). Others will display help information or other messages. The majority will perform UNIX commands.

The syntax of the second line is:

type [parameters]

where "type" refers to the type of task that the item will perform and "parameters" are additional information if required.

There are currently 10 types of task supported which are defined briefly in Tables 1, 2 and 3.

MENU	The item is a sub-menu, or branch down the tree from the menu the item is in.
GOTO <item>	This provides cross-branching between menus. Instead of an item being a sub-menu, it can refer to any menu or item, anywhere in the tree.
GO <item>	Same as GOTO <item>.
UP	UP is a special form of GOTO. It is equivalent to GOing to the parent menu, one hop up the tree.
MESSAGE <file>	The file is read during compilation of the menu script and will be displayed as a text message whenever this option is chosen. This is the best way to present help information. Message files may contain special characters to adjust the screen attributes or display graphics symbols.

Table 1: Menu Script Tasks for Sequencing

The ultimate purpose of the menu system is to execute UNIX commands. These are handled by the COMMAND, EXECUTE and LOOP instructions illustrated in Table 3.

<p>FLAG <x></p>	<p>Many UNIX commands have toggles or flags that can change the performance of the command. This item type will determine how those parameters are set according to the following presets:</p> <p style="padding-left: 40px;"> FLAG 0 (Item is initially "off") FLAG 1 (Item is initially "on") </p>
<p>THREE <x></p>	<p>A THREE is a special form of a FLAG. It has three possible states; set, reset, or neither. This is used by commands that can switch a feature on, off, or leave it unchanged. The definitions are:</p> <p style="padding-left: 40px;"> THREE + (Item is initially "set") THREE - (Item is initially "reset") THREE . (Item is initially "no change") </p>

Table 2: Menu Script Tasks for Setting Flags

<p>COMMAND</p>	<p>Following the word COMMAND there may be as many separate lines of shell commands as are required. After they have all been executed the user is prompted to press any key to continue with the menus (so that the terminal output is not lost).</p>
<p>EXECUTE</p>	<p>EXECUTE is similar to COMMAND but does not wait for a key to be pressed. The menu re-appears as soon as the command has finished.</p>
<p>LOOP</p>	<p>If LOOP is specified at the very beginning (prior to the first COMMAND or EXECUTE), the whole lot will repeat until the user directs it to stop.</p>

Table 3: Menu Script Tasks for Process Initiation

A menu item may contain several COMMANDs and EXECUTEs, in any order, each with a list of many shell commands. This allows for pauses between commands and, when combined with functions (described shortly), provides a very powerful way to execute groups of UNIX commands intertwined with user input. An example of this approach is shown in figure 5.

```
8.1 Read News: Use the News system
COMMAND
  echo 'You are about to enter the news system.'
  echo
  echo 'News articles are presented one page at'
  echo 'a time. You will be asked "more?" at the'
  echo 'end of each page.'
  echo
  echo '  to continue:      press RETURN (more).'
  echo '  to skip to next:  press  "n" (next).'
  echo '  to quit:          press  "q" (quit).'
  echo '  for help:         press  "?" (help).'
  echo
EXECUTE
  echo 'Getting news.'
  echo
  vnews
```

Figure 5: An Example of The Use of COMMAND & EXECUTE

There are two stages to this item. The first echoes the instructions and waits for the user to read them; the second puts the message "Getting news." on to the screen and loads the news system. When the user exits vnews, the menu will re-appear without waiting.

An extension of the EXECUTE and COMMAND types is the LOOP type which allows continual repetition of the commands.

8 Functions

Parameters are passed to UNIX commands via functions. There are several functions defined by the menu system. Each returns a text string that is included as part of the command to be executed.

Functions have the form:

```
@function(parameter)
```

where "function" is the name and "parameters" are any parameters that it needs. When a function appears in a UNIX command, it is processed and replaced by its result.

Example: the "Make Directory" command.

2.2 Make Directory: Create a new Directory

EXECUTE

```
mkdir @ask("Enter the name of the directory to be created - ")
```

When this item is chosen it will:

- Ask the user the question "Enter the name of the directory to be created - " and read the answer.
- Substitute the answer into the UNIX command in place of the @ask(...) such that if the user answers "xyz", the command becomes "mkdir xyz".
- Operate the UNIX command and return immediately to the main menu.

The @ask("question") is a function which asks a question and returns the answer.

Functions are defined in the file *menu.include* and are illustrated in Table 4.

@flag(number, "on", "off")	The @flag function determines the state of the item <number> which must be defined as FLAG, and substitutes <on> if on or <off> if off. Example: See the "ls" command further on.
@three(number, "set", "reset", "neither")	The @three function is very similar to the @flag function, except that it works on items defined as THREE instead of FLAG.
@ask("question")	A simple question-answer system is provided by the @ask function. It asks the <question> and substitutes the answer.
@file("question")	@file displays a list of the files in the working directory and lets the user select one. Alternatively, the user may type the name of any file anywhere in the machine, in response to the <question>.
@files("question")	The @files function allows for several files to be selected from a displayed list.
@confirm("question")	@confirm allows the user to abort a command if it is not desired. This may be useful for preventing accidental mistakes. It returns nothing if okay, or aborts the item if not.
@exit()	The @exit function terminates the menu system. It should be used in conjunction with the @confirm function, to ensure the user doesn't accidentally drop out.

Table 4: Functions Pre-defined In *menu.include*

The "ls" command demonstrates the usage of flags. It is defined in the menu "1.2 Examine Directory" which is shown in Figure 6.

```

1.2 Examine Directory: Examine the contents of the working directory
  MENU
1.2.0 Help: How to interpret the contents of the directory
  MESSAGE menu.help.ls
1.2.1 List Directory: List the files in the working directory
  COMMAND
    echo -n 'Directory of  '; pwd; echo
    ls @flag(1.2.3,"-l","") @flag(1.2.4,"-a","") \
    @flag(1.2.5,"-g","") @flag(1.2.6,"-R","")
1.2.2 Choose Directory: Select working directory
  EXECUTE
    chdir @subdir("Enter the name of the new working directory - ")
1.2.3 Long Listing: Displays mode,owner,size and other details
  FLAG 1
1.2.4 All Entries: \
  Lists all files (including names which start with a period)
  FLAG 1
1.2.5 Group Ownership: Includes group owner with Long Listing
  FLAG 1
1.2.6 Sub-Directories: Recursively lists all sub-directories
  FLAG 0
1.2.7 Quit: Go back to File Maintenance
  UP

```

Figure 6: Script for Examine Directory – part of *menu.in*

Note that a backslash “\” at the end of a line in the file *menu.in* indicates continuation on to the next line.

Item 1.2.3 is defined as FLAG and can be toggled by the user. The function @flag(1.2.3, “-l”, “”) determines its state and substitutes “-l” if it is on, or “” if it is off. Similarly the other items.

The initial state comprises flags 1.2.3 on, 1.2.4 on, 1.2.5 on, and 1.2.6 off. The resulting command at item 1.2.1 would therefore be “ls -l -a -g”. This will change with each change of the flags.

All functions within a COMMAND or EXECUTE group are substituted before the whole group is executed. By careful usage of COMMAND, EXECUTE, LOOP and functions it is possible to specify a very wide variety of tasks.

9 The Include File

The file *menu.include* contains all the functions that the menu system requires to operate. It is copied directly to the start of *menu.c*.

Included are:

- cursor strings (cursor movement, underlining etc.)
- procedures required by MENU, FLAG / THREE, COMMAND / EXECUTE, MESSAGE. (menu display, flag toggling, wait for key, message display).
- procedures required by '@' functions within commands. (flag, three, ask, confirm, exit, subdir, file, files.)

As mentioned earlier, the functions provide much of the power of the menu system. Parameters are passed to them from the definitions in *menu.in* and their results are substituted into the UNIX commands.

The menu compiler adjusts the function name and calling parameters to suit the requirements of C compiler. If the function specified in *menu.in* is

```
@ask("question")
```

the way it will be called in *menu.c* will be

```
ask_("question",result)
```

where *result* is a pointer to a character string in which to store the answer. All functions are called with this extra parameter. The string placed there will be substituted into the UNIX command by the caller.

The function name has the '@' removed to make it begin with a letter (as required by the C compiler). It is distinguished from other procedures in *menu.include* by the addition of the '_'.

There are a number of subroutines in *menu.include* that can be called by the functions. These perform special operations on the screen such as graphics, highlighting, displaying instruction messages and selection.

10 Conclusion

The system has been successfully tested by a number of users. It is expected that it will be shortly installed on many of the UNIX computers at the Telecom Australia Research Laboratories.

