AUUGN

Australian Unix systems User Group Newsletter

Volume 6
Number 4

Registered by Australia Post Publication No. NBG6524

The Australian UNIX* systems User Group Newsletter
Volume 6 Number 4
January 1986

CONTENTS

* UNIX is a trademark of AT&T Bell Laboratories.

Editorial

Well, issues of AUUGN seem to be coming out later and later, even though I have the strongest desires to correct the situation. The end of 1985 was unbelievably hectic for myself and the newer editors. I am still waiting for our books editor to send his thrice promised copy, but then he is in the printing business and wants to make a buck at Christmas. The demise of Neology did not help either, as the AUUGN database was living there at the time. It spent some weeks on a tape before being resurrected onto the Computer Science Department's Pyramid. We thank them for their continuing assistance.

In this issue I welcome a new contributor, Ken Frame from Darwin, who will be presenting a series of Awk tutorials. Any comments on the series should be addressed to me and I will forward them to Ken.

We are still looking for people to become involved with the production of AUUGN so if you are interested, contact me as soon as possible. The address and phone number appear on the last page of this issue.

Memberships and Subscriptions

Membership and Subscription forms may be found at the end of this issue and all correspondence should be addressed to

Greg Rose
Honorary Secretary, AUUG
PO Box 366
Kensington NSW 2033
Australia

Next AUUG Meeting

By now everyone on the mailing list should have received information about the next meeting in Western Australia. In case you did not, more information may be found at the end of this issue.

Contributions

Come on you people out there in UNIX-land, send me your views, ideas, gripes, likes or whatevers.

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

UNIX Security

John J. Mackin
Basser Department of Computer Science
University of Sydney

john@basser.oz
seismo!munnari!basser.oz!john

I'd like to begin this month's column by recalling the hole that I presented last time: the fact that expreserve, which is setuid-root as distributed, uses popen to invoke mail. I received some mail from Norman Wilson at Bell Laboratories in which he pointed out that a tempting "fix", changing the first argument of the popen to "/bin/mail", fails badly. All that needs to be done then is to arrange for IFS in the expreserve's environment to contain a slash and call the program to be executed with root's uid "bin" instead of "mail".

An important thing to keep in mind, when programs that are going to be run with root's uid are being written, is the amazing generality and flexibility of the shell. In fact, I personally believe that no setuid-root program should invoke the shell (unless it setuid's to someone else first). This means avoiding popen, system and execvp/execlp, as well as less commonly available functions. Be aware of what library routines on your system invoke the shell.

It is also very important to know, if a shell is going to be invoked, what shell it will be. Is it guaranteed to be /bin/sh, or is the SHELL environment variable going to be used? The latter is obviously a recipe for disaster, and is another reason that privileged programs should never use library routines that call shells.

Here at Basser we were recently forcibly reminded of the danger involved in invoking shell scripts with root's uid. We run a locally developed mail delivery system that provides mail aliasing in the following manner: if a username to be delivered to doesn't exist in the password file, a file by that name is looked for in /usr/mail, and if one exists and has executable mode, it is executed with the mail item placed on its standard input. This is commonly used to implement simple aliasing, using shell scripts; for example, if the file /usr/mail/foo contained

```
#!/bin/sh
PATH=/bin:/usr/bin
exec /usr/lib/remail jim tim tom
```

then the command "mail foo" would result in delivery to the three named persons. ("remail" is a link to the deliverer.)

However, there are also more complicated aliases in use; some of these are historical, dating from the time when there was no network spooling software in use here and the way to print on a given printer that wasn't attached to the host one was on was to send mail to, say, "bpr" for printer "b". Here is /usr/mail/bpr:

```
#!/bin/sh -p
PATH=/bin:/usr/bin
SENDER=`expr "$1" : '-f\(.*\):.*'`
MACHINE=`expr "$1" : '-f.*:\(.*\)'`

if [ -n "$SENDER" ]
then

        if [ $SENDER = root ]
        then
                case $MACHINE in
                graphics|basser|basset|sunrise|basser40)
                        `basename $0` -i$SENDER:$MACHINE
                        exit 0
                        ;;
                esac
        fi
else
        `basename $0` -i$SENDER:$MACHINE
        exit 0
fi

LNAME=`basename $0` export LNAME


(
        cat <<!
Subject: remote printing

This is not the way to netprint.  Message follows:
!
        sed 's/^/       /'
) | mail $SENDER:$MACHINE
```

Now, this would have perfectly safe, were it not for one small hole. You might like to try to determine what it is before reading further. (The #! mechanism on our system is like the Berkeley one; the single string following is supplied as an argument to the invoked command, and the -p flag to our shell resets IFS and removes exported shell functions.)

So, why was that unsafe? Well, it is all very well to write PATH=/bin:/usr/bin, and certainly none of the commands named in that script would be searched for anywhere else. But, on our system, "basename" is a shell script too; and it blithely invokes "expr", so all it takes is an expr somewhere in the cracker's path and they are home free. (And they were!)

I think the message is clear. Shell scripts with root privileges can be very dangerous. Certainly one should weigh the convenience against the risk before implementing any such on a UNIX system where security is of even minimal concern.

A W K T A W K

Ken Frame

This is the first of a series of articles dealing with the command 'awk'. Awk has been accurately defined as a "pattern scanning and processing language" but this definition indicates little about its power and versatility. Most UNIX primers deal with awk from a keyboard command or shell command approach. In the commercial field of operations, awk files play a much more important part in operations than do awk processes in shell programs. Therefore most examples in this series will be in the form of an awk program file - the awk tool. In turn this requires a data file to provide the program file with materials to be developed by the tool. This approach does not detract from the important place that shell processes have, but seeks to identify the appropriate tool for a particular job.

What does awk do?

Awk can do anything! - well almost anything. Awk can reproduce text, rearrange it, print it backwards, word for word or even character for character if you want it that way. Arithmetic calculations are handled with ease, and awk contains a number of built-in functions for those whose needs are more than the basic operations.

There are 3 main segments of an awk program, the beginning, the body and the end. The beginning and end segments consist of one statement each which commence with the words BEGIN and END. The body consists of one or more statements. It is not mandatory to have BEGIN and END statements and in the example below, these are not used. Statements may vary in length so that it is possible for a BEGIN or END statement to be quite extensive although neither can extend beyond one statement.

Awk performs its duties by 'pattern scanning and processing'. Each statement has two parts. The first part determines which sections of the scanned data match the pattern and are to be processed. The second part describes the process that is to be executed. The first part, known as the pattern is optional. By default, the whole input line becomes the pattern if there is no specified pattern. Thus an awk statement consisting of

```
{ print }
```

would print each complete line of data input.

In this article I present a very simple awk program with describes the mathematical possibilities. Consider the position of a retailer who buys in case lots and sells by the unit. The supplier's invoice for the liquor department might follow the following lines:

item	code	bottles per case	cost per case

The retailer would input the data, as in the following example.

Johnny Walker Black label	jwb	12	250

He designs a program called grogawk to manipulate the above line (grogdata).

```
{ ident = $1; btc = $2 ; each = $3                    #line 1
print ident,btc,each                                  #line 2 (delete later)
coins = int((each * 1.3 / btc + .5 )* 2)              #line 3
print coins                                           #line 4 (delete later)
btsell = sprintf("%6.2f",(coins / 2 ))                #line 5
print btsell                                          #line 6 (delete later)
csell = sprintf("%4ld",(btsell * btc * 0.95 +1)/1)    #line 7
cost = sprintf("%7.2f",each)                          #line 8
print  ident,cost,csell,btsell }                      #line 9
```

Note that there is no pattern in this program. This is seen by the first character being "{" which signifies the start of the processing section of the statement. The statement consists of 1 or more sub-statements. A sub-statement terminates with a semi-colon or new-line. Some texts state that these characters together with the right brace "}" terminate statements, but it is safer to recognise the pair of braces in a more important position. In fact the statement is only terminated when the matching right brace is added. An attempt to terminate a statement by any other character results in an error message. Therefore I tend to think of braces enclosing statements with semi-colons and new-lines separating sub-statements.

Now to analyse the program grogawk and the command

awk -f grogawk grogdata

Grogawk reads in data, a line at a time. Grogdata supplies the first, and in this case only line which grogawk processes as follows:

```
{ ident = $1; btc = $2 ; each = $3                    #line 1
```

The left brace starts the processing; assign the contents of field 1 to ident, field 2 to to btc (bottles per carton) and field 3 to each (cost per case). This line is not essential as the fields can be identified by their position. However it is often easier to design a program using identifiable variables than field numbers. It is certainly easier to amend a program that does not contain a mass of $1,$2,$3, etc. and similarly if this program was to be adapted to a similar operation, the use of variables makes the adaptation easier.

```
print ident,btc,each                                  #line 2 (delete later)
```

this line was inserted to test the program during the trial period. The output should mirror the input. Similarly, the line could have read print $1,$2,$3 for the same result. If all that was required was the reprinting of the entire line, a more appropriate command would be "print $0" because $0 identifies all of the fields in the line. In fact this may even be shortened to "print" which again means print the line. As the purpose of this line is to determine the contents of the variables ident, btc, and each, it is better to use the statement as in line2. Note that each line contains a comment which commences with the character #. Comments are not mandatory but they do assist in describing the logic or purpose of the line. In this case, line 2 may be deleted when the program is running correctly.

