January 1982 Vol IV No I

AUUGN

Australian UNIX Users Group

N E W S L E T T E R

This is the first issue of Volume 4 of AUUGN. It is also the first issue edited by Bob Kummerfeld and Chris Rowles of Sydney University. The subscription rate has been doubled to $24, this change has allowed us to turn AUUGN into a journal rather than a newsletter (but we will keep the old name!). Many issues of AUUGN have been over 50 pages and most readers wish to keep and refer to old AUUGN's. The newsletter format (photocopied sheets with a single staple) was not sturdy enough and so has been changed. The AUUGN "year" has been changed to coincide with the calendar year, partly because the change in editorship caused a delay and partly because we felt it made more sense.

This edition of AUUGN marks the introduction of a number of regular departments. Contributions for each of these sections are solicited from our readers. If we don't receive contributed items AUUGN will shrink and die so please send us info on what YOU are doing with UNIX.

This issue was really produced by Tim Long, Jason Catlett, Chris Maltby and Bruce Ellis. I am just the front man!

Bob Kummerfeld

Chris: Hi Bob, whats new?

Bob: . . . groan . . . I just realized we have to produce our first issue of AUUGN . . .

C: It was your idea that we become AUUGN editors, you think of something. Besides, I'm getting married next week and going to the UK for six months!

B: If Piers were here now he would cry "J**** S***!"

pause

B: . . . Hey! why not call in that group of hackers from Basser to help . . . what do they call themselves . . . I know, the "Conceptual Integrity Agency" they should be able to give some ideas.

C: Hold on. That bunch is pretty way out, won't the regular readers of AUUGN be offended?

B: No! As long as there are plenty of interesting articles about Unix I think most people will be happy. Besides, it should be very entertaining.

C: I'm not so sure. The last time I went past the room where they hangout I heard someone singing "Hello Unix" to the backing of a synthesiser playing "Hello Dolly".

B: Relax! Let's go and talk to them . . .

some time later

rhubarb . . . rhubarb . . . mumble
. . . hmmmmm . . . yeh, what about a giant christmas edition in the shape of a christmas tree . . . how about a Denis Ritchie centrefold . . . rhubarb . . .

B: All right, come back to earth, we've got to get something out that has good material on Unix, so please tone it down.

CI people:
Yeah sure Bob. Leave it to us . . .

Formatting C

Tim Long

Basser Department of Computer Science
University of Sydney
(timl:basservax)

1. Introduction

Every C programmer has strong views on idiom, style and formating. Unfortunately these views are as idiosyncratic as they are inflexible. In C many semantically distinct constructs have only minor syntactic differences. For human beings formatting is often the only reasonable method of distinguishing them.

2. Object and type declarations

To establish some terminology we present the following example:

```
static unsigned int     stab_segs, stab_size = 1109, ref_counts[MAX_N];

<----base type---->     <--item-> <--item->          <------item------>

<----first part--->     <---------------second part----------------->

<----------------------------declaration------------------------------>
```

A declaration usually has two parts. The first part, which we will call the base type, is a list of storage class specifiers, basic type specifiers and adjectival modifiers of basic types. Some examples of storage classes are static and register. The term storage class has lost much of its original intuitive meaning. For instance the modifier typedef is considered a storage class, but it clearly has nothing to do with storage. Examples of basic types are int, float and enum. Examples of adjectives are long and short.

The second part is a comma separated list of items to be declared and their initialisations. Each of these items includes an identifier, possibly surrounded by *, ( ) or [ ]. Any item may be followed by an = and an initial value.

2.1. Formatting simple declarations

Only one item should be declared per declaration: there should be no comma separated lists. For example:

```
char    *p, c;          /* WRONG */

char    *p;             /* RIGHT */
char    c;              /* RIGHT */
```

The reasons for this are

(a) all but the first identifier in the WRONG case are hidden and often missed in a quick glance;

(b) the mixture of types (pointer to character and character in the above example) can cause confusion;

(c) it is harder to add a comment or initialisation to an item in the WRONG case.