;login:

The USENIX Association Newsletter

Volume 12, Number 2

March/April 1987

CONTENTS

MINIX: A UNIX Clone with Source Code for the IBM PC	3
<i>Andrew S. Tanenbaum</i>	
Program for the Phoenix Technical Conference	10
Ten Years Ago in UNIX NEWS	12
The DASH Project: Design Issues for Very Large Distributed Systems	13
<i>David P. Anderson and Domenico Ferrari</i>	
Book Review: The Nutshell Handbooks	15
<i>Lou Katz</i>	
Summary of the Board of Directors' Meeting October 1-2, 1986	19
Summary of the Board of Directors' Meeting January 19-20 & 22, 1987	20
Future Meetings	22
Financial Statements of the USENIX Association	23
WEIRDNIX Competition	26
Publications Available	27
Software Tapes	27
4.3BSD UNIX Manuals	28
4.3BSD Manual Reproduction Authorization and Order Form	29
Local User Groups	30

The closing date for submissions for the next issue of ;login: is April 24, 1987

USENIX THE PROFESSIONAL AND TECHNICAL
UNIX® ASSOCIATION

;login:

MINIX: A UNIX Clone with Source Code for the IBM PC

Andrew S. Tanenbaum

Dept. of Mathematics and Computer Science
Vrije Universiteit
Amsterdam, The Netherlands
Usenet: minix@cs.vu.nl

ABSTRACT

This article describes a new operating system, called MINIX, that is functionally compatible with Version 7 UNIX[®]. It has been rewritten completely from scratch. Neither the kernel nor the utility programs contain any AT&T code, so the source code is free from AT&T licensing restrictions and may be studied by individuals or in a course. The system runs on the IBM PC, XT, or AT, and does not require a hard disk, thus making it possible for individuals to acquire a UNIX-like system for home use at a very low cost. If a hard disk is available, it is fully supported, however.

Internally, MINIX is structured completely differently from UNIX. It is a message passing system on top of which are memory and file servers. User processes can send messages to these servers to have system calls carried out. The paper describes the motivation and intended use of the system, what the distribution contains, and discusses the system architecture in some detail.

Introduction

MINIX is a new operating system for the IBM PC, XT, and AT. Functionally, it is system-call compatible with Version 7 UNIX, but inside it has been rewritten from scratch. It contains no AT&T code at all, neither in the kernel nor in the utilities. Furthermore, the internal structure is also completely different (it is much more modular). The source code is being released without a restrictive licence for the benefit of those people who would like to have access to the source code of a UNIX-like system. This point is discussed in more detail at the end of this paper.

Another feature of this system is that it has been designed to run with inexpensive hardware. Many people whose work involves computers also have a computer at home. While many of these people have an ego that says "VAX" their wallets often say "IBM PC." MINIX has therefore been designed to run on a 256K IBM PC with one floppy disk if it has to, but the system is then highly restricted. On a 640K IBM PC with two floppy disks, it runs quite well and can even recompile itself from the source code provided, using its own C compiler. On a hard disk system or a PC-AT it

works even better, of course. It also runs on those clones that are 100% hardware compatible with the PC, XT, or AT. Experiments have shown that about 80% of all clones are compatible, but 20% are not.

As a natural consequence of this "small is beautiful" philosophy, I decided to have MINIX be compatible with Version 7 rather than, say, 4.3 BSD or System V. Making a 4.3 BSD or System V compatible system that runs on a 256K IBM PC with one 360K floppy disk is left as an exercise for the reader. Besides, there are many people who believe that Version 7 was not only an improvement on all its predecessors, but also on all its successors, certainly in terms of simplicity, coherence, and elegance.

MINIX implements all the V7 system calls, except ACCT, LOCK, MPX, NICE, PHYS, PKON, PKOFF, PROFIL, and PTRACE. The other system calls are all implemented in full, and are exactly compatible with V7. In particular, FORK and EXEC are fully implemented, so MINIX can be configured as a normal multiprogramming system, with several background jobs running at the same time (memory permitting), and even multiple users.

;login:

The MINIX shell is compatible with the V7 (Bourne) shell, so to the user at the terminal, running MINIX looks and feels like running UNIX. Over 60 utility programs are part of the software distribution, including `ar`, `basename`, `cat`, `cc`, `chmem`, `chmod`, `chown`, `cmp`, `comm`, `cp`, `date`, `dd`, `df`, `echo`, `grep`, `head`, `kill`, `ln`, `login`, `lpr`, `ls`, `make`, `mkdir`, `mkfs`, `mknod`, `mount`, `mv`, `od`, `passwd`, `pr`, `pwd`, `rev`, `rm`, `rmdir`, `roff`, `sh`, `size`, `sleep`, `sort`, `split`, `stty`, `su`, `sum`, `sync`, `tail`, `tar`, `tee`, `time`, `touch`, `tr`, `umount`, `uniq`, `update`, and `wc`. A full-screen editor loosely inspired by `emacs`, a full Kernighan and Ritchie compatible C compiler, and programs to read and write MS-DOS diskettes are also included. All of the sources of the operating system and these utilities, except the C compiler source (which is quite large and is available separately), are included in the software package.

In addition to the above utilities, over 100 library procedures, including `stdio`, are provided, again with the full source code.

To reiterate what was said above, all of this software is completely new. Not a single line of it is taken from, or even based on the AT&T code. I personally wrote from scratch the entire operating system and some of the utilities. This took about three years. My students and some other generous people wrote the rest. The C compiler is derived from the Amsterdam Compiler Kit (*CACM*, Sept. 1983), and was written at the Vrije Universiteit. It is a top-down, recursive descent compiler written in a compiler writing language called `LLGEN` and is not based on or in any way related to the AT&T portable C compiler, which is a bottom-up, LALR compiler written in `yacc`.

Overview of the MINIX System Architecture

UNIX is organized as a single executable program that is loaded into memory at system boot time and then run. MINIX is structured in a much more modular way, as a collection of processes that communicate with each other and with user processes by sending and receiving messages. There are separate processes for the memory manager, the file system, for each device driver, and for certain

other system functions. This structure enforces a better interface between the pieces. The file system cannot, for example, accidentally change the memory manager's tables because the file system and memory manager each have their own private address spaces.

These system processes are each full-fledged processes, with their own memory allocation, process table entry and state. They can be run, blocked, and send messages, just as the user processes. In fact, the memory manager and file system each run in user space as ordinary processes. The device drivers are all linked together with the kernel into the same binary program, but they communicate with each other and with the other processes by message passing.

When the system is compiled, four binary programs are independently created: the kernel (including the driver processes), the memory manager, the file system, and `init` (which reads `/etc/tty`s and forks off the login processes). In other words, compiling the system results in four distinct *a.out* files. When the system is booted, all four of these are read into memory from the boot diskette.

It is possible, and in fact, normal, to modify, recompile, and relink, say, the file system, without having to relink the other three pieces. This design provides a high degree of modularity by dividing the system up into independent pieces, each with a well-defined function and interface to the other pieces. The pieces communicate by sending and receiving messages.

The various processes are structured in four layers:

4. The user processes (top layer).
3. The server processes (memory manager and file system).
2. The device drivers, one process per device.
1. Process and message handling (bottom layer).

Let us now briefly summarize the function of each layer.

Layer 1 is concerned with doing process management including CPU scheduling and interprocess communication. When a process does a `SEND` or `RECEIVE`, it traps to the kernel, which then tries to execute the

;login:

command. If the command cannot be executed (e.g., a process does a RECEIVE and there are no messages waiting for it), the caller is blocked until the command can be executed, at which time the process is reactivated. When an interrupt occurs, layer 1 converts it into a message to the appropriate device driver, which will normally be blocked waiting for it. The decision about which process to run when is also made in layer 1. A priority algorithm is used, giving device drivers higher priority over ordinary user processes, for example.

Layer 2 contains the device drivers, one process per major device. These processes are part of the kernel's address space because they must run in kernel mode to access I/O device registers and execute I/O instructions. Although the IBM PC does not have user mode/kernel mode, most other machines do, so this decision has been made with an eye toward the future. To distinguish the processes within the kernel from those in user space, the kernel processes are called tasks.

Layer 3 contains only two processes, the memory manager and the file system. They are both structured as servers, with the user processes as clients. When a user process (i.e., a client) wants to execute a system call, it calls, for example, the library procedure read with the file descriptor, buffer, and count. The library procedure builds a message containing the system call number and the parameters and sends it to the file system. The client then blocks waiting for a reply. When the file system receives the message, it carries it out and sends back a reply containing the number of bytes read or the error code. The library procedure gets the reply and returns the result to the caller in the usual way. The user is completely unaware of what is going on here, making it easy to replace the local file system with a remote one.

Layer 4 contains the user programs. When the system comes up, init forks off login processes, which then wait for input. On a successful login, the shell is executed. Processes can fork, resulting in a tree of processes, with init at the root. When Control-D is typed to a shell, it exits, and init replaces the shell with another login process.

Layer 1 — Processes and Messages

The two basic concepts on which MINIX is built are processes and messages. A process is an independently schedulable entity with its own process table entry. A message is a structure containing the sender's process number, a message type field, and a variable part (a union) containing the parameters or reply codes of the message. Message size is fixed, depending on how big the union happens to be on the machine in question. On the IBM PC it is 24 bytes.

Three kernel calls are provided:

- RECEIVE(source, &message)
- SEND(destination, &message)
- SENDREC(process, &message)

These are the only true system calls (i.e., traps to the kernel). RECEIVE announces the willingness of the caller to accept a message from a specified process, or ANY, if the RECEIVER will accept any message. (From here on, "process" also includes the tasks.) If no message is available, the receiving process is blocked. SEND attempts to transmit a message to the destination process. If the destination process is currently blocked trying to receive from the sender, the kernel copies the message from the sender's buffer to the receiver's buffer, and then marks them both as runnable. If the receiver is not waiting for a message from the sender, the sender is blocked.

The SENDREC primitive combines the functions of the other two. It sends a message to the indicated process, and then blocks until a reply has been received. The reply overwrites the original message. User processes use SENDREC to execute system calls by sending messages to the servers and then blocking until the reply arrives.