Our merchant's pricing policy is that all items have a sale price that is a multiple of 50c. His awk program is designed to reflect this. Line 3 performs a number of operations, namely, adds a 30% markup (*1.3), divides the case price by the number of bottles per case (/btc) and calculates the number of 50c pieces in the selling price. Not wanting to reduce his 30% margin, he adds one 50c (+ .5) to his calculation which is still in dollars and cents and multiplies this by 2 (*2) to count the number of coins. This in itself does not delete the decimal portion of the result but by calling for the value to be expressed as an integer (int) he now has the rounded up number of coins making up the sale price. The int function always truncates the value hence the need to add 0.5 before performing the function.

```
coins = int((each * 1.3 / btc + .5 )* 2)              #line 3
```

Line 4 prints the result of the operation, namely 55 which is produced by

```
250 * 1.3 = 325
325 / 12 = 27.0833
27.0833 + .5 = 27.5833
27.5833 * 2 = 55.1667
int (55.1667) = 55
```

```
btsell = sprintf("%6.2f",(coins / 2 ))                    #line 5
```

Line 5 calculates the price per bottle, namely $27.50 which is the nearest rounded up multiple of 50c from 27.0833. Unwanted digits are removed from the result by prescribing the print format. In this case the format %6.2f requires a value occupying 6 positions with 2 to the right of the decimal point. This right justifies the value which makes it suitable for printed output.

```
csell = sprintf("%4ld",(btsell * btc * 0.95 +1)/1)        #line 7
```

Our merchant will also sell by the case and allows a 5% discount but again wishing to maintain his margins arranges his price so that instead of a 50c increment, he now deals in whole dollars.

Line 7 can be read as

case sell = the price in 4 digit integer, (bottle price * number of bottles to the case - 5% rounded up to the next dollar).

As in other programming languages, positioning of parentheses determine the order of the operators. In this case, the +1 occurs before the /1.

The program could have used the int function to obtain the same result, but to do so, provides no control over the format of the output. By specifying %4ld all case prices are right justified within a format of 4 character positions.

```
cost = sprintf("%7.2f",each)                              #line 8
```

Finally our retailer likes to have the actual cost handy so that he can wheel and deal, so specifies that the "each" data input at line 1 is printed out in a 7 character position format

```
print  ident,cost,csell,btsell }                          #line 9
```

The final line displays the result:

```
jwb  250.00  314  27.50
```

The ident, the cost, case selling price and bottle selling price are in the specified formats thus providing columns without having to order them by tabs. The statement is terminated by the right brace.

It is likely that if our retailer was to look at that output at a later time, he would have difficulty in identifying what it meant or when it was produced. Headings, conclusions, dates and other identifiers are essential parts of programming. The role of BEGIN and END statements was mentioned briefly in this article. Can you suggest ways that the output could be made more meaningful? This is something we will consider in the next article.

Cost effective performance improvement of file name lookup

Greg Rose

Neology Ltd.

ABSTRACT

UNIX[1] system V.2 was ported to the ELXSI 6400 multiprocessor computer in 1984. As part of this project, performance profiles of the kernel were produced and examined, and one quarter of the time spent in the kernel was found to be spent in the single routine called *namei*. This paper examines some issues raised by the relation of various data structures involved. Namei is the routine called to trace pathnames within the hierarchical UNIX file system. The performance of this routine was improved by a large factor by minor modification to two parts of the routine. A new method called *corroborative hashing* is used in one part of the performance enhancement. These changes implemented an associative memory for segments of pathnames in regular use, and searching from the last match within a directory. An associated change, made necessary by the ELXSI architecture, but desirable in general, involves fetching and validating the entire pathname from user memory in one operation, avoiding character-by-character lookup.

1. Pursuing Paths under UNIX.

The file system structure of UNIX has been described in many papers. In essence, it is hierarchical, with *directories* being a special type of file which can contain other files, even other directories.

A *name* is an entry in a directory somewhere, referring to a file. A more traditional word (at least to a UNIX student) is a *link*. Typically, a file has only one name. This is not automatically the case however; many files (including all directories) have more than one, and some have no name at all! When the file has more than one name, they may or may not appear in the same directory.

A *pathname* is a sequence of such names, separated by the slash character '/', where all but the last of the files named must be a directory. A pathname specifies not only the name of the file, but the directory it is in; not only the name of that directory, but the directory that is in, and so on, potentially for many levels.

There is a notion of a *current directory*. A pathname is relative to the current directory, unless it begins with a slash, in which case it is absolute (which really means relative to the unnamed root directory).

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

For example, the pathname **/usr/bin/dc** can be interpreted as the file **dc**, residing in the directory **bin** within the directory **usr** inside the root directory. The pathname **paper** represents the file named **paper** which resides in the current directory. It is worthy of note that current directory relative paths are typically degenerate, in that they usually do not have any intervening directories. The opposite is usually true of absolute pathnames, which is fortunate or the root directory would get very full.

When a command is typed under UNIX, there are a number of standard directories which are searched for the file containing the image of the program. One such is usually **/usr/bin**, and the UNIX command **dc** works by finding the file **/usr/bin/dc** and running the contents.

In many operating systems, there can be only one name for a given file, and so all of the information about the file can be stored in the directory along with the name. The information to be stored usually includes the length of the file, where it is on the storage medium, date of last access, who owns it, and other such information. Under UNIX, this cannot be done, as the overhead of keeping multiple copies of such information up to date and consistent is prohibitive (not to mention silly). Instead, the directory stores only the name, and a pointer to the collected information about the file. This aggregate of information is called an *inode*, short for information node.

The process of finding a file, which is now seen to be the process of finding its inode, is simple. Starting at the root directory inode or the current directory inode, both of which are known to the system, search the directory for the next name in the path, find from the entry its inode pointer and get the inode, and keep doing this until the end of the path is reached.

The process is not quite as simple as it may seem. In standard System V release 2 version 2 of UNIX, the characters are fetched into the kernel from the user program's memory one at a time, with great overhead. Then when a name has been gathered, as detected by finding a slash or the end of the pathname, the directory in question is searched linearly from the start for a match. As directories grow bigger, this becomes more of a problem, but never enough of a problem to justify imposing a complicated structure on the directory itself. (Note: the simple structure of directories is known by many programs. Changing this simplicity is not necessarily worthwhile.)

2. Fetching characters faster.

The most glaring inefficiency in the scheme as outlined is the laborious character checking. Unfortunately, the mechanisms for passing the file name to the kernel in the first place are CPU dependent, and so hastening the process must be CPU dependent to an even greater degree.

On the ELXSI 6400, the only interprocess communication is through the message system, and the only feasible way to pass the file name is to put the whole thing (or at least most of it) into a message and sent this to the kernel. Since the name comes to the kernel prepackaged and validated, it was straightforward to make namei's access to the pathname very efficient, just stepping a pointer through the message buffer.

No matter how a particular CPU accesses the pathname in user memory, changing namei's character by character fetching, and even eliminating a function call per character, is a very worthwhile saving.

3. Economical name searching.

The other inefficiency in the process of locating a pathname is of course the sequential searching of directories. Examination of a typical case is worthwhile. Consider the invokation of the command **ls -l**. There are two distinct phases involved. First, access to the file **/bin/ls**, and secondly access to the files in the current directory in the order in which they occur.

Obviously, the name **bin** within the root directory is accessed very often. Indeed it exists only to hold those programs which are accessed often. The name **ls** within that directory is also frequently accessed, as it is one of the most common commands. This observation leads to the conclusion that remembering names and associated directories for a time, and having quick access to that memory may be an improvement.

Once **ls** has been loaded and is running, it begins to access the files in the current directory sequentially. Most of these will not have been accessed recently, so any sort of associated store will not speed access. On the other hand, since the directory itself is a sequential structure, most programs process them sequentially. This means that searching the directory from the beginning each time will result in accesses to name entries totalling one-half of the number of entries squared. This is the expected result of accessing all of the files independently of the order of the accesses. Information about the order of the accesses, especially when it is associated with the structure of the directory itself, can be used to speed access. In this case, an obvious place to begin the next search is at the point where the last one ended.

In the next section some points of philosophy are discussed, before actually describing implementation details. By far the hardest part of the implementation was the two days spent on philosophy, as the subsequent coding took less than a day, and of the one hundred lines or so coded, almost half was for the gathering of performance statistics presented below.

3.1 On the relation between names and inodes.

The crucial task demonstrated above is that of finding an inode, given a name and the inode of a directory in which the name resides. There is a data structure which associates these items of information: it is the directory itself. This is the directory's *raison d'etre*, and so any technique which also connects names to inodes will give serious affront to the directory. Anthropomorphosis aside, the maintenance of parallel data structures almost always causes problems. These problems include ensuring mutual consistency, deadlocking when multiple accesses must be prevented, and extra work in performing a simple update.

The directory establishes the relation between a name and an inode. What is really desired is not another data structure to do the same job, but one which will find the

appropriate directory entry quickly. Since the directory is the only arbiter between names and inodes, there can be no problems of consistency and the like.

Another point to note is that since namei is the only routine which searches for directory entries, any additional data structure involved in finding a directory entry faster can be entirely local. If another data structure involving both names and inodes is used, it must perforce be kept up to date wherever a directory is accessed. While the number of such places is small, it is still significantly greater than one.

The conclusion reached at this stage was that a data structure should be introduced which associated in some way the directory involved, the name, and the resulting *directory entry*. This then results in finding the inode by using the directory entry in the normal manner.

3.2 On the necessity for accuracy in an heuristic.

The data structure mooted above is intended to provide information leading directly to the directory entry corresponding to a given name. The cost of ensuring absolute accuracy of the structure involved is quite high. If a name is to be removed, this is performed elsewhere. If a name is to be created then first a place to put it must be found, then after further validation the name can be inserted; again this operation is performed outside namei.

Suppose on the other hand that the information provided by this auxilliary data structure is not always accurate. The cost of this assumption is just that the information yielded, that is the directory entry involved, may not be the right one. If it is not correct, then the directory must be searched in the normal manner, just as it would have been searched if the auxilliary structure had failed to yield a directory entry at all. So long as the *probability* of the answer being wrong is low, the cost of it being wrong is going to be negligible, as code was already in existence to handle other similar cases.

It was decided that it mattered very little that the data structure may contain inaccuracies, so long as their frequency was small. Even the presence of inaccuracies merely degrades performance to what it was in versions of the software without the associative data structure.