All base types, items and initialisations within a group of declarations should be verticaly aligned. For example:

```
char *tape_name = "/dev/rht0" /* WRONG */
unsigned long offset; /* WRONG */
int state = st_idle; /* WRONG */

char            *tape_name      = "/dev/rht0";  /* RIGHT */
unsigned long   offset;                         /* RIGHT */
int             state           = st_idle;      /* RIGHT */
```

We can now consider a declaration to have three parts.

(a) The base type, which is never omitted.

(b) The item being declared, which may be omitted.

(c) The initialisation, which will probably be omitted.

It is this three part nature which dominates the layout of simple declarations.

2.2. Complex type definitions

The definition of complex types such as structs, unions and enums should be isolated and typedefed. The definition of a complex type in C is a side effect of its appearance in the base type part of a declaration. To make this clearer, consider the following declarations:

```
enum states             state;
struct point            where;
```

Clearly the enum states and the struct point are base types and state

and where are items. Now consider this (badly formatted) example.

```
enum states {st_idle, st_active}        state;
struct point {int x; int y;}            where;
```

This is equivalent to the first example except that definitions are bound to the identifiers states and point. Notice that the definition of the members of the complex type is part of the base type. Finally it should be noted that it is not necessary to bind the complex type definition to an identifer, as the following example shows:

```
enum {st_idle, st_active}       state;
struct {int x; int y}           where;
```

2.3. Formatting complex type definitons

The complex type declarations in the previous section were in poor style: a new type name should be created for each complex type generated. There are two ways of doing this. This example demonstrates one:

```
typedef enum                    /* RIGHT */
{
        st_idle,
        st_active,
}
        states;

typedef struct                  /* RIGHT */
{
        int     x;
        int     y;
}
        range;
```

Much of the above formatting will be explained latter. The main point is that the enum and struct are not bound to any identifier. A new type name is created to refer to the types as a whole. The declaration of the objects state and where becomes:

```
states          state;          /* RIGHT */
point           where;          /* RIGHT */
```

Unfortunately this method cannot always be used. When a struct or union references itself (in the form of a pointer) the type of the pointer can not be named because its declaration is not complete. In this situation the following variation can be used.

```
typedef struct struct_node      node;
struct struct_node
{
        int     node_value;
        node    *node_link;
};
```

This binds the definition of the structure to the identifier struct_node in order to achieve a forward reference. But the following declaration is also valid (and preferable):

```
typedef struct node     node;   /* RIGHT */
struct node
{
        int     node_value;
        node    *node_link;
};
```

Notice that this binds the definition of a structure and a new type to two identifiers, both of which are called node. These identifiers come from logically distinct symbol tables. The structure binding is irrevelant and serves only as a mechanism for the forward definition of the type.

Formatting the member list of a complex type is straightforward. The on curly brace should be placed on a new line directly under the base type. The elements of the member list are indented one tab stop, and the formatting rules are applied recursively. The off curly brace is aligned with its matching one. In the second variation this is followed by the semicolon. But if a type name is being defined, the name is placed on a new line indented one tab stop from the off brace, followed by the semicolon.

There are several justifications for this layout.

(a) The conceptually independent acts of type definition and storage allocation are separated.

(b) The indenting and positioning of brackets serves to surround the memberlist declaration with white space, separating it from peripheral activity and placing it where it can be seen and modified. The same arguments apply here as for simple declarations.

(c) The use of a typedef makes the programmer's intention clear.

(d) Subsequent declarations become clean and narrow enough for the author to be consistent with vertical alignment.

The following is a trimmed example of large structure declaration. The source fragments comes from an include file. Near the top of this file is found the following block of typedefs:

```
/*
 *      Forward declarations of general purpose data types.
 */
typedef struct cfrag    cfrag;
typedef struct cnode    cnode;
typedef struct ident    ident;
typedef struct xnode    xnode;
typedef union data      data;
```

Although not all of these forward refrences were necessary all stuctures and unions were given them in this case for consistency.