There are two ways to enter the kernel. One way is by the trap resulting from a process' attempt to send or receive a message. The other way is by an interrupt. When an interrupt occurs, the registers and machine state of the currently running process are saved in its process table entry. Then a general interrupt handler is called with the interrupt number as parameter. This procedure builds a message of type INTERRUPT, copies it to the buffer of the waiting task, marks that task as runnable

;login:

(unblocked), and then calls the scheduler to see who to run next.

The scheduler maintains three queues, corresponding to layers 2, 3, and 4, respectively. The driver queue has the highest priority, the server queue has middle priority, and the user queue has lowest priority. The scheduling algorithm is simple: find the highest priority queue that has at least one process on it, and run the first process on that queue. In this way, a clock interrupt will cause a process switch if the file system was running, but not if the disk driver was running. If the disk driver was running, the clock task will be put at the end of the highest priority queue, and run when its turn comes.

In addition to this rule, once every 100 msec, the clock task checks to see if the current process is a user process that has been running for at least 100 msec. If so, that user is removed from the front of the user queue and put on the back. In effect, compute bound user processes are run using a round robin scheduler. Once started, a user process runs until either it blocks trying to send or receive a message, or it has had 100 msec of CPU time. This algorithm is simple, fair, and easy to implement.

Layer 2 — Device Drivers

Like all versions of UNIX for the IBM PC, MINIX does not use the ROM BIOS for input or output because the BIOS does not support interrupts. Suppose a user forks off a compilation in the background and then calls the editor. If the editor tried to read from the terminal using the BIOS, the compilation (and any other background jobs such as printing) would be stopped dead in their tracks waiting for the the next character to be typed. Such behavior may be acceptable in the MS-DOS world, but it certainly is not in the UNIX world. As a result, MINIX contains a complete set of drivers that duplicate the functions of the BIOS. Like the rest of MINIX, these drivers are written in C, not assembly language.

This design has important implications for running MINIX on PC clones. A clone whose hardware is not compatible with the PC down to the chip level, but which tries to hide the differences by making the BIOS calls functionally identical to IBM's will not run an

unmodified MINIX because MINIX does not use the BIOS.

Each device driver is a separate process in MINIX. At present, the drivers include the clock driver, terminal driver, various disk drivers (e.g., RAM disk, floppy disk), and printer driver. Each driver has a main loop consisting of three actions:

1. Wait for an incoming message.
2. Perform the request contained in the message.
3. Send a reply message.

Request messages have a standard format, containing the opcode (e.g., READ, WRITE, or IOCTL), the minor device number, the position (e.g., disk block number), the buffer address, the byte count, and the number of the process on whose behalf the work is being done.

As an example of where device drivers fit in, consider what happens when a user wants to read from a file. The user sends a message to the file system. If the file system has the needed data in its buffer cache, they are copied back to the user. Otherwise, the file system sends a message to the disk task requesting that the block be read into a buffer within the file system's address space (in its cache). Users may not send messages to the tasks directly. Only the servers may do this.

MINIX supports a RAM disk. In fact, the RAM disk is always used to hold the root device. When the system is booted, after the operating system has been loaded, the user is instructed to insert the root file system diskette. The file system then sees how big it is, allocates the necessary memory, and copies the diskette to the RAM disk. Other file systems can then be mounted on the root device.

This organization puts important directories such as */bin* and */tmp* on the fastest device, and also makes it easy to work with either floppy disks or hard disks or a mixture of the two by mounting them on */usr* or */user* or elsewhere. In any event, the root device is always in the same place.

In the standard distribution, the RAM disk is about 240K, most of which is full of parts of the C compiler. In the 256K system, a much smaller RAM disk has to be used, which explains why this version has no C compiler:

;login:

there is no place to put it. (The */usr* diskette is completely full with the other utility programs and one of the design goals was to make the system run on a 256K PC with one floppy disk.) Users with an unusual configuration such as 256K and three hard disks are free to juggle things around as they see fit.

The terminal driver is compatible with the standard V7 terminal driver. It supports cooked mode, raw mode, and cbreak mode. It also supports several escape sequences, such as cursor positioning and reverse scrolling because the screen editor needs them.

The printer driver copies its input to the printer character for character without modification. It does not even convert line feed to carriage return + line feed. This makes it possible to send escape sequences to graphics printers without the driver messing things up. MINIX does not spool output because floppy disk systems rarely have enough spare disk space for the spooling directory. Instead one normally would print a file *f* by saying

```
lpr <f &
```

to do the printing in the background. The *lpr* program inserts carriage returns, expands tabs, and so on, so it should only be used for straight ASCII files. On hard disk systems, a spooler would not be difficult to write.

Layer 3 — Servers

Layer 3 contains two server processes: the memory manager and the file system. They are both structured in the same way as the device drivers, that is a main loop that accepts requests, performs them, and then replies. We will now look at each of these in turn.

The memory manager's job is to handle those system calls that affect memory allocation, as well as a few others. These include FORK, EXEC, WAIT, KILL, and BRK. The memory model used by MINIX is exceptionally simple in order to accommodate computers without any memory management hardware. When the shell forks off a process, a copy of the shell is made in memory. When the child does an EXEC, the new core image is placed in memory. Thereafter it is never moved. MINIX does not swap or page.

The amount of memory allocated to the process is determined by a field in the header of the executable file. A program, *chmem*, has been provided to manipulate this field. When a process is started, the text segment is set at the very bottom of the allocated memory area, followed by the data and *bss*. The stack starts at the top of the allocated memory and grows downward. The space between the bottom of the stack and the top of the data segment is available for both segments to grow into as needed. If the two segments meet, the process is killed.

In the past, before paging was invented, all memory allocation schemes worked like this. In the future, when even small microcomputers will use 32-bit CPUs and 1M × 1 bit memory chips, the minimum feasible memory will be 4 megabytes and this allocation scheme will probably become popular again due to its inherent simplicity. Thus the MINIX scheme can be regarded as either hopelessly outdated or amazingly futuristic, as you prefer.

The memory manager keeps track of memory using a list of holes. When new memory is needed, either for FORK or for EXEC, it searches the hole list and takes the first hole that is big enough (first fit). When a process terminates, if it is adjacent to a hole on either side, the process' memory and the hole are merged into a bigger hole.

The file system is really a remote file server that happens to be running on the user's machine. However it is straightforward to convert it into a true network file server. All that needs to be done is change the message interface and provide some way of authenticating requests. (In MINIX, the source field in the incoming message is trustworthy because it is filled in by the kernel.) When running remote, the MINIX file server maintains state information, like RFS and unlike NFS.

The MINIX file system is similar to that of V7 UNIX. The *i*-node is slightly different, containing only 9 disk addresses instead of 13, and only 1 time instead of 3. These changes reduce the *i*-node from 64 bytes to 32 bytes, to store more *i*-nodes per disk block and reduce the size of the in-core *i*-node table.

Free disk blocks and free inodes are kept track of using bit maps rather than free lists.

;login:

The bit maps for the root device and all mounted file systems are kept in memory. When a file grows, the system makes a definite effort to allocate the new block as close as possible to the old ones, to minimize arm motion. Disk storage is not necessarily allocated one block at a time. A minor device can be configured to allocate 2, 4 (or more) contiguous blocks whenever a block is allocated. Although this wastes disk space, these multiblock zones improve disk performance by keeping file blocks close together. The standard parameters for MINIX as distributed are 1K blocks and 1K zones (i.e., just 1 block per zone).

MINIX maintains a buffer cache of recently used blocks. A hashing algorithm is used to look up blocks in the cache. When an i-node block, directory block, or other critical block is modified, it is written back to disk immediately. Data blocks are only written back at the next SYNC or when the buffer is needed for something else.

The MINIX directory system and format is identical to that of V7 UNIX. File names are strings of up to 14 characters, and directories can be arbitrarily long.

Layer 4 — User Processes

This layer contains *init*, the shell, the editor, the compiler, the utilities, and all the user processes. These processes may only send messages to the memory manager and the file system, and these servers only accept valid system call requests. Thus the user processes do not perceive MINIX to be a general-purpose message passing system. However, removing the one line of code that checks if the message destination is valid would convert it into a much more general system (but less UNIX-like).

Documentation

Since one of the purposes of MINIX is to provide a system that can be taught in classes and studied individually ample documentation is essential. For this reason I have written a textbook (719 pages) treating both the theory and the practice of operating system design. The bibliographic data is:

Title: *Operating Systems:
Design and Implementation*
Author: Andrew S. Tanenbaum
Publisher: Prentice-Hall, Inc.
Publication date: January 1987

The table of contents is as follows:

CHAPTERS

1. Introduction
2. Processes
3. Input/Output
4. Memory Management
5. File Systems
6. Bibliography and Suggested Readings

APPENDICES

- A. Introduction to C
- B. Introduction to the IBM PC
- C. MINIX Users Guide
- D. MINIX Implementers Guide
- E. MINIX Source Code Listing
- F. MINIX Cross Reference Map

The heart of the book is chapters 2-5. Each chapter deals with the indicated topic in the following way. First comes a thorough treatment of the relevant principles (thorough enough to be usable as a university textbook on operating systems). Next comes a general discussion of how the principles have been applied in MINIX. Finally there is a procedure by procedure description of how the relevant part of MINIX works in detail. The source code listing of appendix E contains line numbers, and these line numbers are used throughout the book to pinpoint the code under discussion. The source code itself contains more than 3000 comments, some more than a page long. Studying the principles and seeing how they are applied in a real system gives the reader a better understanding of the subject than either the principles or the code alone would.