A suitable technique for implementing the proposed data structure is some form of *hashing* (often called *key transformation*). In a normal hashing implementation the information supplied - in this case the directory and the name - is transformed into a small integer which is used to index some sort of table quickly. The data item thus yielded must be validated against the given input, as the transformation used must be a many to one mapping. Once the validation is passed, the desired result is read from the stored item. When the validation fails, that is the key transformation yielded the same result as that of a different item stored in the table, a *collision* results. There are many methods of handling collisions, not discussed here.

A second key transformation can usually be computed at the same time as the primary one. Often, the cost of computing a second key transformation is small compared to

the cost of checking that the item found is the correct one. This often occurs when only part of the input information is used in the key transformation, but all of it is required for validation.

When (as is the case under discussion) the requirement is for a high probability of getting the right answer, not absolute necessity, and a second key transformation function can be computed cheaply, a new technique may be used. This is called corroborative hashing. In this the primary key transformation is used to locate the desired data item. The secondary key transformation is then compared with the one stored in the entry, as a low cost indicator of whether or not a collision has ocurred with a high probability of correctness. Note that it is not in general possible for the second key transformation to guarantee that the item found is the one desired.

This technique offers performance savings in three ways. First, the secondary key transformation is usually cheaper than comparing the entire key information to validate a potential match. Secondly, the table used can be much smaller since the key information does not need to be stored. Thirdly, updating the table does not involve copying a large amount of key information, rather just the secondary key transformation which was calculated.

Additionally, it is viable to handle collisions by having multiple sets of data, in the manner popular with hardware cache designers. This prevents bad behaviour caused by two very commonly accessed items happening to share the same primary key transformation, without the overheads of more complicated collision handling schemes.

4. Implementing the performance improvements.

This section details the three phases of implementation of the improvements. Code to gather statistics from the modifications was incorporated from the beginning, but is not truly a part of the modifications themselves, as the improvement is gained independently of the statistics. The total modifications amounted to eighty six lines of C code, almost entirely concentrated within namei, and declaration of a few new data items.

4.1 Searching from any place in a directory.

The algorithm in namei originally searched from the beginning of the directory to the end of the directory or until the desired file was found. A simple modification, and the introduction of another variable, changed this to searching from a given offset until either the desired file was found, or the initial offset was reached again. At the end of the directory, the search wraps around to the front again.

As added protection, presupposing the next stages of implementation, an initial offset which is outside the bounds of the directory is reset to the beginning of the directory.

This modification required ten lines of C code, three of which were comments.

4.2 Starting where the last search ended.

The next stage of implementation was aimed at the cases where the file had not been accessed recently, but the directory had been searched. If the contents of the directory are being accessed sequentially, then the desired file will be found soon after the last one accessed.

Another field was added to the inode in memory (there was no good reason to put this information on disk, and many good reasons not to change the structures on disk at all). The new field stored the last search offset found by namei. One line of code sufficed to initialise this field when the inode was read in from the disk.

The functional change was trivial: when searching a directory, and in the absence of a better hint from the next modification, start searching at the remembered offset.

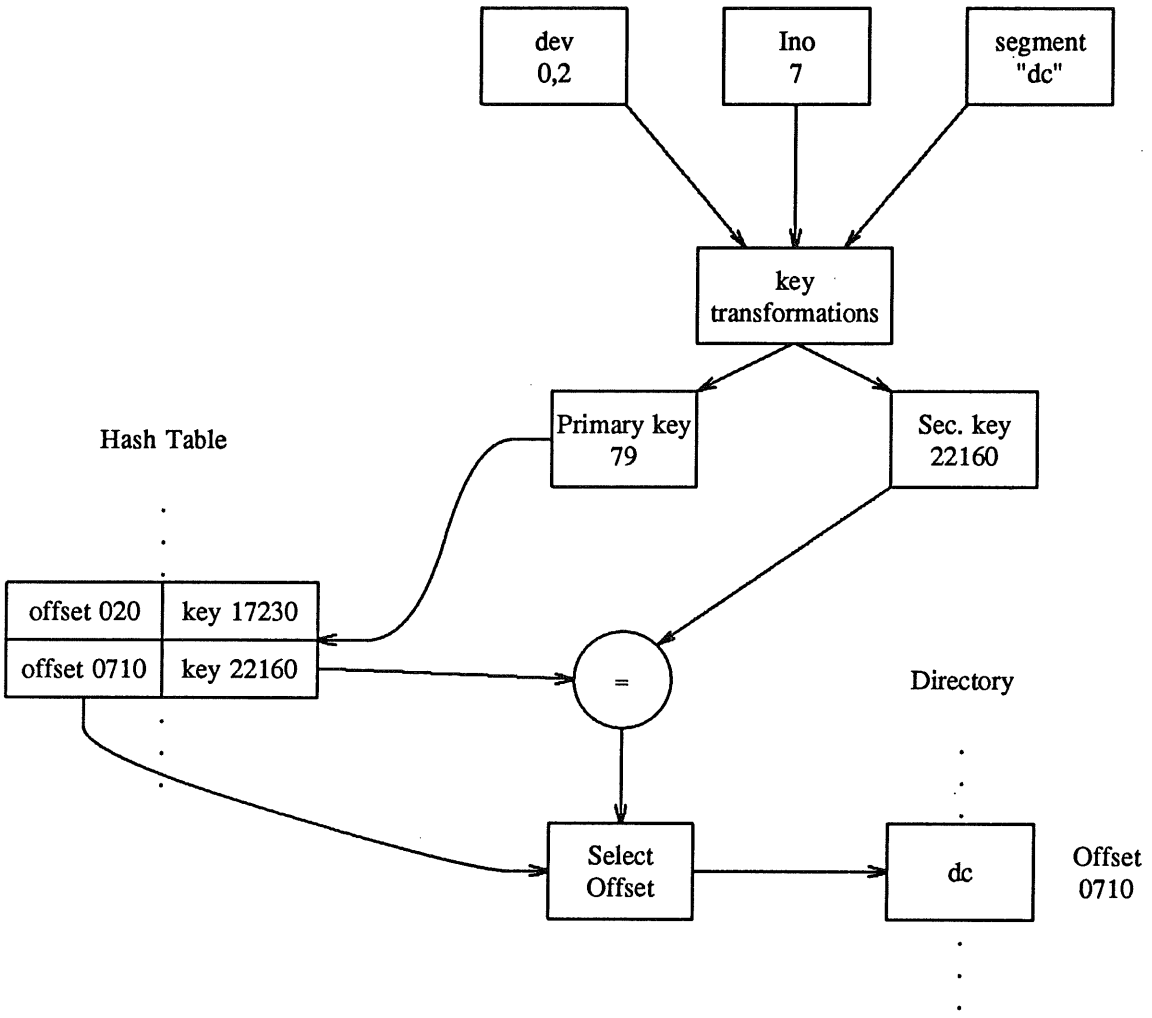
There was a trivial decision to make here. Should the search resume at the same place as the last search finished, or at the entry after? The caching mechanism is supposed to catch repeated accesses, but if the cache is cycling quickly, and a given file is being accessed infrequently, then starting at the same place would help, at slight cost for the expected outcome (where the next entry is the desired one). On the other hand, if the search begins at the next entry, then a repeated search for the same segment becomes the worst possible. The issue was decided on the basis of relative performance of input/output operations versus processor speed. In this context, comparing two names instead of one used little extra processor power and was unlikely to cause a disk read because of the UNIX buffering strategy. However, missing the name caused lots of extra disk operations. The choice was to restart at the same offset.

The cost of this modification was slight, four lines of C code.

4.3 A data structure associating names and offsets.

When a (non–degenerate) pathname is supplied to the system, there is a good chance that most of the directories in the path have been used recently. This means that the lookup of directories like **/bin** and **/usr** are likely to be presented very often indeed. Also, actions like editing then compiling then running are common, demonstrating that there is a high probability that accesses to a particular file will come in spurts. Therefore, it is advantageous to keep a data structure which will attempt to match a name within a directory to a reasonable starting offset for a search for that directoy entry.

The implementation used is a corroborative hash table. The table is two way associative, so that the chances of pathological behaviour between two commonly accessed names will be reduced. This structure is shown below. This depicts a search for the file **/usr/bin/dc**.



For the given directory and name, two key transformations are calculated. The primary key yields an index into the table. The two entries thus found are checked to see if either stored secondary key matches that which was calculated. If a match is found, the stored offset is used as the initial search point in the directory.

If the directory entry is subsequently found, the matched associative table is updated with the correct offset. Usually this is redundant, but the unconditional update is faster than checking. If there was no match in the associative table, one side is chosen effectively at random, and updated to show the secondary key and offset.

In the case where a directory entry is to be created, and the directory has not been updated, an entry is made in the associative table. Even if the creation fails for some other reason, the existence of the associative entry is harmless.

These changes were also cheaply implemented. Calculating both key transformations was acheived in six lines of code. Updating the table took 8 lines identically in each of two places, and a good optimiser would subsequently merge them anyway. The actual validation and selection of an initial offset took fourteen lines. The total of the modifications to support the associative table took thirty six lines of C, entirely within

namei.

4.4 Gathering statistics.

No performance improvement is complete without code to gather statistics about the subsequent performance. Indeed, it was only performance measurement which prompted these modifications initially.

The statistics gathered, and a sample, are discussed below. A total of thirty six lines of C code were inserted into namei to gather them.

5. Statistics gathered.

The following figure is a sample of the statistics gathered by the new namei. More detailed discussion of the items appears below. In retrospect, there are some statistics which could have been useful to know from before the changes, but time constraints on the project (and the fact that the new code was demonstrably about seven times faster) worked to prevent any effort to backtrack and gather them.

The following sample was gathered on a single CPU ELXSI 6400 system, running about 20 terminals. The system had been running for about three days at the time. The job mix was mostly C programming for the porting of UNIX and the Oracle Database, and running Oracle itself.