The following structure definition was found further down the file along with all the other complex type definitions.

```
struct xnode
{
        union
        {
                xnode   *xu_xnd;
                ident   *xu_id;
        }
                x_left;
        union
        {
                xnode   *xu_xnd;
                cnode   *xu_cnd;
        }
                x_right;
        xnode   *x_type;
        xnodes  x_what;
        data    x_value;
        short   x_flags;

};
```

Typical declarations involving this and related types look something like:

```
register xnode  *x;
register ident  *id;
place           where;
```

3. Function definitions

```
char    *
strcpy(s1, s2)
char    *s1;
char    *s2;
{
```

The above function definition has a useful characteristic. Although the function returns a non int object, its name appears at the start of a line. This both improves readability and lends itself to automated searching methods. The alternative

```
char    *strcpy(s1, s2)
char    *s1;
char    *s2;
{
```

is readable but does not allow an easy distinction between invocations and the definiton in an editor search. In general the same rules apply to a function definition as a simple type except that a new line is taken immediately before the identifier.

The leading bracket of the formal parameter list should be placed immediately after the function name. The formal parameters themselves should be placed on the same line with a space after each comma. The closing bracket should be placed hard against the last formal parameter (or the opening bracket if there are no formals). For example:

```
main(argc, argv, env)

main( )
```

Declaration of the formal parameters follows, hard against the left margin and obeying the rules of simple declarations.

4. Formatting blocks

Blocks have two parts, surrounded by curly braces. These parts are

(a) declarations local to this block;

(b) executable statements.

Where the block is the body of a function the on curly brace is placed on a line of its own, hard against the left margin. Each time a sub-block is opened the on curly brace is indented one further tab stop from the level of the enclosing block. The brace always appears on a line of its own. For example:

```
while (i < n)
{
        dothis( );
        dothat( );
}
```

This positioning of the opening curly bracket is important to

(a) visualy seperate the body of the block from surrounding peripheral acitivity;

(b) act as a pointer to any flow control construct controlling the block;

(c) allow a similar visual clue to any controling expression.

Placing the opening curly brace on the end of the previous line both embedes any controlling expression in blocks of text and leads to special cases when blocks are opened to gain local variables.

The local declarations are started on a new line indented one tab stop from the initial brace. Formatting is as described above. One blank line should be left between the local declarations and the executable statements. If there are no declarations the code should start on a new line immediately after the on brace. For example:

```
{
        char    *p;
        int     i;

        p = "this is a demo";
        {
                i = 0;
                return i;
        }
}
```

The occasional blank line between executable statements is acceptable but should be not be over-indulged. The significance of such blank lines is easily lost. Often a block comment is more appropriate (see "Comments").

5. Formatting executable statements

Statements are placed on new lines indented one tab stop from the level of the on and off braces of their surrounding block. Its is unacceptable to have more than one statement on one line.

```
i = 0; j = 10;                  /* WRONG */
return;                         /* WRONG */

i = 0;                          /* RIGHT */
j = 0;                          /* RIGHT */
return;                         /* RIGHT */
```

Placing many statements on one line banishes all but the first to oblivion. Although it may be argued that some statements are logically related this is not sufficient justification for the devaluation of statements tacked onto the end of another.

6. Formatting expressions

When an expression forms a complete statement, it should, like any other statement, occupy one or more lines of its own and be indented to the current level. Binary operators should be surrounded by spaces. Unary operators should be placed hard against their operand.

```
* p ++;                  /* WRONG */
i=i*10+c-'0';            /* WRONG */

*p++;                    /* RIGHT */
i = i * 10 + c - '0';    /* RIGHT */
```

The ternary operators ? and : should also be surrounded by spaces.