Appendices A and B are quickie introductions to C and the IBM PC for readers not familiar with these subjects. Appendix C tells how to boot MINIX, how to use it, and how to shut it down. It also contains all the manual pages for the utility programs. Most important of all, it gives the super-user password.

Appendix D is for people who wish to modify and recompile MINIX. It contains a wealth of nutsy-boltsy information about everything from how to use MS-DOS as a

;login:

development system, to what to do when your newly made system refuses to boot.

Appendix E is a full listing of the operating system, all 260 pages of it. The utilities (mercifully) are not listed.

Distribution of the Software

Software distribution is being done by Prentice-Hall. Four packages are available. All four contain the full source code; they differ only in the configuration of the binary supplied. The four packages are:

- 640K IBM PC version
(eight 360K diskettes)
- 256K IBM PC
(no C compiler; eight 360K diskettes)
- IBM PC-AT
(512K minimum; five 1.2M diskettes)
- Industry standard 9-track tape

The 640K version will also run on 512K systems, but it may be necessary to change parts of the C compiler to make it fit. The tape version, in addition to the MINIX software, also contains a complete IBM PC simulator in C and other software that allows MINIX to be experimented with to some extent on a VAX or other time sharing computer, rather than a bare IBM PC. This option may be useful for courses for which IBM PCs are not available.

The book (\$34.95) and the software (\$79.95) are being issued separately. The book can be bought in any technical bookstore, or ordered specially. The book contains a postcard that can be sent back to Prentice-Hall to order the software.

A final word about the legal status of the code is in order. The software does not come with a 10-page licensing document that only the Dean of the Harvard Law School understands. However, it *is* protected by copyright. The software is *not* public domain. However, the publisher does not object to a limited amount of copying being done for noncommercial use. In other words, professors may make copies of the system for their students, students may make copies for their professors, and you may make copies for your friends. If you wish to port the software to another computer and then sell it, you need written permission from Prentice-Hall. In general they will be quite reasonable about granting such permission.

A Usenet newsgroup called `comp.os.minix` has been set up. This channel is being used by people wishing to contribute new programs, point out and correct bugs, discuss the problems of porting MINIX to new systems, etc. Although the publisher does not object to the network being used to broadcast a few files that have been improved, it is not intended to publish the full distribution (eight 360K diskettes) this way.

Acknowledgements

I would like to thank the following people for contributing utility programs and advice to the MINIX effort: Martin Atkins, Erik Baalbergen, Charles Forsyth, Richard Gregg, Michiel Huisjes, Patrick van Kleef, Adri Koppes, Paul Ogilvie, Paul Polderman, and Robbert van Renesse. Without their help, the system would have been far less useful than it now is.

The DASH Project: Design Issues for Very Large Distributed Systems

David P. Anderson

Domenico Ferrari

Computer Science Division

Department of Electrical Engineering and Computer Science

University of California, Berkeley

Berkeley, California 94720

Introduction

A *very large distributed system* (VLDS) is a (currently hypothetical) system that:

- is large in the following senses: *numerical* (it contains thousands or millions of connected hosts), *geographical* (hosts may be thousands of miles apart), and *administrative* (the system encompasses hosts and networks owned by many organizations and individuals).
- offers access to non-local resources such as databases, processing power, software, and communication with remote human users.
- is *transparent* in the senses that 1) at some level (perhaps the user interface) the same syntax can be used to access both local and remote resources, and 2) there is little performance difference between local and remote access.

Such a system is preferable to a collection of connected but unintegrated local distributed systems because it allows efficient resource sharing on a much larger scale. However, a VLDS cannot be realized by extending an existing distributed system, because many of the assumptions underlying the designs of these systems do not hold for VLDS. Fundamental differences exist in the areas of security, naming, communication paradigms and architectures, and kernel architecture.

VLDS design involves close interaction of levels ranging from network hardware up to the user programming model, and optimal solutions cannot in general be reached by extending techniques developed for small distributed systems. Furthermore, a VLDS has significant advantages over unintegrated collections of LAN systems, and a properly-designed VLDS restricted to a LAN is potentially as efficient as a

specialized LAN system. For these reasons, VLDS design should be viewed as a distinct and important research area.

Previous projects have considered VLDS-related problems such as scalable nameservers [3,6,7] and file services with many clients [4]. Efforts at large-scale integration of existing centralized systems are described in [8] and [5]. These projects, for the most part, address restricted problems or develop solutions based on technology that will soon be outdated.

In contrast, the DASH project at UC Berkeley is taking a unified approach to VLDS design, and is seeking solutions that will not be made obsolete by foreseeable technology advances. The goals of the DASH project are to 1) project the advances in computing and communication hardware that will make a VLDS feasible, and the software systems and applications that will be possible in a VLDS; 2) identify a set of design principles for VLDS, 3) propose mechanisms based on these principles, and 4) develop a prototype VLDS that can be used to test and compare these mechanisms.

Principles and Research Areas

The following are some of the principles for VLDS design that we have arrived at; a more complete discussion can be found in [2]. The DASH prototype incorporates all of these principles.

- Separate the levels of network communication, execution environment, execution abstraction, and kernel structure, and provide an open framework where possible.
- Use a hybrid naming system using a tree-structured symbolic naming for global permanent entities, and capabilities to communication streams for other entities.

;login:

- When possible, put communication functions such as security and interface scheduling at a *host-to-host* rather than *process-to-process* level, and consolidate these functions in a *sub-transport* layer.
- Provide flexible support for stream-oriented communication.
- Provide a service abstraction that allows for replication, local caching and fault-tolerance, but does not directly supply them.
- Support real-time computation and communication at every level.

Our discussion raises a number of research areas that should be investigated before a VLDS is put in place:

- Exploring the limits of size and granularity in distributed computing, and identification of possible bottlenecks (process creation, name resolution, authentication, network latency).
- The design of high-performance servers for distant access, and in particular the use of replication, streaming, and caching.
- Utilization of multiprocessors: how important is kernel parallelism, and how does performance depend on processor scheduling, locking mechanisms, and locking granularity?
- Assessing the usefulness of bundle-like stream mechanisms for long-distance high-performance services.
- Investigation of real-time network performance in terms of its implications for network and sub-transport layer design, its implications for process scheduling, and its possible applications.
- Exploring the limits of network performance with very fast networks, network interfaces, and I/O systems.

Project Status

The DASH project currently consists of 2 faculty members, 2 graduate students, and several undergraduates. The DASH prototype kernel is being implemented on Sun-3 workstations in C++. We are using UNIX as our development environment, and have made a

modified version of the UNIX dbx debugger that allows us to do remote symbolic debugging of the DASH kernel running on bare machines. The DASH kernel contains no UNIX code.

At this point (February 1987) the lowest layer of the kernel (processes and message-passing) and of the network communication facility (security mechanism and network drivers) have been completed. We have recently performed a study of the performance of our network security mechanisms, described in [1].

References

- [1] D. P. Anderson, D. Ferrari, P. V. Rangan and B. Sartirana, "A Protocol for Secure Communication in Large Distributed Systems," UCB/Computer Science Report No. 87/342, CS Division (EECS Dept.) UC Berkeley, February 1987.
- [2] D. P. Anderson, D. Ferrari, P. V. Rangan and S. Tzou, "The DASH Project: Issues in the Design of Very Large Distributed Systems," UCB/Computer Science Report No. 87/338, CS Division (EECS Dept.), UC Berkeley, January 1987.
- [3] A. D. Birrell, B. W. Lampson, R. M. Needham and M. D. Schroeder, "A Global Authentication Service without Global Trust," *IEEE Symposium on Security and Privacy*, 1986.
- [4] M. Satyanarayanan, J. M. Howard, D. A. Nichols, R. N. Sidebotham, A. Z. Spector and M. J. West, "The ITC Distributed File System: Principles and Design," *Proceedings of the 10th Symposium on Operating System Principles*, Operating Systems Review 19, 5 (December 1985), 35-50.
- [5] R. E. Schantz, R. H. Thomas and G. Bono, "The Architecture of the Cronus Distributed Operating System," *Proceedings of the 6th International Conference on Distributed Computing Systems*, May 1986, 250-259.
- [6] M. D. Schroeder, A. D. Birrell and R. M. Needham, "Experience with Grapevine: the Growth of a Distributed System," *ACM Transactions on Computer Systems* 2, 1 (February 1984), 3-23.
- [7] D. B. Terry, M. Painter, D. W. Riggle and S. Zhou, "The Berkeley Internet Domain Server," *USENIX Summer Conference Proceedings*, June 1984, 23-31.
- [8] T. Truscott, B. Warren and K. Moat, "A State-Wide UNIX Distributed Computing System," *Proceedings of the 1986 Summer USENIX Conference*, June 1986, 499-513.

;login:

Book Review

The Nutshell Handbooks

(Newton, MA: O'Reilly & Associates, 1986) \$7.50 each

Reviewed by Lou Katz

Metron Computerware, Ltd.