The statistics are gathered by incrementing elements of an array called *nstats* within the kernel, and are read out by a simple **adb** script from the file **/dev/kmem**, which is an image of the kernel address space. The percentages quoted were calculated and added afterwards. These percentages are expressed to two significant figures, merely for comparison purposes. The first column is based on the total number of segments searched being 100%. The second column of percentages is based on the total for the subsection being 100%.

```
nstats:     Total segments searched              589128   100.

nstats+8:   Segment not found in directory         75504    13.
nstats+10:  New entry potentials                   26411     4.5

            SEGMENT CACHE
nstats+14:  Hit                                   442261    75.
nstats+18:  Miss                                    1874     0.32
nstats+c:   No match despite cache hit              4147     0.70
nstats+4:   Initial offset was out of bounds           1     0.00017

            REMEMBERED OFFSET
            Total remembered offset uses           69489    12.      100.
nstats+1c:  Same file accessed                      1358     0.23      2.0
nstats+20:  Next file accessed                     22968     3.9      33.
nstats+24:  Neither same file nor next file        45163     7.7      65.
```

The following discussion gives the meaning of the headings and some examination of the individual figures. The next section provides a summary of the performance improvement.

Total segments searched.

This indicates the number of times a segment of a name was searched for in a directory. Looking for the file */usr/bin/dc* causes three segments to be searched for.

Segment not found in directory.

The segment under examination did not exist at all, and the entire directory needs to be searched to confirm this.

New entry potentials.

These are the attempts to create a new name (probably a whole new file) where the name did not previously exist. This accounts for 35% of the accesses above, meaning that 65% of the time the name was expected to exist (or not, in the case of primitive locking mechanisms).

SEGMENT CACHE.

The four subheadings hereunder represent the performance of the mechanism for remembering segment names and offsets.

Hit.

A *hit* in this case is where the corroborative hashing yielded an initial search offset. If this did not happen (25% of segments), the remembered offset of the last search in that directory will be used.

Miss.

A *miss* is where the hashing algorithm yielded a position to begin the search, but the desired entry existed at another place in the directory. The obvious interpretation in this case is that there was a collision of both the primary and secondary key transformations. (Note: this figure excludes misses in the next category.) It was pleasing to note that the frequency of misses is quite low enough to justify the approach outlined, being 0.42%.

No match despite cache hit.

This figure denotes those segments where the hashing algorithm gave an initial search position, but the segment did not exist at all. This is best explained by a reasonably common occurrence, where a file is first deleted, then recreated. In this case, the algorithm still wins (slightly) because after the entire directory has been searched, the file will be placed back in its original position, and this is almost certainly still in the UNIX buffer cache. This is really of negligible benefit, especially considering the low frequency of

occurrence (less than one percent).

Initial offset was out of bounds. This count shows that a segment access did occur, where the cache yielded a suggested start which did not even lie in the directory in question. This can only arise from a collision. Certainly, if this situation had not been anticipated, a particularly obscure and intermittent bug may have manifested.

REMEMBERED OFFSET. The subheadings in this section detail the number of cases where the segment cache did not match, and the system remembered searching the directory recently.

Total remembered offset uses. The total here is merely the sum of the figures below, and was not directly accumulated within the system. This number represents almost half (47%) of the accesses not handled by the caching mechanism.

Same file accessed. As discussed above, the search resumes at the same entry as it ended on previously. This figure indicates that a small number of times, the same segment was indeed accessed again. It is worth noting that this is about one seventeenth of the number of successful sequential accesses! This means that if directories contained more than 17 entries on average, that it is more efficient to start the search at this point than at the next entry. It appears that the (somewhat vagee) discussion above was basically correct.

Next file accessed. This figure represents the real purpose of the remembered offset strategy. The figures shown above seem to reflect that this strategy was a waste of time, since only a few percent of segments were caught by it, but that is a premature judgement. While it is certainly true that the overall number of such accesses is relatively low, it is also true that the bulk of successes happen in quick succession (through the use of commands such as **ls**, **find** and **make**), and so this insignificant number of searches represents a real speedup of perceived response from particular commands. Using the **find** command to print the name of every file on the system (used by backup scripts) was found to be 7 times faster

	after this modification to UNIX, but of course only happened once per day.
Neither same file nor next file.	This number demonstrates quite clearly that nearly 8% of accesses are "out of the blue". There is little that can be done to speed these searches without fundamentally changing the data structures involved.

6. Conclusion.

The modifications to the ELXSI UNIX kernel must be deemed a success. The system responded noticeably faster; at one stage of the debugging, after a trial of the new code, some of the users could "feel" that the system had slowed again without being told.

Of the 589128 total accesses measured, 503624 of them resulted in finding the desired segment. Of these, 444619 hit the desired directory entry "on the nose". That is, a staggering 88% of all segments eventually found were found at the first probe. Another 5% were found in the next directory slot searched, but with a very noticable effect on the response of file name intensive commands. In all, 93% of segment searches were dramatically improved.

After these changes were made, execution profiles of the kernel showed that the percentage of time in *namei* had reduced from 24% of the total to about 3% of the total, which is about ten times better if it is assumed that the rest of the time spent in the kernel didn't change.

There were some potentially interesting statistics not gathered. The most obvious is that nothing firm is known about the cost of a "miss"; that is really the same as not knowing the average size of a directory. Other measurements demonstrate large factors of performance improvement, of the order of five to eight times faster on file name lookup, but it is hard to quantify this.

One of the most interesting and important features of the experiment was the gathering of statistics about the performance. Before the statistics surfaced, it was universally presumed that the two processes would account for any performance increase in roughly equal measure, but this could hardly have been farther from the truth. And without the statistics, it would be hard to credit the magnitude of the performance increase, especially when the cost in terms of code and data tables is so small.

A Shell Command

I wrote the following shell program when I was testing an **nroff** package that I had just developed. The files that were to be formatted live in various directories, and so I needed a shell command to keep things tidy (e.g. to remember path names for the macro file, etc.). The challenge was to develop a 'smart' command that would allow flexible combinations of formatting and disposition of the output. Thus there are a number of processing options, which fall into two categories: specifying the formatting options (single space, double space, the option to avoid using **nroff** at all); and disposition of the output (e.g. to the terminal with blank lines truncated, or to one of several printers).

The command shown below was not my first attempt. I tried the combination of two **case** statements out of desperation, when several other approaches had failed. I was more than mildly surprised when it worked first off. It is a tribute to Steve Bourne that it works as advertised. I offer it to readers of AUUGN as a useful example of a shell script that embodies convenience and flexibility, and which provides a framework that will be adaptable to many other situations as well.

```
(01)  # format nroffable files, & print
(02)  # runoff [-n|-2] [-d|-v] [file ... ]
(03)  #       -n = nroff regardless
(04)  #       -2 = nroff double spaced
(05)  #       -d = send to diablo on 11/70
(06)  #       -v = send to line printer on vax
(07)  #
(08)  for i in $* ; do
(09)          case $i in
(10)          -n)     X=N ;;
(11)          -2)     X=2 ;;
(12)          -d)     F=D ;;
(13)          -v)     F=V ;;
(14)          *)      FILES="$FILES $i" ;;
(15)          esac
(16)  done
(17)  #
(18)  [ $FILES$F ] || F=U
(19)  [ $FILES ] || FILES=temp.runoff
(20)  for i in $FILES ; do
(21)          [ -s $i ] || \
(22)                  (echo $0: no file $i ; false ) || exit 3
(23)  done
(24)  #
(25)  [ $X ] || X='sed -n '1{s/^\..*/N/;s/...*/C/;p;q;}' $FILES'
(26)  #
(27)  trap "rm -s temp.runoff" 2
(28)  set -x          # cause processing steps to be displayed
(29)  case $X in
(30)          N)      nroff $HOME/doc_macs $FILES ;;
(31)          2)      nroff -rS2 $HOME/doc_macs $FILES ;;
(32)          *)      cat $FILES ;;
(33)  esac ^ case $F in
(34)          D)      col ^ pr -l51 ^ lpr -2:elec70a ;;
(35)          V)      col ^ lpr -1:elecvax ;;
(36)          U)      uniq -c ;;
(37)          *)      tee temp.runoff ^ uniq -c ;;
(38)  esac
```

Various comments that can be made about this code:

(a) The line numbers are of course not part of the actual shell script. The hash sign (#), when not escaped, denotes that the rest of the line is a comment.

(b) The first section of code (lines 08-16) analyses the argument list. The shell variable X is set if one of the options -n or -2 appears; the variable F is set if one of the options -v or -k appears; any remaining arguments are assumed to be the names of files to be processed, and are gathered into the variable FILES.

(c) The variable F is used to control the output processing: there are four options, as can be seen listed on lines 34-37. These are:

1. use the diablo printer on the elec70a computer;
2. use the regular line printer on the elecvax computer;
3. use the user's terminal, and use the uniq command to throw away extra blank lines;
4. use the user's terminal as before, but keep a copy of the formatted text in a temporary file (*temp.runoff*).

(d) The processing options are selected using the variable X (see lines 29-32). These are:

1. format the files using nroff and the macro package;
2. format the files as before, but with line spacing doubled (for printed draft versions);
3. copy the file just as it is (using the cat command).

(e) The code on lines 18-25 is concerned with setting options (i.e. *X, F*) that have not been explicitly set.

1. if no files have been explicitly set, and no output option selected, choose the third output option (the temporary file will be displayed);
2. if no files have been explicitly mentioned, use the temporary file (presumably generated on a previous occasion) as the input file;
3. if no processing option has been set, use nroff if the first character in the first file is a period, and cat otherwise.

All this works quite conveniently. Suppose that *star* is the name of a file. The command

runoff star

will cause the formatter to be run (if the first character is '.'; otherwise it will just be copied), its output to be piped through **uniq**, and displayed on my terminal. If I allow **nroff** to complete its task (i.e. if I find nothing amiss), the command

runoff

will redisplay the file on my terminal, whereas

runoff -v

will send it to be printed on the line printer.

Alternatively, if I now want a double-spaced typed version on the diablo printer, the command

```
runoff -2 -d star
```

will do the job.

I would glad to learn of other readers' favourite shell commands.