When a sub-expression is enclosed in brackets, the first symbol of the sub-expression should be placed hard against the opening bracket. The closing bracket should be placed immediately after the last character of the sub-expression.

```
a = b * ( c - d );       /* WRONG */

a = b * (c - d);         /* RIGHT */
```

Note that the symbols ->, ., and [] which build up primaries (factors) are not considered binary operators in this context. They should not be surrounded by spaces. For example:

```
addr = addrs[ (d >> 3) & 037 ];          /* WRONG */
addr -> csr = 0;                         /* WRONG */

addr = addrs[(d >> 3) & 037];            /* RIGHT */
addr->csr = 0;                           /* RIGHT */
```

The round brackets which surround the arguments of a function call attract no spaces.

```
puts ( "hi\n" );         /* WRONG */

puts("hi\n");            /* RIGHT */
```

Commas, whether used as operators or separators, should be placed hard against the previous symbol and followed by a space.

```
write(2,"whoops\n",7);          /* WRONG */

write(2, "whoops\n", 7);        /* RIGHT */
```

White space in expressions is useful as much by its lack as its presence. For instance placing spaces in the inside edges of brackets merely spreads out the expression and loses the suggestion of binding. Excessive white space causes inflation and promotes devaluation.

Occasionally expressions become too large to fit on a single line. Breaking at an arbitrary column is distasteful and often unreadable. Rewriting the expression as two, possibly using a temporary, may destroy its conceptual integrity and efficiency. The solution is to reformat the expression over several lines. Consider the following:

```
fprintf
(
        stderr,
        "%s: Could not open %s for reading. %s\n",
        my_name,
        tape_name,
        errno > sys_nerr ? "" : sys_errlist[errno]
)
```

This demonstrates the formatting of the most common cause of long lines, the function call with many arguments. Notice the position of the opening and closing brackets. The actual parameters are aligned vertically one tab stop in from the current level. Each actual parameter occupies a line of its own.

```
if
(
        (id->id_type == NULL)
        ||
        (
                (id->id_type->x_what == xt_arrayof)
                &&
                (item->x_left->x_what == xt_arrayof)
                &&
                (id->id_type->x_subtype == item->x_left->x_subtype)
                &&
                (id->id_type->x_flags & XIS_DIMLESS)
        )
)
```

Here we see another common line length transgressor put in its place. Notice the placement of binary operators and brackets on lines of their own.

The basic message in the above exmaples is don't be afraid of using more lines to make the expression clear.

7. Formatting flow control constructs

In order to give visual distinction between flow control constructs (such as for and while) and function calls, a small variation in formatting is introduced. A space is used to seperate a flow control keyword from any controlling expression. For example:

```
if (p != NULL)
{
        dothis();
        dothat();
}
return p;
```

The space seperates the keyword in order to emphasise the flow control dominating the following statement or block.

In view of the above the formatting of for and while statements is straightforward:

```
for (p = root; p != NULL; p = p->next)
        process(p->data);

while ((c = getchar()) != EOF)
        putchar(c);
```

When formatting if statements several alternatives are possible. The simple if statement is again straightforward:

```
if ((fid = open(name, O_READ)) == SYSERROR)
        perror(name);
```

In a simple if-else combination the else keyword should be placed on a line of its own at the same indentation as the if:

```
if (c == '\\')
{
        ...
}
else
{
        ...
}
```

Although these are the only variations of if statements distinguished in the language the author feels that it is often desirable to consider an if-else chain as a flow control construct in its own right. In this case the following layout is acceptable:

```
if (c == '\\')
{
        ...
}
else if (c == '"')
{
        ...
}
else if (c == '\'')
{
        ...
}
else
{
        ...
}
```

The formatting of switch statements is simple:

```
switch (pid = fork( ))
{
        ...
}
```

However the placement of case labels and labels in general often gives trouble. The keyword case should be placed on a line of its own at the same indent level as the controlling switch keyword. A space should separate the word case from the constant expression which is immediately followed by the colon. A blank line should be left above a case label if program flow does not fall through it. For example:

```
switch (pid = fork( ))
{
case SYSERROR:
        fprintf(stderr, "%s: Could not fork.\n", my_name);
        exit(1);

case 0:
        ...
}
```

Ordinary labels and defaults follow the same rules.