Oakland, CA

ucbvax!metron!lou

O'Reilly & Associates has created an ambitious set of small volumes intended to serve as a relatively basic introduction to a number of important UNIX[†] facilities. Uniform in style and presentation, there are currently seven books:

- #1 Learning the UNIX Operating System
Grace Todino and John Strang
- #2 Learning the Vi Editor
Linda Lamb
- #3 Reading and Writing Termcap Entries
John Strang
- #4 Programming with Curses
John Strang
- #5 Managing UUCP and USENET
Grace Todino and Tim O'Reilly
- #6 Using UUCP and USENET
Grace Todino
- #7 Managing Projects with Make
Steve Talbott

The first book of this series, *Learning the UNIX Operating System*, declares its intention to give a "good overview of ... UNIX survival materials for the new user," "not to overwhelm you with unnecessary details but to make you comfortable as soon as possible in the UNIX environment." In that respect, the book accomplishes its aim. It is well-organized and flows in a logical manner. The text is simply written, and should make a novice user comfortable. I like the small (8½" × 5½") format, which is compatible with many of the commonly available UNIX manuals.

However (now for the bad news), this volume is seriously flawed in detail. I get the very strong impression that its authors are new to UNIX, mastered the one system they had access to, and by virtue of being more facile

[†] UNIX is a registered trademark of AT&T. All sorts of other things are trademarks of other companies.

than their friends came to delude themselves that they were experts. This seems clear to me by omission, for surely anyone with any depth of experience would have gone to some pains to remark on the existence of different versions of UNIX and to specify (at least in an introduction or appendix) which version or versions this volume referred to. Although one can smile with wry amusement at the statement "ed was first developed when text editing was done on a line printer," one wonders whether the authors were born yesterday or just arrived from Mars. Statements of this sort certainly do not inspire confidence! In fact, this book is loaded with inaccuracies in details or concepts, flaws which can easily lead a new or naive user to formulate a fundamentally incorrect model of the system. There is a blurring and mixing of the distinction between UNIX the operating system, the shell user interface and other user-level commands which are supplied with a UNIX system. The assertion that "If you make a mistake in specifying the options (to a command) UNIX (sic!) will display the correct form" is both conceptually and factually incorrect. Some commands will give a "usage: syntax" response when faced with unknown flags or missing but required arguments, but many commands do not do so, and certainly UNIX isn't doing this service.

Almost all topics are explained in narrative and then at least one specific example is presented. This is very useful, and is reasonably well done, though there are enough errors in detail to make me worry. The overall presentation of I/O redirection is good and there is an excellent discussion of the danger of qrm. The discussion of cd has a good comment regarding the fact that you cannot cd to a filename, but follows that with an example which shows a system response which is incorrect. The description of ls -a

;login:

poses the common question "What are these files '.' and '..?'" explicitly, but then doesn't answer it. Furthermore, the parent directory of the user's directory is shown in an `ls -l` listing as being owned by the user himself, a highly unlikely organization of file system ownership.

Users are encouraged to try a terminal's BREAK key, among others, if they need the INTERRUPT function, a practice which usually will not work, and will often cause big problems. The much-needed section on what to do if your terminal seems hung or unresponsive contains a number of excellent suggestions, but omits crucial ones: to try pressing LINE-FEED or control-J!

Typographically, O'Reilly & Associates have chosen to use a rather poor sans serif font when displaying literal computer-user interactions. I would have preferred greatly the computer's prompts and responses to be bold-faced and the user's entries to be normal rather than the reverse scheme used. In the section on the need for white space between commands and their arguments, the example `ls-l` (without whitespace) was broken at the margin right after the `ls`, totally destroying the visual information of the missing space and the intelligibility of the example!

Although this volume is clearly and simply written and covers a very suitable selection of topics for novice users, it is much too flawed. I cannot recommend it.

Volume #2, *Learning the Vi Editor*, covers that common utility. Since I am not a vi user, I cannot vouch for its detailed accuracy. However, the discussion of cursor position on page 3 is totally confusing. The WYSIWYG statement on page 6 is wrong, and the table on customization is unclear. Some of the examples in the regular expression section, though correct, were more complicated than was needed: to delete all trailing blanks, they suggested `:g/\(.*\) *$ /s//\1/` while the expression `:g/ *$ /s///` would work just as well and didn't need the saved sub-string `\1`. However, the overall description seemed good enough that I expect to try use this book on the next occasion that I have to cope with a system which does not happen to have my favorite editor but does have vi.

Volume #3, *Reading and Writing Termcap Entries*, is written for system administrators, albeit ones with limited experience. It proceeds through the features and capabilities of the termcap system and gives expanded explanations in rather clear English. The "Terminal Capabilities by Function" table is a very nice restatement of the capabilities codes. The blow-by-blow annotation of the entries for two different types of terminals is good, as are the hints on how to write, test, and debug entries. The alphabetic list of capabilities at the end also serves as an index to the booklet. In a few places, the descriptions get bogged down — the written discussion of the '%' arguments being an example, though the following examples partly compensate for the confusion in the paragraph. In general this volume is a useful and helpful addition to the standard documentation.

Volume #4, *Programming with Curses*, the companion volume to #3, is a different story. I actually used this book during a recent project which required curses. The expanded explanations were a great help in dealing with the standard CURSES document in the UNIX reference manuals, but the illustrative examples were poor and badly explained. Detailed explanations of some of the functions were either inaccurate or misleading, leading me to believe that the author did not fully understand, among other things, the gory details of terminal I/O.

Discussion of the `clearok()` function was confusing and left me no wiser than before I read it. Physical echo does not mean that "characters are also echoed to the screen locally by your terminal." The definition of `crmode()` says that `^S`, `^Q`, `^C`, `^Y` go to the kernel for processing long before they are defined in the text as flow control, interrupt and quit, and there is NO mention of local variability and freedom in assigning these functions to the control key of your religious choice. "Raw mode is good if you do not want to handle interrupts and quits and would rather ignore them" is a rather bizarre and dangerous view of this topic.

In general, the usage examples are poor. For me, the greatest lack was in not coming to grips with using curses for overlapping windows. This topic was not treated in any useful fashion. Since curses is concerned

;login:

with the care and feeding of visual material, the lack of illustrations (each potentially worth 1000 words) is regrettable.

The Quick Reference Table at the end of the book did serve as a partial index. Although seriously flawed, this book was somewhat useful to me, and I would recommend it to experienced programmers, but only very cautiously to novices.

Volume #5, *Managing UUCP and USENET*, and volume #6, *Using UUCP and USENET*, are another pair of pamphlets, one targeted towards the system administrator, the other towards a user who may not even be a programmer. I became concerned at the outset with the technical breadth of experience of the author because I found on page 3 of Volume #5 the statement "we use the system prompt # to indicate that the commands can only be executed in the *superuser* mode because of UUCP file permissions. Commands that are preceded by the system prompt `unix%` can be invoked in multi-user mode." Anyone who confuses permissions with operating mode (I have never heard *single-user mode* referred to as *superuser mode*), or who cannot find the right words to express this concept, makes me wary, as does someone who believes the 50 foot limit on direct RS-232 connections. The discussion of null modems is too trivial (again lack of knowledge?), for a new system administrator really needs some discussion of modem control lines, not just how to cross send-data and receive-data wires.

There are many detailed tidbits which seemed strange, such as putting comment lines in `/etc/passwd` by starting a line with #, doing a `setuname` on every reboot from `/etc/rc`, or putting several entries for the same username but with different passwords in `/etc/passwd` and expecting `login` to find the one which matched. `getty` monitors incoming lines, and does not start a `login` when a call out is initiated. Setting the "time to call" field in `L.sys` does not `enable uucico`.

The output from `uucico` in debug mode is confusing and not really explained in standard UNIX documentation, and this book adds nothing to that situation. A two page reproduction of a session is dropped in the user's lap without so much as one line of comment, aside from dividing the printout into sections and identifying the major activity

underway (establishing contact, sending a file, etc). However, the table of STST and LOGFILE messages is good and useful.

As I have never had to install or maintain `netnews`, I cannot vouch for the accuracy of the chapters on that subject. What I read seemed clear and useful. Appendix A, which lists the names, formats and use of the `uucp` working files is a welcome addition of useful information on this cryptic system. Despite the glaring flaws there is value in this book, and I give it a qualified recommendation.

The companion volume #6 is meant for users, and suffers from the same defects as the other volumes. `UUENCODE` is not a `UUCP` command for sending binary files, and even the example shows encoding a file and then piping it through mail. One strongly suspects that much of the information that is presented was obtained by reading the documentation rather than by trying it.

The fact that tilde escapes like `~!` or `~%put` had to start on a new line was never mentioned. The author seems not to have known that the `~>`: file diversion facility of `cu` could be invoked without the trailing `:`, and so wastes considerable time explaining how to run scripts to see material, when `~>` enables you to see the input while capturing it in a file. If one never gets past the `~%take` and `~%put` options, one doesn't quite understand the full power of the `cu` program. As with the other booklets, the discussion of `netnews` is quite thorough, but I was tired of looking for defects by then. With the recent reorganization of the entire newsgroup naming conventions, some of the material is out of date, but that isn't the fault of the author. This book is a useful addition to the standard documentation, but don't go to any special trouble to find it.

Volume #7, *Managing Projects with Make*, is a rather comprehensive tour through the `make` facility. Written in an entirely different style from the standard UNIX documentation, it provides a clear and comprehensive description of the flags, options, and features of `make` along with examples. The booklet discusses the less than obvious syntax of `make` and its macros in some detail, and I learned a few things about maintaining libraries using it. My main criticism is that I would have preferred more explanation of why certain

;login:

constructs were used in the final rather elaborate example. All in all I would recommend this book, especially for the experienced programmer who may be working alone and without much contact with a make guru, and who need to get started using this facility.