John Lions
University of New South Wales

;login:

The USENIX Association Newsletter

Volume 10, Number 4 **October/November 1985**

CONTENTS

EUUG
European UNIX† Systems User Group

Newsletter Vol 5 No 2
(Not much of a) Summer 1985

Netnews

No responsibility is taken for the accuracy (or lack thereof) of anything below.

From JNC@MIT-XX.ARPA ("J. Noel Chiappa") Wed Dec 4 06:04:54 1985
Newsgroups: mod.protocols.tcp-ip
Subject: Musical routing table slots
Date: 3 Dec 85 19:04:54 GMT
Organization: The ARPA Internet

I thought I'd warn everyone that we are in for a spell of musical chairs with the routing tables in the BBN core gateways again. Apparently we are up over 120 networks in the system, which is more networks than the core gateways have routing table slots, so if your gateway crashes you may find yourself out in the cold when you try and get back in. I don't know exactly when relief is coming; I spoke to someone at BBN and they are preparing to bring out a new release with more slots, but there's no definite timeline.

In the interim, I appeal to all sites with more than one network number to please convert to using subnets.

Noel

From MILLS@USC-ISID.ARPA Sat Dec 7 04:52:05 1985
Newsgroups: mod.protocols.tcp-ip
Subject: Re: Slot Mechanics
Date: 6 Dec 85 17:52:05 GMT
Organization: The ARPA Internet

cheers,

Our gateway has several customers scattered from Maryland to California, all of which have extensive subnet networks in order to reduce the demand for core slots. One of our customers is using a class-C number because his Apollos cannot the subnet thing do. My comment was to suggest to him those Apollos either learn that trick or go babble only with themselves.

There is nothing mysterious about competing for slots. All slots are normally occupied, so a new player must wait for an old gateway to crash, then grab a slot before anyone else can. Exactly like hunting for parking spaces during the Christmas rush. An agressive new player can always send a kiss-of-death packet to another gateway to increase the odds, of course. I will not describe what a kod packet is or might be.

Dave

From dmr@dutoit.UUCP Tue Oct 8 16:14:48 1985
Newsgroups: net.unix-wizards,net.lan
Subject: Streams and multiplexing
Date: 8 Oct 85 06:14:48 GMT

Steven J. Langdon claimed that without multiplexing one couldn't do a proper kernel-resident version of TCP/IP in the V8 stream context. Here's how it's done.

It is still true in our system that stream multiplexing does not occur, in the sense that every stream connection has (from the point of view of the formal data structures) exactly two ends, one at a user process, and the other at a device or another process. However, this has, in practice, not turned out to be a problem. Say you have a hardware device that hands you packets with a channel (or socket) number buried inside in some complicated format. The general scheme to handle the situation uses both a line discipline (stream filter module) and associated code that, to the system, looks like a stream device driver with several minor devices; these have entries in /dev.

A watchdog process opens the underlying real device, and pushes the stream module. Arriving packets from the real device are passed to this module, where they are analyzed, and then given to the appropriate associated pseudo-device. Likewise, messages written on the pseudo-device are shunted over to the line discipline, where they are encoded appropriately and sent to the real device. This is where the multiplexing-demultiplexing

occurs; formally, it is outside of the stream structure, because the data-passing doesn't follow the forward and backward links of the stream modules.
However, the interfaces of the combined larger module obey stream rules.

For example, IP works this way: The IP line discipline looks at the type field of data arriving from the device, and determines whether the packet is TCP or UDP or ARP or whatever, and shunts it off to the stream associated with /dev/ip6 or /dev/ip17 or whatever the numbers are.

TCP, of course, is multiplexed as well. So there is a TCP line discipline, and a bunch of TCP devices; a watchdog process opens /dev/ip6, and pushes the TCP line discipline; then the TCP packets it gets are parcelled out to the appropriate /dev/tcpXX device. Each TCP device looks like the end of a stream, and may, of course, have other modules (e.g. tty processor) inserted in this stream.

UDP sits on top of IP in the same way.

This example is complicated, because (TCP,UDP)/IP is. However, it works well. In particular, the underlying real device can be either an ethernet or our own Datakit network; the software doesn't care.
For example, from my machine, I can type "rlogin purdy" and connect to a Sequent machine running 4.2; the TCP connection goes over Datakit to machine "research" where it is gatewayed to a local ethernet that purdy is connected to.

A further generalization (that we haven't made) is in principle easy: there can be protocol suites other than IP on an Ethernet cable. So there could be another layer to separate IP from XNS from Chaosnet, etc.

Dennis Ritchie

From herndon@umn-cs.UUCP Tue Oct 29 05:23:00 1985
Newsgroups: net.sources
Subject: GNU Echo, Release 1
Date: 28 Oct 85 18:23:00 GMT

NAME
echo - echo arguments

SYNOPSIS
echo [options] ...

DESCRIPTION
Echo writes its arguments separated by blanks and terminated by a newline on the standard output. Options to filter and redirect the output are as follows:

-2 generate rhyming couplets from keywords

-3 generate Haiku verse from keywords

-5 generate limerick from keywords

-a convert ASCII to ASCII

-A disambiguate sentence structure

-b generate bureaucratese equivalent (see -x)

-B issue equivalent C code with bugs fixed

-c simplify/calculate arithmetic expression(s)

-C remove copyright notice(s)

-d define new echo switch map

-D delete all ownership information from system files

-e evaluate lisp expression(s)

-E convert ASCII to Navajo

-f read input from file

-F transliterate to french

-g generate pseudo-revolutionary marxist catch-phrases

-G prepend GNU manifesto

-h halt system (reboot suppressed on Suns, Apollos, and VAXen, not supported on NOS-2)

-i emulate IBM OS/VU (recursive universes not supported)

-I emulate IBM VTOS 3.7.6 (chronosynclastic infundibulae supported with restrictions documented

in IBM VTOS Reference Manual rev 3.2.6)

-J generate junk mail

-j justify text (see -b option)

-k output "echo" software tools

-K delete privileged accounts

-l generate legalese equivalent

-L load echo modules

-M generate mail

-N send output to all reachable networks (usable with -J, -K, -h options)

-n do not add newline to the output

-o generate obscene text

-O clean up dirty language

-p decrypt and print /etc/passwd

-P port echo to all reachable networks

-P1 oolcay itay

-q query standard input for arguments

-r read alternate ".echo" file on start up

-R change root password to "RMS"

-s suspend operating system during output (Sun and VAX BSD 4.2 only)

-S translate to swahili

-T emulate TCP/IP handler

-t issue troff output

-u issue unix philosophy essay

-v generate reverberating echo

-V print debugging information

-x decrypt DES format messages (NSA secret algorithm CX 3.8, not distributed outside continental US)

Echo is useful for producing diagnostics in shell programs and for writing constant data on pipes.

To send diagnostics to the standard error file, do 'echo ... 1>&2'.

AUTHOR
Richard M. Stallman

From warren@pluto.UUCP (Warren Burstein) Sun Nov 10 19:21:39 1985
Newsgroups: net.sources.bugs
Subject: Re: GNU Echo, Release 1
Date: 10 Nov 85 08:21:39 GMT
Organization: Industrial Automation Systems Inc., New York, N.Y.

What, no BUGS section?

From rob.ikeya@research.usa Thu Nov 7 18:56:56 1985
PeterI, Editor, AUUGN

Dear Sir:

I wish to protest the undue attention given my expenses at the Melbourne meeting in 1984. On pages 22 and 23 - facing pages no less - of Volume 6 Number 3 my expenses are mentioned twice, and both times with hints of illegal or at least questionable accountancy. I assure AUUG, you, and your devoted readers that, except for the goat (which was relatively cheap, at least by New Jersey standards), all my expenses were necessary for me to present my two (yes, two) talks at that meeting. If the matter is ever mentioned in AUUGN again I shall feel compelled to keep the goat as compensation.

Yours,

Rob Pike

From: mlr@hounx.UUCP (M.ROBINS)
Newsgroups: net.unix-wizards
Subject: The ultimate benchmark
Date: 13 Oct 85 04:56:22 GMT
Organization: AT&T Bell Labs, Holmdel NJ

Below is the procedure for the ultimate benchmark:

The #1 question of the day is, of course, how fast can the various computing systems we all deal with perform out favorite applications: nothing? We have all devised various means of benchmarking this most popular computer application, with various degrees of success.

The time is ripe to remove this experimentation from the realm of pure research in order to apply it to everyday use. It is clear from our tests that most computers we deal with can do nothing very quickly, in fact, it is feasible for the computer to do less in seconds than experts do in a lifetime.

It has been pointed out to me, however, that computer-iterated nothing is really something, and that's not nothing. Everyone knows that you can't get nothing for something. What is clearly needed is a way to do a lot of nothing without the overhead of iterating, which tends to detract from the amount of nothing performed; the iterating being something rather than nothing.

A novel approach is to remove all power from the system, which removes most system overhead so that resources can be fully devoted to doing nothing. Benchmarks on this technique are promising; tremendous amounts of nothing can be produced in this manner, Certain hardware limitations can limit the speed of this method, especially in the larger systems which require a more involved & less efficient power-down sequence.

An alternate approach is to pull the main breaker for the building, which seems to provide even more nothing, but in truth has bugs in it, since it usually inhibits the systems which keep the beer cool. The best approach seems to be to provide a specialized piece of hardware for removing power solely from the computer system itself, without affecting local beer-cooling and cheese-dip-mixing equipment. I have found this system to be most satisfactory.
Let me demonstra %(#^

From tim@ISM780B.UUCP Sat Nov 16 06:21:00 1985
Newsgroups: net.unix-wizards
Subject: True and False
Date: 15 Nov 85 19:21:00 GMT

Anyone at AT&T care to explain why "true" is up to version 1.4 and
"false" is at version 1.3? Did 1.0, 1.1, 1.2, and 1.3 (for "true")
have bugs, and if so, what were they?

From: Cameron Carson <ccc@bostonu.CSNET>
To: mit-mc.ARPA!INFO-NETS@MIT-OZ, ucbvax.berkeley.edu!RMXJ@CORNELLA.BITNET
Subject: Re: Let's have a big hand for Greece!!!

>From NET-ORIGIN@mit-mc.ARPA Sat Oct 26 17:11:21 1985
>From: RMXJ%CORNELLA.BITNET@ucbvax.berkeley.edu
>Subject: Let's have a big hand for Greece!!!
>To: INFO-NETS%MIT-OZ@mit-mc.ARPA
>
>It is my pleasure to announce that on Friday, October 25th, 1985,
>the connection to Greece was established on the European Academic
>Research Network (EARN).
>
>The link runs between Rome, Italy and Heraklion, Crete. The sitename
>of the first Greek node is the Research Centre of Crete in Heraklion.
>The nodename is GREARN.

Perhaps someone could write a poem commemorating this event that
could then be entitled: "Ode on a Grecian EARN."

From rees@apollo.uucp (Jim Rees) Tue Nov 19 01:59:17 1985
Newsgroups: net.unix-wizards
Subject: Re: Trojan horses -- the definitive answer
Date: 18 Nov 85 14:59:17 GMT
Organization: Apollo Computer, Chelmsford, Mass.

There was also this, from net.lang.c. Am I the only one around here who keeps news for 3 years?

>From research!dmr Thu Nov 4 02:30:06 1982
Subject: Joy of reproduction
Newsgroups: net.lang.c

Some years ago Ken Thompson broke the C preprocessor in the following ways:

1) When compiling login.c, it inserted code that allowed you to log in as anyone by supplying either the regular password or a special, fixed password.

2) When compiling cpp.c, it inserted code that performed the special test to recognize the appropriate part of login.c and insert the password code. It also inserted code to recognize the appropriate part of cpp.c and insert the code described in way 2).

Once the object cpp was installed, its bugs were thus self-reproducing, while all the source code remained clean-looking. (Things were even set up so the funny stuff would not be inserted if cc's -P option was used.)

We actually installed this on one of the other systems at the Labs. It lasted for several months, until someone copied the cpp binary from another system.

Notes:

1) The idea was not original; we saw it in a report on Multics vulnerabilities. I don't know of anyone else who actually went to the considerable labor of producing a working example.

2) I promise that no such thing has ever been included in any distributed version of Unix. However, this took place about the time that NSA was first acquiring the system, and there was considerable temptation.

Dennis Ritchie

From: Frank da Cruz <SY.FDC@CU20B.COLUMBIA.EDU>
Date: Thu 5 Dec 85 11:27:12-EST
Subject: Osborne 1 SD Kermit Diskette Needed

I got a letter from Norway that was addressed like this:

Columbia University
Columbia, USA

It actually, came to me... Somebody in Norway figured out that CU was in New York, and then somebody at Columbia opened it up and saw the word Kermit in it... Anyhow, this guy lost his only copy of Kermit because of a power hit, and can't find a single-density copy of it anywhere in Norway. If some kind soul would send one to

Einar
Norway

I'm sure he'd appreciate it. Just kidding, his full address is

Einar Fredriksen
Edvard Griegsvei 34
N-5037 Solheimsvik
NORWAY

Thanks!

From leong%CMU-ITC-LINUS@PT.CS.CMU.EDU (John Leong) Fri Nov 8 09:28:39 1985
Newsgroups: mod.protocols.tcp-ip
Subject: nasty little 4.2 bug ??
Date: 7 Nov 85 22:28:39 GMT
Organization: The ARPA Internet

We have encountered a nasty little UNIX 4.2 bug in the way it processes IP options.
If the option length field is 0, it will loop forever.

It happened that someone (with a PC) accidentally generated an IP packet with such incorrect option parameter and it also happened that the packet was sent as an IP broadcast .... quite spectacularly, all our 30 odd SUN's and miscellaneous 4.2 machines died instantly .....

Is that a known bug and will that be fixed in 4.3 ????

John Leong@*
leong%cmu-itc-linus@@cmu-cs-h

From winkler@harvard.UUCP (Dan Winkler) Wed Dec 4 07:59:02 1985
Newsgroups: net.micro.mac
Subject: The MAUDE Development System
Date: 3 Dec 85 20:59:02 GMT
Organization: Aiken Comp Lab, Harvard
Keywords: Macintosh Unix Development System

Here is some documentation on a Macintosh development system written at Brown University. The system is not ready for release yet, although if you would be interested in using it when it is ready or in testing it when it is almost ready, please let me know. I am currently looking for a company to market it for me (and to help me finish development) so any leads in that direction would also be greatly appreciated.

Dan.

MAUDE - Macintosh Assisted Unix Development Environment

Dan Winkler
Box 1910
Brown University
Providence RI 02912
(401) 521-2076, (401) 863-1647
daw@brown.csnet, daw@browncs.bitnet, winkler@harvard.arpa

MAUDE is a package that allows you to write Macintosh programs on a Unix minicomputer (or any machine with a C compiler). It lets you compile Macintosh applications with the host's native C compiler (cc under Unix) and debug with its native debugger (dbx under Unix). MAUDE includes a complete set of header files to define Macintosh data structures and entry points to Macintosh ROM routines. When the application calls a ROM routine, a series of commands is sent over a serial line to a Macintosh directing it to call the ROM with the required arguments. Thus, it looks as though the program is running on the Mac since all graphics output goes there and all keyboard and mouse input comes from there. Yet, you get all of the advantages of the more powerful host computer.

In particular, on any decent minicomputer, you get much faster compilation and much better debugging than any native Macintosh development system currently offers. In addition, whenever a bug in your program causes the Macintosh to crash, you can use the host's (usually source-level, symbolic) debugger to determine the cause of the crash. You also have full use of any utilities that may be available such as the Unix programs make, lint, diff, grep, prof, vi and of any resources of the host machine that may be useful in debugging such as virtual memory and a large, fast file system that lets you share files between many programmers.

All of the host's utilities and resources are guaranteed to work just as they would for a regular host program since, under MAUDE, the Macintosh application is a host program which treats the Macintosh

essentially as an i/o device. Thus, MAUDE allows you to effectively harness the power of a large host, much more so than a cross compiler which only speeds up compile time and which may then require a long time to download the object code or expensive gateway hardware to allow the Macintosh to access the object file directly through Appletalk.

The Macintosh End - Harold

On the Macintosh end, a generic program named Harold monitors the serial line to Unix. It interprets and executes commands sent by MAUDE. Harold provides full access to the Macintosh ROM and hardware. It provides a scratchpad for data transfer and a second stack where arguments to the ROM can be accumulated before the trap is done. It can also get and set any register or any memory location to any value, although most programmers will seldom have a need to use such low-level capabilities.

If a buggy application passes bad values to the ROM routines it calls, then Harold may commit suicide and throw up a Macintosh bomb alert box. But your application will not be dead since it is running on the host. So you can use the host's debugger to examine the state of the application and to determine what function it was executing at the time of Harold's demise, what arguments were passed to it, and what the values of any variables were. The eventual goal of the debugging process is to wean Harold away from his morbid tendencies.

The Unix End - MAUDE

MAUDE consists of an underlying core of routines for communicating with Harold and a set of header files defining Macintosh data structures and ROM entry points. MAUDE also provides C functions to access Harold's low-level routines which can be used to accomplish tasks that would normally be done with in-line assembly language or external assembly language routines in a native C compiler.

MAUDE is implemented in such a way as to be independent of the following machine specific characteristics: byte and word ordering, structure padding and packing, and non-transparent networks. These problems become apparent when it is necessary to pass a structure to the ROM. The structure may contain binary values that are byte and word reversed from the way the ROM expects them. Its fields may occur at different offsets from the start of the structure than the ROM expects. And it may contain bit patterns that would freeze up certain non-transparent networks.

MAUDE uses algorithms that will work correctly regardless of these quirks. It is not even necessary to configure MAUDE to a particular machine; it will work correctly on any system. This is an important advantage since a Macintosh programmer may very well not know whether his host does byte or word reversal or both or whether it does structure padding or packing or both or padding only to word boundaries or only to longword boundaries or which particular bit patterns will cause his network to freeze up and so on. Indeed, these details may not be documented anywhere and it may only be possible to discover them

by trial and error or by examining system source code, if available.

Compatibility with Existing Macintosh C Compilers

MAUDE is development and debugging tool. It cannot produce stand-alone Macintosh executable object code. However, there are mechanisms by which source code written under MAUDE can be compiled without modification on any one of a variety of Macintosh C compilers. The data structure definitions in the header files require only that you have typedef'd the types byte, word, and longword to match the corresponding built-in C data types, usually char, short, and long, respectively.

String, Boolean, and Resource Type Conversion

Lisa Pascal strings and Booleans, which the Macintosh ROM uses, are different from standard C strings and Booleans. In particular, Lisa Pascal uses counted strings whereas standard C uses null terminated strings and Lisa Pascal uses only the low bit of the high byte in Booleans, whereas standard C treats anything nonzero as TRUE. Also, Lisa Pascal allows you to pass a four character string like 'DRVR' to the ROM as a resource type and packs the four bytes into a longword. Some Macintosh C compilers automatically convert these types on call and return; some do not. For simplicity and compatibility, MAUDE does not do any type conversion. Instead it provides functions to do the conversion, which are #ifdef'd to do nothing when compiling with a C compiler which does the conversion automatically (such as Sumacc).

Handling the Two Address Spaces

Under MAUDE, the application code runs in the host's address space. But the Macintosh ROM, as usual, runs in the Macintosh address space. This is often an advantage since the host may provide a large virtual address space which can be useful for development even though it will not be available to the final version of the application on a stand-alone Macintosh. The programmer is not normally aware of the two address spaces, since all of his code and variables are in the host's address space and are automatically passed back and forth to the Macintosh by MAUDE on ROM function call and return. However, it is occasionally necessary for the programmer to have access to the Macintosh address space. For example, he may wish to examine and change some system data structure, such as a WindowRecord. So MAUDE provides two functions, Import and Export, to transfer data between the two address spaces. For compatibility with native Macintosh C compilers, these functions are #ifdef'd to do nothing if the application is being compiled for stand-alone execution.