In summary, O'Reilly & Associates have tackled many of the functions new or inexperienced users, programmers, and system administrators need to know in a coherent and unified fashion, and should be commended for their efforts. Each book is complete in itself and modestly priced (about \$7), though the cost for the entire set is no longer trivial. Each volume contains a Table of Contents at the beginning and Summary pages at the end, but none contains an index. The subjects are covered thoroughly and in detail and the writing is generally clear and simple. This is not just a re-phrasing of material found in the standard UNIX manuals.

However, when examined closely, the books are riddled with inaccurate, misleading, or downright incorrect technical details or concepts, so that their utility is in some cases severely compromised. In a discipline where literal accuracy is required for learning by example, this failing is inexcusable and cannot be taken lightly. The printing style, photo offset onto small format stapled booklets with inexpensive paper should make revision less costly, and many of the problems noted above are correctable. With the exception of volume #1, the other six booklets are all more useful than flawed, and can be recommended to a greater or lesser degree. The table below summarizes my ratings for each of the books. The rating scale is:

Unacceptable	0
Poor	1
Satisfactory	2
Good	3
Excellent	4

	#1	#2	#3	#4	#5	#6	#7
Clarity of Presentation	2	3	3	3	2	3	4
Organization and Style	3	3	3	3	3	3	4
Examples	2	3	3	1	2	2	3
Completeness	3	3	3	2	3	3	3
Accuracy	1	2	3	3	2	2	3
Overall Value	1.5	3	3	2.5	2.5	2.5	3.5

Minutes of the AUUG General Meeting February 10, 1987

1. The meeting opened at 09:16. Present were an undetermined number of members of the AUUG, and several others. The secretary, and one general committee member (Chris Campbell) were present.
2. In the absence of the president, Chris Campbell was elected chairman of the meeting.
3. The minutes of the previous GM, the 1986 AGM (Canberra, September 1986) were read.

Moved (John Carey, seconded Peter Harding) **That the minutes be accepted.**
Carried (with one abstention).

4. There was no business arising from the minutes, other than that already scheduled on the agenda.
5. No presidents report was given, as the president was absent.
6. The secretary (Robert Elz) presented his report. There were 148 ordinary members, and 70 unfinancial members still owed newsletters on Sep 1. We now have 5 institutional members. There were 24 newsletter subscribers who were not members. 10 newsletters are exchanged with other groups, or otherwise sent without payment. There are 144 members currently unfinancial.

The secretary made a brief report on progress with respect to negotiations with other user groups for reciprocal rights. These include USENIX /usr/group NZUSUGI JUS EUUG SINIX and THAINIX.

The secretary also mentioned the high number of unfinancial members, and indicated to members the method of determining their membership expiry date from the mailing label of the newsletter.

Moved (Peter Tyres, seconded Stephen Frede) **That the secretaries report be accepted.** Carried without dissent.

7. In the absence of the treasurer, the secretary presented the treasurer's report. A financial summary for the previous financial year, and budget for the current year were presented.

Moved (Michael Selig, seconded Taso Hatzi) **That the treasurer's report be accepted.** Carried without dissent.

8. There was considerable discussion on the issue of incorporation, and the motion moved at the AGM in Canberra instructing the committee to hold a ballot of members.

The chairman presented a brief summary of the issues.

An unfinancial member asked why the ballot had not been held, and attempted to move a motion that the ballot be held earlier than the committee's intended date of May. He indicated that he was an unfinancial member because of concern of the status of members in unincorporated associations.

The secretary indicated that legal advice was being obtained, and the results would be made available to members with the ballot when held.

The meeting expressed concern that incorporation be proceeded with as soon as possible.

It was also suggested that the committee plan to hold meetings soon after each general meeting, to avoid delays such as that which was incurred here, where there was no committee meeting for 2 months after the AGM, so the committee were unable to consider the direction for this period.

9. Meeting closed 10:13.

Letters to the Editor

Computer Centre
Monash University
Clayton, Victoria 3168
AUSTRALIA

Wednesday 18th February, 1987

Ryerson E. Schwark
Account Executive - Software Licencing
AT&T Unix Pacific Co., Ltd.
No. 1 Nan-oh Bld., 5th Floor
2-21-2, Nishi-Shinbashi
Minato-ku, Tokyo 105 JAPAN

Dear Sir,

I enclose the following correspondence between Greg Rose and myself concerning publishing AT&T literature in the AUUGN.

I am happy to publish press releases and related material concerning AT&T UNIX* products and activities as a service to the AUUG members.

It would be appreciated if AT&T would consider providing some form of support to the production of the AUUG's Newsletter. My most urgent need is to upgrade the version of *Documenter's Work-Bench* I am currently using to Version 2.0. At the moment I am struggling to keep up with articles produced with *grap* and the *mm* macros which respectively I currently do not have (*grap*) and have an old version (*mm*). Also a 3B2 or a UNIX-PC and a text previewer or a *fast* laser-printer dedicated to Newsletter production would be wonderful.

I look forward to your reply.

Yours Faithfully,

John Carey,
AUUGN Editor.

* UNIX is a trademark of AT&T Bell Laboratories



AT&T Unix Pacific Co.,Ltd.
No. 1 Nan-oh Bldg., 5th Fl.
2-21-2, Nishi-Shinbashi
Minato-ku, Tokyo 105 Japan
Tel : 03-431-3305
Telefax : 03-431-3680
Telex : J34936 ATTUP

March 11, 1987

John Carey
Computer Centre
Monash University
Clayton Victoria 3168
AUSTRALIA

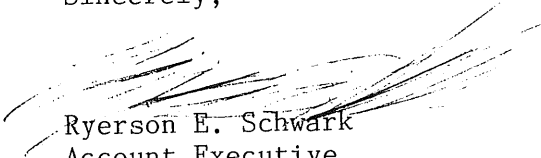
Dear Mr. Carey,

I have spoken with Mr. Crume about your request, and we have agreed that your request is quite reasonable. We are therefore going to give a free upgrade to Monash University from Documentor's Workbench 1.0 to Documentor's Workbench 2.0. This is, of course, under the understanding that it is done for the purpose of supporting AUUGN, and not as a gift to the university per se. The contracts involved in this upgrade will follow in a few days.

I understand your need for some computer hardware, but AT&T Unix Pacific does not sell hardware, so I am passing a copy of your letter on to Olivetti-Australia with the hopes that they might give you some assistance in this matter.

We appreciate your efforts in informing the UNIX* System community. If there is anything I can do to assist you, please don't hesitate to contact me.

Sincerely,



Ryerson E. Schwark
Account Executive
Software Licensing

Tokyo-Japan-RS-hy

* Registered Trademark of AT&T in the USA and other countries

Computer Centre
Monash University
Clayton, Victoria 3168
AUSTRALIA

Tuesday 21th April, 1987

Ryerson E. Schwark
Account Executive - Software Licencing
AT&T Unix Pacific Co., Ltd.
No. 1 Nan-oh Bld., 5th Floor
2-21-2, Nishi-Shinbashi
Minato-ku, Tokyo 105 JAPAN

Dear Sir,

Thank you for your generous offer to upgrade Monash University Computer Science's *Documentor's WorkBench* to Version 2.0 for no charge to assist with the production of the AUUG Newsletter.

I have passed your two letters and the contacts to:-


Sue Rees
Adminstrative Officer
Department of Computer Science
Monash University

for processing.

Unfortunately Olivetti Australia have not made any comment as far as hardware is concerned.

Thank you again for showing interest in the Newsletter.

Yours Faithfully,

A handwritten signature in cursive script that reads "John Carey". The signature is written in dark ink and is positioned above the typed name and title.

John Carey,
AUUGN Editor.



C.P. EXPORT PTY. LTD.
INCORPORATED IN VICTORIA
613 ST KILDA ROAD
MELBOURNE 3004
VICTORIA AUSTRALIA

April 9th, 1987

The Editor
The Australian UNIX Systems Users Group Newsletter
Computer Centre
Monash University
CLAYTON VIC 3168

Dear Sir,

I am writing this letter with the intention of informing members of the AUUG about the formation of CP Export, an international software publishing company whose major objective is to successfully identify, promote, distribute and sell Australian-sourced software on the international market.

By way of background, CP Export was formed in late 1986 as a joint venture between the Victorian Government and Computer Power, providing the software developer with a sizeable marketing, distribution and support presence in North America, Europe and Asia, thereby creating a ready made outlet for Australian products.

We are looking for Australian software products developed either by individuals or companies which will compete well in export markets, specifically in the UNIX area. We believe that some excellent products have been developed within this environment and as such are very keen to identify them irrespective of their size or functionality. The product developed by an individual to address a specific problem may well have significant potential.

If any of your members have a product which they believe is worth consideration or would like to have further information regarding CP Export, we would be pleased if they could contact CP Export, telephone (03) 520.5480.

Yours sincerely,

Brian R. Nicholson
Business Manager

AUUG

Australian UNIX^{*} systems User Group.