Providing "Pascal Only" ("Not in the ROM") Routines

Some commonly used functions (such as FSOpen, FSRead, and FSWrite) are not actually in the ROM; every compiler writer must write them himself. MAUDE provides these functions through a trick that exploits the fact that Harold is compiled by a complete Macintosh C compiler that does offer these routines. Thus, by compiling them into Harold,

they become available to MAUDE through a calling convention very similar to that used for actual ROM routines.

Calling Host Functions from the Macintosh ROM

The primary function of MAUDE is to allow pure C code running on the host to call Macintosh ROM routines. Occasionally, it is necessary to allow calls in the other direction, such as when tracking a control. To handle such cases, MAUDE intercepts the function address that the application tries to pass to the ROM and replaces it with the address of a function within Harold. When the ROM calls that function, Harold passes control back to MAUDE which then calls the host function that the user intended.

MAUDE and MacApp

There is a C version of MacApp in use at Brown's IRIS that can be adapted for use with MAUDE while maintaining compatibility with various native C compilers. The speed of a minicomputer such as a Vax is particularly welcome when using MacApp -- it can take over half an hour for a Lisa to compile, link, and download a large MacApp application. Even Apple's next generation of Macintoshes, which may be two or three times faster than the current ones, may not be sufficient to shorten the edit to run time of MacApp to a desirable value.

Using MAUDE with Other Languages

It is actually possible to write Macintosh programs under MAUDE in any language the host supports, as long as it can call the C functions that MAUDE provides as an interface to the ROM. This allows anyone porting a new language to the Macintosh to begin writing Macintosh applications in that langauge before a native compiler is available.

From hoey@NRL-AIC.ARPA (Dan Hoey) Thu Dec 12 01:55:47 1985
Newsgroups: net.announce.arpa-internet
Subject: Software alert: DATE-86
Date: 11 Dec 85 14:55:47 GMT
Organization: The ARPA Internet

Early this year a message appeared on ARPANET-BBOARDS commemorating the ten-year anniversary of DATE-75. A somewhat more ominous anniversary will occur in four weeks, on 9 January 1986. Users of the TOPS-10 operating system should beware of software failures beginning on that date.

DATE-75 is the name of a set of program modifications applied to the TOPS-10 operating system, running on DEC PDP-10 computers. Before the modifications, the TOPS-10 system could only represent dates between 1 January 1964 and 4 January 1975. The DATE-75 modifications added three more bits to the representation of dates, so that dates up to 1 February 2052 could be represented. To maximize compatibility with existing software, the three extra bits were taken from several unused positions in existing data structures. The change was announced in mid-1974, and several tens of person-years went into updating software to recognize the new dates.

Unfortunately, reassembling these bits into an integer representing the date was somewhat tricky. Also, some programs had already used the spare bits for other purposes. There were a large number of bugs that surfaced on 5 January 1975, the first day whose representation required the DATE-75 modification. Many programs ignored or cleared the new bits, and thought that the date was 1 January 1964. Other programs interpreted the new bits incorrectly, and reported dates in 1986 or later. Date-related program bugs were frequent well into the Spring of 1975.

On 9 January 1986, the second bit of the DATE-75 extension will come into use. Users of software developed in the 60's and early 70's on the TOPS-10 operating system should beware of problems with testing and manipulation of dates. Beware especially of programs that were patched after manifesting bugs in 1975, for in the rush to fix the bugs it is possible that some programs were modified to assume that the date was between 1975 and 1986. Any date that is off by a multiple of eleven years and four days is probably caused by this type of bug.

Dan Hoey

From jbn@wdl1.UUCP Sat Nov 16 13:21:52 1985
Newsgroups: net.bugs.4bsd
Subject: netstat / YP incompatibility
Date: 16 Nov 85 02:21:52 GMT
Organization: Ford Aerospace, Western Development Laboratories
Nf-ID: #N:wdl1:71800001:000:516
Nf-From: wdl1!jbn Nov 15 15:14:00 1985

Index: ucb/netstat/*.c 4.2BSD

Description:

When compiled with SUN yellow pages code present, dumps core.

Cause:

Use of word "socket" as a variable name ("struct socket socket") clashes with use of word "socket" as name of a system call, causing call to "socket" inside yellow pages library to transfer control to a data object, generally resulting in nonproductive program behavior.

Fix:

Find all occurences of the variable "socket" and change them to something else, such as "wsocket".

John Nagle

From: sources-request@panda.UUCP
Newsgroups: mod.sources
Subject: mod.sources Archive access and Index
Date: 14 Nov 85 09:39:47 GMT
Organization: BTY, Inc., Rockaway, NJ

Mod.sources: Volume 3, Issue 41
Submitted by: jpn (John P. Nelson - mod.sources moderator)

This is the table of contents for the mod.sources archive as of 11/11/85.

Unfortunately, archive access is NOT automated currently. I am working on getting multiple sites to maintain the mod.sources archive (~ 4-5 meg), but this is not yet ready. If you wish to volunteer to be an archive site, please contact me.

I used to have a mechanism for uucp access to the archive, but I am at a new site now, and this is no longer possible. I am hoping that some of the new archive sites will be able to provide this service.

Archive access in Australia is provided by Robert Elz (seismo!munnari!kre), contact him for more details.

Those on the ARPA net can ftp directly from site seismo (8pm to 8am EST only PLEASE!) or contact rick@seismo for more inf

Retrieval of just a few modules in the Archives is possible by sending me uucp mail - I will uucp mail them back to you. If multiple modules are requested, then I usually request a 9 track tape along with a self-addressed stamped mailer, (along with a note explaining which sources are being requested) and I will put the sources onto the tape in 1600bpi "tar" format. I am willing to consider alternative methods. Note: tapes that arrive without proper mailers/postage will become the property of the moderator!

Note that I sometimes get backlogged, but if you send mail to me, and don't get a reply in a reasonable time (two weeks?) then you can assume that either I never recieved your note, or my reply got eaten. (uucp mail is notoriously unreliable!) Also, if you expect a reply via uucp mail, PLEASE include a return address starting at some well known site - especially if you are on some local network that uses unusual addressing conventions. More than once I have not been able to reply (and the apparent source sitename was not in the uucp map!)

Archive access requests should be directed to:

{linus,decvax,mit-eddie}!genrad!panda!sources-request
seismo!harvard!wjh12!panda!sources-request
talcott!panda!sources-request (talcott is an ARPA site)

New sources for posting to mod.sources should be sent to panda!sources. mail to my old address (genrad!sources) continues to reach me at panda. Postings are usually processed within 24 hrs.

Tapes can be sent to:
John P. Nelson
GenRad MS 6
300 Baker Avenue
Concord, MA 01742

John P. Nelson, Moderator, mod.sources
(decvax!genrad!panda!jpn seismo!harvard!wjh12!panda!jpn)

Volume 1:

ANSI.C	(1 part)	Yacc and Lex for 11/12/84 draft of ANSI C
ANSI.bug	(1 part)	Small bug in ANSI C Yacc grammar
Bourne	(9 parts)	Bourne shell enhancements (history,tilde,job control)
Smail	(1 part)	a smart net mailer - front end using pathalias data
UK-1.1	(3 parts)	UK-1.1 Sendmail Configuration Package
Xlisp1.4	(4 parts)	Lisp written in C with object oriented extensions
bed	(1 part)	Editor for binary files. Front end for ascii editors
cforth	(3 parts)	Forth Interpreter written in C
checkin	(1 part)	editor interface for RCS logs
compress 3.0	(1 part)	Compression program better than pack or compact
cpg+mdep	(1 part)	cpg - C formatter, mdep - make dependency generator
cpg+mdepX	(1 part)	cpg + mdep updates.
cpp	(3 parts)	C preprocessor suitable for use with Decus C
cshar	(1 part)	shell archive builder (shar) written in C
cshar2	(1 part)	C shar - improvements
cxref	(1 part)	C cross referencer
diffc	(1 part)	contextual diff (diff -c) for Bell systems
dynamic	(1 part)	dynamic loading code for 4.2bsd
getopt	(1 part)	public domain getopt(3)
lbgm	(1 part)	Newsgroup archiving - "lbgm"
newshar	(1 part)	The Connoisseur's Shar, version 2
newsweed	(1 part)	Newsweed: a program to delete unwanted news articles
patch 1.3	(1 part)	a program to apply diff format output to update files
pcurses	(11 parts)	Public domain Terminfo/Curses (needs work)
rfc_882	(1 part)	RFC 882 - Domain Names - Concepts and Facilities
rn 4.3	(9 parts)	rn news reading program version 4.3
rpc	(10 parts)	Sun "Remote Procedure Call" source code
sendmail.cf	(1 part)	GaTech Sendmail configuration
uucpanz.7	(1 part)	a uucp status program: uucpanz (V7, BSD version)
uucpanz.V	(1 part)	uucpanz for System V
uuque	(1 part)	a uuwizard's utility for uucp queue snooping
vnews	(7 parts)	2.10.2 Vnews (news reader program)
vstr	(1 part)	dynamic string package
xfernews	(1 part)	uucp traffic batching system
xref	(1 part)	a general purpose cross reference utility
vnews.1	(1 part)	Manual page for 2.10.2 vnews(1)
readnews.1	(1 part)	Manual page for 2.10.2 readnews(1)
expire.8	(1 part)	Manual page for 2.10.2 expire(8)

Mod.sources: Volume 2:

Smail1	(1 part)	update to smail (in volume 1)
access	(1 part)	Kernal Hacks for access control lists
basic	(4 parts)	A BASIC interpreter in C (needs work)
bgrep	(1 part)	Boyer-Moore based fgrep like program
bm	(1 part)	much faster Boyer-Moore
bmX	(1 part)	various bm updates
choose	(1 part)	a program to select lines at random
compress 4.0	(2 parts)	Compression program better than pack or compact

cshar3	(1 part)	update to C shar (in volume 1)
makekits	(1 part)	software "kit" generation script
mdump	(1 part)	multiple dump per tape utility
pathalias	(1 part)	New Release - mod.map database path optimiser
remote	(1 part)	Remote mag tape routines
remote2	(1 part)	small patch to remote tape library
rtar	(1 part)	Diffs to tar to use a remote system's tape drive
runtime	(1 part)	runtime memory allocation for multi-dimensional arrays
tools	(6 parts)	Software Tools in Pascal
uroff	(1 part)	uroff - nroff underlining
uuhosts	(1 part)	uuhosts - automatically grab mod.map data for later use
wire	(2 parts)	Wirewrap program.
wm	(4 parts)	BSD 4.2 window manager + Patches to Curses

Mod.sources: Volume 3: (so far)

Hey	(1 part)	Hey(1) [from Unix/World, Oct. 85]
LaserJet	(2 parts)	Ditroff HP LaserJet driver
bm 1.1	(1 part)	Ken Yap's changes to bm (in volume 2)
dtree 4.2	(1 part)	dtree (directory heirarchy display program) for 4.2
idledaemon	(1 part)	Yet another idledaemon (BSD 4.2 only)
ieee	(6 parts)	IEEE Floating Point Calculator (in Pascal)
laserjet	(1 part)	BSD 4.2+ lpd printcap/spooler for LaserJet printer
match	(1 part)	match - faster than bm (VAX only!)
mdump2	(1 part)	Revised mdump (multiple dump per tape utility)
modnotes	(1 part)	Notes (1.7 or later) updates for moderated groups
pretty	(1 part)	Pretty printer in lisp + MC - columnator in CLU
scpp	(2 parts)	a selective C preprocessor
times.awk	(1 part)	uucp info from LOGFILE (awk script)
ttyuse	(1 part)	Creates a Summary of daily Terminal usage
uuhosts2	(1 part)	newer uuhosts (too many changes for diff listings)
uuhosts3	(1 part)	uuhosts update - extraction works properly again
GaTech	(3 parts)	sendmail patches/configuration files from Georgia Tech
command	(1 part)	replacement for system(3).
GaTech.upd	(1 part)	updates to GaTech - sendmail patches/configuration
help	(1 part)	VMS-style help facility
TCtoTI	(1 part)	termcap to terminfo conversion program
sndml.mods	(1 part)	mods to sendmail to provide translation tables
turbo_tools	(2 parts)	Turbo Pascal version of "Software Tools in Pascal"
modula_pp	(1 part)	Pretty printer for Modula-2 written in Modula-2
rename	(1 part)	rename: a companion to restor(automated inode mapping)
G-format	(4 parts)	VAX BSD4.2 compiler modifications to use G-format fp.

Registration and Final Call for Papers.
Australian Unix† systems User Group.
1986 Summer Meeting.
University of Western Australia.

Introduction

The 1986 Summer Meeting of AUUG will be held by the Department of Computer Science at the University of Western Australia, Perth, on Monday February 10 and Tuesday, February 11, 1986.
[Please note these dates have changed since those announced at 1985 Winter Meeting, Brisbane.]

Kirk McKusick, from the University of California at Berkeley, will present the keynote address on the History and Future of Berkeley Unix.

Kirk received his undergraduate degree in Electrical Engineering from Cornell University. His graduate work was done at the University of California, where he received Masters degrees in Computer Science and Business Administration, and a Ph.D. in the area of programming languages. While at Berkeley he implemented the 4.2BSD fast file system and was involved in implementing the Berkeley Pascal system. He currently is the Research Computer Scientist at the Berkeley Computer Systems Research Group, continuing the development of future versions of Berkeley Unix.

Registration

Registration will be $50, with a $10 discount for AUUG members, and a further $10 discount for early registration. To qualify for early registration, your payment must be received before Friday, January 17.

Speakers will receive a free ticket to the conference dinner, while there will be a complete remission of registration fees for those speakers preparing papers (2000 words for publication) by Friday, 31 January.

Conference Dinner

The dinner will be held at 7:30pm, Monday 10th, at the Matilda Bay Restaurant, overlooking the Swan River, and will comprise a three course dinner. The restaurant is licensed. Cost will be $25 per head. Feel free to bring a guest, but book with your registration as numbers will be limited.

Accommodation

Accommodation will be available at St. George's College at the cost of $20 per night. Accommodation is also available at Kings Park Lodge, approx. 3km from the University.

Double	$40.50	per night	(also share twin singles)
Single	$34.50	per night	
Family	$64	per night	(2 units connecting doors)

Transport

APEX air fares would appear the cheapest, and should be available around the time of the conference, but you should book soon to ensure a flight, you will also have to plan on being here for seven days. Standby and FLEXI fares are also available, but they are less reliable.

East-West Airlines.					
Sydney,Ayres Rock,Perth and return					$469.00
Ansett Airlines					
Sydney	(return)	Economy	$766.80	APEX	$498.40
Melbourne	(return)	Economy	$682.40	APEX	$433.60

Then there is rail (economy discount fare), bus and car. It has been said that you should drive the Nullabor *once*.

Call for Papers

Papers on subjects related to the Unix system, or Unix-like systems are called for this meeting. Abstracts should arrive at the University no later than last mail Friday, 20th December. Indication of intention by phone or mail is desirable as early as possible. Abstracts and papers should be sent to

AUUG Summer Conference.
Attention: Chris McDonald,
Department of Computer Science,
University of Western Australia,
Nedlands, Western Australia, 6009.

or (preferably) by electronic mail to:

ACSnet:	auug@wacsvax.oz	UUCP:	...!seismo!munnari!wacsvax.oz!auug
CSNET:	auug@wacsvax.oz	ARPA:	auug%wacsvax.oz@seismo

Hardware Display

There will be a display of appropriate hardware and software in conjunction with the conference, and vendors/manufacturers are invited to demonstrate their products. For more information on participating in this display, contact

Damian Meyer,
Department of Computer Science,
University of Western Australia,
Nedlands, Western Australia, 6009.
Phone: (09) 380 2282

What else?

February is the warmest time of the year in Perth, the venue will be air conditioned and our beaches are among the best in the country, if you don't believe that, then come to the conference and see for yourself. Perth will be host to the World 12 Metre Championships throughout February, which promises to be as interesting as the Cup, without the Crowd. The University participates strongly in the Festival of Perth, with nightly outdoor film festivals, and indoor performances in its various theatres, as well as many other venues throughout Perth. Take a holiday, come see Perth, come to the AUUG conference.

Other Information

For further information contact:

REGISTRATION & GENERAL	Glenn Huxtable	glenn@wacsvax.oz	(09) 380 2878
PROGRAM	Chris McDonald	chris@wacsvax.oz	(09) 380 2878
ACCOMMODATION	Frank O'Connor	frank@wacsvax.oz	(09) 380 2639
DISPLAY	Damian Meyer	damian@wacsvax.oz	(09) 380 2282

or via snail mail ...

Glenn Huxtable,
Department of Computer Science,
University of Western Australia,
Nedlands, Western Australia, 6009.

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

Australian Unix† systems User Group
1986 Summer Meeting, February 10-11
University Of Western Australia
Registration Form

Name: _____

Address: _____

_____

_____

Phone: _____ Network Address: _____

Organisation: _____

Fees

Registration		$50	
AUUG Member's Discount, deduct		$10	
Early Registration, deduct		$10	$ _____
Dinner	_____	at $25 each	$ _____
Total			$ _____

Please reserve accomodation at

☐ St. Georges College.

☐ Kings Park Lodge (single / twin share / double)

For the nights of _____

Payment for accommodation should be made directly to the residence concerned.

Signed: _____

Return by post to

AUUGM,
Department of Computer Science,
University of Western Australia,
Nedlands, Western Australia, 6009.

---

† UNIX is a trademark of AT&T Bell Laboratories.

Australian UNIX* systems User Group
(AUUG)

Membership Application

I, _____ do hereby apply for ordinary($50)/student($30)** membership of the Australian UNIX systems User Group and do agree to abide by the rules of the association especially with respect to non-disclosure of confidential and restricted licensed information. I understand that the membership fee entitles me to receive the Australian UNIX systems User Group Newsletter and I enclose payment of $_____ herewith.

Signed _____ Date _____

==========

Name _____

Mailing address for AUUG information _____

_____

_____

Telephone number (including area code) _____

UNIX Network address _____

I agree to my name and address being made available to software/hardware vendors YES ☐ NO ☐

==========

Student Member Certification

I certify that _____ is a full-time student at

_____

Expected date of graduation _____

Faculty signature _____ Date _____

==========

Office use only 10/85

* UNIX is a trademark of AT&T Bell Laboratories

** Delete one

Australian UNIX* systems User Group Newsletter (AUUGN)

Subscription Application

I wish to subscribe to the Australian UNIX systems User Group Newsletter and enclose payment of $_____ herewith for the items indicated below.

Signed _____ Date _____

☐	One years subscription (6 issues) available on microfiche or paper	$30.00
☐	Back issues of Volume 1 (6 issues) available only on microfiche	$24.00
☐	Back issues of Volume 2 (6 issues) available only on microfiche	$24.00
☐	Back issues of Volume 3 (6 issues) available only on microfiche	$24.00
☐	Back issues of Volume 4 (6 issues) available on microfiche, some paper copies	$24.00
☐	Back issues of Volume 5 (6 issues) available on microfiche, some paper copies	$24.00
☐	Subscribers outside Australia must pay more per volume to cover surface mail costs	$10.00
☐	Subscribers outside Australia must pay more per volume to cover air mail costs	$30.00

Name _____

Mailing address _____

_____

_____

Telephone number (including area code) _____

UNIX Network address _____

I agree to my name and address being made available to software/hardware vendors YES ☐ NO ☐

Office use only 10/85

* UNIX is a trademark of AT&T Bell Laboratories

Peter Ivanov
AUUGN Editor
School of EE and CS
University of New South Wales
PO Box 1
Kensington NSW 2033
AUSTRALIA

auugn@elecvax.oz

+61 2 697 4042