P.O. Box 366, Kensington NSW 2033, Australia.

auug@munnari.oz.au {seismo,hplabs,mcvax,ukc}!munnari!auug

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

Saturday 11th April, 1987

Mr Greg Rose,
Managing Director,
Softway Pty Ltd,
P.O. Box 305
Strawberry Hills
N.S.W. 2012

Dear Greg,

I refer to your letter to John Carey, the Editor, AUUGN, of January 27, in which you make a complaint about the publication in AUUGN Volume 7 Issue 2-3 of an advertisement for AT&T without charge.

I wish to point out that that particular advertisement was considered by the AUUG Management Committee at its meeting on November 17, and expressly authorised for publication at no charge.

This was reported in the minutes of the meeting, published in the same AUUGN issue. I refer you to item 11, on page 31.

Thus, if you have a complaint, it would be more correctly addressed to the AUUG Management Committee than to the Editor.

Yours sincerely,



Robert Elz
Honorary Secretary
AUUG

cc: The Editor, AUUGN

AUUG

Australian UNIX^{*} systems User Group.

P.O. Box 366, Kensington NSW 2033, Australia.

auug@munnari.oz.au {selsmo,hplabs,mcvax,ukc}!munnari!auug

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

Sunday 12th April, 1987

The Editor
AUUGN.

Dear John,

I enclose a paper from the NZUSUGI I received some time ago, together with a covering letter from the secretary of the NZUSUGI, Keith Hopper.

I would appreciate it if you would arrange to publish this paper in the next issue of AUUGN.

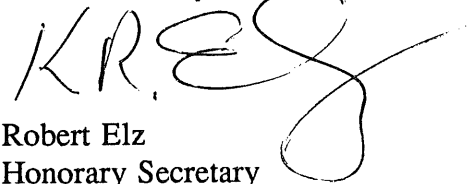
I was delaying this until I received a reply to a request I sent asking permission to publish this, however, after six months it appears that no reply is likely, so I think we should go ahead and publish it now.

I would also be grateful if you would request that any readers who have comments on this paper would forward them to me, either at the above address, or by electronic mail to

auug@munnari.oz

This will assist the management committee of AUUG to decide what position, if any, AUUG should take on this issue.

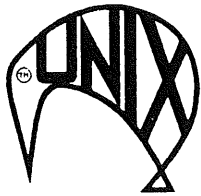
Yours sincerely,



Robert Elz
Honorary Secretary
AUUG

Enc

NEW ZEALAND UNIX SYSTEMS USER GROUP, INC.



P.O. Box 13056
University of Waikato
HAMILTON
New Zealand

8 October 1986

The Secretary
Australian Unix Systems User Group
P O Box 366
Kensington
NSW 2033

Dear Sir

UNIX STANDARDISATION

The Board of NZUSUGI has received conflicting suggestions about the use and applicability of the standardisation efforts of IEEE in the USA and the X/OPEN group in Europe.

I was asked to conduct a study of the documents and co-ordinate a position paper, a copy of which is attached.

A number of detailed technical discrepancies are also under consideration for incorporation in any interim standard.

Your comments are invited on this group's position in standardisation. These may be addressed as above.

Electronic mail may be sent via JANET to kh@nz.ac.waikato.

Yours faithfully



K. Hopper
Secretary

+

UNIX

STANDARDISATION

by

K. Hopper

University of Waikato, New Zealand

Abstract The ground rules for standardisation of the UNIX operating system are queried and one possible consistent set suggested. The place of the existing informal documents on UNIX standardisation is investigated and an outline plan for full formal standardisation proposed.

Introduction

The recent initiatives to "standardise" some facets of the UNIX operating system are unfortunately, on very shaky ground. They have arisen in Europe and the US for what are conceived to be urgent commercial reasons.

Before taking any action in regard to UNIX standardisation, however, there are a number of ground rules which should be agreed. Most of these are, at present, open questions.

Standards for Whom?

There are a number of viewpoints from which the need to standardise may be seen:-

- a. The commercial programmer wishing to invoke OS services.
- b. The commercial package user wishing to make use of a package.

+ UNIX is a trademark of AT&T Bell Laboratories in the USA and other countries.

- c. Any user wishing to use the UNIX tool-set.
- d. Any user wishing to interact with a UNIX system.

These four different "users" may, of course, be one person at different times.

Standardise What?

Is the attempt to "standardise" UNIX to be made from the point of view of the UNIX implementers - or is it to be made from the point of view of the eventual users (any or all of the kinds just described)? Whichever view is taken will result in quite different standards.

Standard in Relation to What?

All standards activities imply that there is necessarily some objective measurement facility which can determine conformance. This offers particularly intriguing situations when dealing with man-machine interactions. Is measurement to be made against some non-proprietary "universal"? Is measurement to involve grades of conformance - allowing sub-setting or super-setting? Is standardisation to be some minimum which may be exceeded by some implementer providing for "optional extras" - or are such extras to be eschewed?

Why Standardise?

Although all the work done in the US & Europe expressed the objective of improving software portability, there are a number of very important additional questions:-

- a. What about people portability?
- b. Why restrict portability to one OS?
- c. What about hardware portability? If not this then what about firmware portability? What about a "UNIX machine" very close to the raw hardware?

How Can a Standard be Specified?

In order to be able to test conformance it is a tautology to say that there must be a specification. However, any new standard must itself be specified in terms of some other "standard(s)" already defined.

The usefulness of a standard is strongly related to the range of people who understand what it means (in terms *solely* of what is "written") because they either know or have access to the standards *in* which it is written.

Who Tests Conformance?

This is the major question in setting standardisation ground rules. The easy answer is to merely say "standards authorities". When it comes to measuring against the International Standard Metre or Kilogram, say, little problem arises, since with due care it is possible to set up Transfer Standards which can be distributed to countries to produce in turn an arbitrary number of National Sub-standards.

Attempting to standardise an operating system requires that a set of standard tests are generated, which an implementation wishing to conform must pass satisfactorily - in terms of the written standard. However, the producer of programs for sale must also test that his program - which becomes an extension of the functionality of the computer system on which it runs - also conforms to the expected standard usage of operating system facilities by being an extension and *not* an alteration! This must be done with every program sold. It is unlikely that every individual programmer could afford to do this, therefore each country must have at least one test facility to be able to affix a "standards label" to every conforming program.

Does this imply the necessity of a Software Standards Laboratory in each country, such as is being done in relation to Ada, Cobol, Fortran, Pascal & Modula-2 programming language compilers? The compiler situation is relatively easy since very few people write compilers and only a sprinkling of test laboratories are required world-wide. Standardising software production of the kind envisaged in standardising UNIX requires a great deal more effort (and, presumably, cost).

Who Uses the Standard?

In order to be used, a standard must show advantage to its putative user community. This returns the discussion neatly to the first ground rule. Advantage may be measured in any appropriate way - but will, after a few transformations, almost always turn into a monetary value or, occasionally, only as a time-saver.

A little thought will show that the short-term immediate benefit arises to commercial programmers who will be able to sell their efforts to run on a wider range of machine hardware. The next most obvious beneficiary is the hardware supplier since, assuming that he offers a duly certified operating system implementation, he may compete more vigorously in a horizontal market knowing that his potential customers' software will not need changing.

The real beneficiary of all this in the long run is the end user who can mix and match software and hardware more readily to his current needs, without any need for retraining/learner costs interfering with his productivity.

THE ONLY WAY that this real benefit may be gained, however, is by THE USER PAYING in cash terms for the COST OF CONFORMANCE TESTING! If the user is not prepared to pay a testing premium then the standard might almost as well not have been written. If the user either can't or won't pay, the standard won't otherwise receive the use which it deserves.

Some Answers

Having posed a number of questions which aim at the ground-rules for standardisation, rather than the technical details which are very much a secondary consideration at this early stage, it is useful to present one set of possible answers as a starting point for discussion.

It is, therefore, proposed that the following ground-rules should be adopted -

- a. The viewpoint of the end user wishing to interact with a UNIX system should be chosen to express the needs which are to be satisfied. This viewpoint may be broadly described as a need for *maximum possible* transparency and *total* package portability, where transparency means that the user neither wishes to know about nor cares exactly what hardware is running which operating system implementation! Total portability is even more severe in that it implies that the user wishes to be able to use packages in identical ways *irrespective* of the run-time environment - ie hardware *and* operating system independent!!
- b. The things which must be standardised from the user's viewpoints are essentially all related to the OSCRL definition. It may be necessary to offer a variety of *mutually consistent* alternative interface processes until more progress is made in developing self-adaptive systems. The OSCRL should be a formally defined language - *not* the "list" of methods invoking commands syntactically which is seen currently in all UNIX "shells". The user wishes to see above all a uniform interface with no syntactic nor semantic peculiarities. The alternative interface processes are suggested in order to provide simple interfaces for those users needing them and more sophisticated facilities for the expert.
- c. A standard should be exactly and only one! There should be no sub-setting and certainly no supersetting - at least until formal revision of the standard provides for increased functionality. Any alternatives to this allow individual implementers to produce the existing Tower of Babel yet again.
- d. A standard should be presented in terms of the operating system interfaces.

(1) With a human user via some expressly provided interface process, built upon the following (2) interfaces.

(2) With other software non-human users in terms of primitive "command and response" languages.

(3) With the underlying "UNIX machine" at the firmware or hardware level. This is a vital interface all too often ignored.

Given the user's requirements for total software portability, the functionality of the primitive interfaces must be defined in **two** quite separate groups - those which are quite independent of what may be the underlying operating system and those which, by their nature, exercise operating system specific features.

e. Specifications should be written in terms of data abstractions - i.e. data structures and formally defined routines. No attempt should be made to use *any* "programming language" syntax for routine specification. The use of programming-language-like syntax to specify data structures *may* be convenient, while using formal routine semantic descriptions in, for example, VDM.

f. The computing industry in each country should set up (at least) one Software Standards Laboratory (where insufficient already exist), to provide the necessary independent conformance testing for programs written in accordance with a standard.

The Path Ahead

Two groups of workers in US and in Europe have produced respectively IEEE Trial Use Standard 1003 and the X/Open Portability Guide.

These large documents both address only a small part of a full standard - in a very primitive non-formal pseudo-programming language way. While these may be the result of many months of work - even years - they are not suitable as a formal standard - nor even part of one.

Their major disadvantage is that they are expressed in terms of the C programming language, which, besides not yet being given a standard definition of its own is totally unsuited to formal standard specification as it has no data abstraction built into its semantics (just into "good practice"!)

The path ahead therefore could well begin by completely rewriting them in formal notation as one document, to which is attached an informal descriptive explanation - rather in the way that the Ada language formal syntax and semantics are expressed in that standard. This preliminary step would also serve to separate the specific from the non-specific functionality required.

ISO work is being undertaken to specify human user OSCRLs. The human user interfaces for UNIX should take this work into consideration, as well as the associated theoretical work which has been undertaken by IFIP WG2.7.

The specification of a "UNIX machine" is something which has not yet been tackled in serious vein by any research or standards group. Specification of such a low-level functional machine is therefore open to anyone interested in starting such work.

Practical Steps

The preliminary informal specifications currently proposed should, it is considered, be used solely as an *interim* standard until formal specifications can be produced as suggested above. Bringing this into force will get the market accustomed to the benefits which even an interim standard can provide.

In parallel with work to produce a formal UNIX Software Interface Standard, work should be undertaken to produce a UNIX Human Interface Standard, in association with ISO TC97/SC21/WG5 standardisation efforts. Users of the existing UNIX interfaces should provide comment and suggestions for alterations through the Standards Officer of their national UNIX User Group. Particularly welcome will be comments from non-programmer users - who form the vast majority although they seem to receive too little attention!

It is hoped that implementers of UNIX kernels can join together, despite the existence of commercial secrets to produce in the longer term a UNIX Machine Standard, which will be of considerable use to hardware

manufacturers as "special" soft machines come to be used more than they are today.

Acknowledgements

Many thanks are due to my colleagues in the New Zealand UNIX Systems User Group who have encouraged me to devote more time to technical matters and less to administrivia. The poor current state of UNIX standardisation needs a great deal of help to correct the existing direction of work.

Bibliography

IEEE, *Trial-Use Standard Portable Operating System for Computer Environments*, (IEEE, New York, 1986).

X/Open Portability Guide, (Elsevier, Amsterdam, 1985).

IFIP WG2.7, *Reference Model for Operating System Command and Response Languages*, (Springer, Heidelberg, 1986).



AT&T Unix Pacific Co.,Ltd.
No. 1 Nan-oh Bldg., 5th Fl.
2-21-2, Nishi-Shinbashi
Minato-ku, Tokyo 105 Japan
Tel : 03-431-3305
Telefax : 03-431-3680
Telex : J34936 ATTUP

February 20, 1987

Enclosed is a press release given out by AT&T and Microsoft Corporation to introduce a new version of the UNIX* System V operating system for Intel Corporation's 80386 microprocessor.

If you have any questions, please do not hesitate to contact us.

Sincerely,

A large, stylized handwritten signature in black ink, appearing to read "Ryerson E. Schwark".

Ryerson E. Schwark
Account Executive
Software Licensing

Tokyo-Japan-RS-hy

* Registered trademark of AT&T in the USA and other countries.



AT&T Unix Pacific Co., Ltd.
No.1 Nan-oh Bldg., 5th Fl.
2-21-2, Nishi-Shinbashi
Minato-ku, Tokyo 105
Japan

For further information:

Ryerson Schwark
Software Licensing Account Executive
Tel: 3-431-3305 (Japanese)
3-431-3670 (English)
Fax: 3-431-3680
Telex: J34936 ATTUP
uucp:seismo!akgua!attunix!upshowa!schwark

PRESS RELEASE

AT&T and Microsoft Corporation today announced they will introduce a new version of the UNIX* System V operating system for Intel Corporation's 80386 microprocessor used in the newest generation of microcomputers.

The new software product -- giving PC's the power of a minicomputer -- will be developed by Microsoft from AT&T specifications and will be distributed under AT&T's trademarked name "UNIX." The product is expected to be available in early 1988.

Vittorio Cassoni, Senior Vice President of AT&T's Data Systems Division, said that "the use of the trademark "UNIX" on operating system products will assure customers that their applications will run unmodified on any computer based on the 80386 microprocessor, regardless of the vendor."

The 80386 microprocessor (commonly referred to as the 386) is the basis for the newest generation of 32-bit computers now being developed and marketed by many hardware manufacturers. The combination of the 386 and the multi-tasking, multi-user features of UNIX System V will be a powerful, cost-effective system for a wide variety of applications.

* Registered trademark of AT&T in the USA and other countries.

"While much of the computer systems industry is experiencing minimal growth, the market based on UNIX System V is accelerating," said Cassoni.

"We expect this version of UNIX System V, combined with the technological advances of the Intel 386 microprocessor, to create tremendous growth opportunities for both computer manufacturers and software vendors," he said.

He added that "the installed base of computers based on UNIX system software grew by about 75 percent in the past year," and that the current installed base is roughly 60 times larger than when UNIX System V was first introduced in 1983.

AT&T will continue to market UNIX System V, and Microsoft will continue to market its XENIX** operating system during the development of the new product. Applications written for Microsoft's XENIX System V and for UNIX System V -- including applications for AT&T's PC 6300 PLUS -- will run on the new implementation of the UNIX system without modification. When the new product becomes available, AT&T and Microsoft will market it as the sole UNIX system product for the 386.

"Compatibility with existing applications for XENIX System V and UNIX System V on the 80286 and the 80386 means that the new implementation of UNIX System V will be born with a large existing base of application software," said Bill Gates, Chairman, Microsoft Corporation of Bellview, Washington.

"Moreover, the establishment of a standard interface combined with the power of the 386 will attract new application vendors," he said.

"I am convinced that this announcement will clear the way for dramatic growth in the market for computers based on UNIX System V," Gates said.

** XENIX is a registered trademark of Microsoft Corporation.

The new implementation will provide software developers with a standard application interface while allowing them the option of using either AT&T or Microsoft software development tools. Computer manufacturers and software vendors will be offered licenses to distribute products based on the new implementation.

AT&T's UNIX System V is a standard portable operating system, available on a wide variety of machines from PC's to mainframes. Microsoft is a licensee of UNIX system software and is one of AT&T's largest distributors of UNIX system-derived products under its XENIX operating system.

Microsoft Corporation develops, markets, and supports a wide range of software for business and professional use, including operating systems, languages, and application programs, as well as books and hardware for the microcomputer marketplace.

AUUG

Membership Categories

Now that the Australian UNIX systems User's Group has existed a while, its time that all members reviewed their membership types, and even more, checked that they are in fact still members!

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 over 2 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.

If your membership has already expired, or is about to expire (many expire in January) then now is the time to renew.

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.

Robert Elz

AUUG Secretary.

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:

Office use only:

Chq: bank _____ bsb _____ - _____ a/c _____ # _____

Date: __/__/__ \$

Who: _____

Memb# _____

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 to:

AUUG Membership Secretary
P O Box 366
Kensington NSW 2033
Australia

Foreign applicants please send a bank draft drawn on an Australian bank, and remember to select either surface or air mail.

..... does hereby apply for

- | | | |
|--------------------------|--|----------|
| <input type="checkbox"/> | Renewal of existing Institutional Membership | \$250.00 |
| <input type="checkbox"/> | New Institutional Membership of the AUUG | \$250.00 |
| <input type="checkbox"/> | International Surface Mail | \$ 20.00 |
| <input type="checkbox"/> | International Air Mail | \$100.00 |

Total remitted

AUD\$ _____
(cheque, money order)

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.

I understand that I will receive two copies of the AUUG newsletter, and may send 2 representatives to AUUG sponsored events at member rates, though I 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 complete the other side.

Please send newsletters to the following addresses:

Name:
Address:
.....
.....
.....

Name:
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 usually 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

Office use only:

Chq: bank _____ bsb _____ - _____ a/c _____ # _____

Date: ___/___/___ \$

Who: _____

Memb# _____

AUUG

Application for Newsletter Subscription Australian UNIX* systems User 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
PO Box 366
Kensington NSW 2033
Australia

Please don't send purchase orders — perhaps your purchasing department will consider this form an invoice. Foreign applicants please send a bank draft drawn on an Australian bank, and remember to select either surface or air mail.

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 \$ 50.00
- International Surface Mail \$ 10.00
- International Air Mail \$ 50.00

Copies requested _____

Total remitted AUD\$ _____

(cheque, money order)

Tick this box if you wish your name & address withheld from mailing lists made available to vendors.

Office use only:

Chq: bank _____ bsb _____ - _____ a/c _____ # _____

Date: ___/___/___ \$

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
PO 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# _____