Placing executable statements on the same line as a label (of any sort) is unacceptable since

(a) the statement is visualy hidden by the label;

(b) it is impossible to be consistent with indenting, there will always be some constant expression too long.

The formatting of do statements is difficult. The intuitive method

is:

```
do
{
        ...
}
while (...);
```

However the duality of the while keyword often leads to confusion, especially if the preceeding block is large. To avoid this an arbitary convention is adopted (as in the case of flow control keywords and function calls). The while keyword should be indented one tab stop from the level of the closing brace:

```
do
{
        ...
}
        while (...);
```

8. Comments

Much of this document has concerned itself with formatting aimed at improving readability. The tacit assumption is that readable code is easier to understand than unreadable code. Comments do not improve readability but attempt to directly aid understanding and maintenance.

Comments embedded in code tend to create a dense mass of text. Comments which begin and end on the same line, intermixed with code, should be avoided. It is better to use a few large comments than many smaller ones distributed through the text.

```
/*
 *      This demonstates the layout of a "block comment".  One
 *      comment such as this at the head of a hundred line
 *      function is often more useful than hundreds of two or
 *      three worders.
 */
main(argc, argv)
int     argc;
char    *argv[];
{
```

```
/*
 *      Block comments such a this and the above should follow
 *      the level of the code they refer to.
 */
if (...)
{
        /*
         *      Indented when the code is indented.
         */
        ...
}
```

One of the most important aspects of comments is their semantic content. Cryptic references should be avoided, "in" jokes should be obviously irrelevant. Comments should contain either

(a) complete english sentences, with capital letters and full stops (periods);

(b) some sort of well defined logical symbolism;

(c) diagrams.

For example:

```
/*
 *      Warning!
 *      i + strlen(str) + base - p <= BUFSIZ
 *      or else.
 */


/*
 * The shape of the file is thus:
 *
 *      ----------------
 *      | header       |
 *      |--------------|
 *      | hashtable    |
 *      |   (hashsize * |
 *      |    TABENTLEN) |
 *      |--------------|
 *      | table        |
 *      |   (tabsize *  |
 *      |    TABENTLEN) |
 *      |--------------|
 *      |  entries     |
 *      \      .       /
 *             .
 *      /      .       \
 *      |      eof     |
 *      ----------------
 *
 * I hope this is a little clearer now.
 */
```

The costs and benefits of moving to Berkeley virtual memory UNIX* (4.1 bsd).

Doug Richardson
Chris Maltby
Tim Long

Basser Department of Computer Science
University of Sydney

This report summarises the results of two sets of benchmarks that we ran for comparison between 4.1 bsd UNIX (a paging system) and the AUSAM enhanced version of UNIX 32V (a swapping system) called unix/aus that we run at the University of Sydney.

In an attempt to estimate the consequences of using the Berkeley 4.1 bsd UNIX system it was decided to benchmark the two systems with some existing tools, deemed to be representative of typical student workloads. These tests were a quick and easy comparison between the two systems, but may not be truly representative of real workloads. The 4.1 bsd version of UNIX was run as supplied, configured for 80 users. There was no attempt to tune 4.1 bsd for this kind of workload (if possible). The unix/aus system used was the current production system, which generally supports 60-75 simultaneous users. Our hardware configuration is given in appendix 1.

We lack sufficient familiarity with the internal workings of 4.1 bsd UNIX to estimate how much it could be optimised for our load characteristics. The Berkeley system is already highly optimised, in fact many of the performance enhancements in unix/aus are descended from it. There are certain overheads associated with any demand paging system for the VAX. For instance, there is no way to record a page as having been recently accessed, except by taking a trap.

We selected two of the original UNIX performance scripts which were part of the our VAX acceptance tests. One script consists of a single student Pascal compilation. The other simulates a student session including editing, compilation and execution of a Pascal program (see appendix 2). Sprinkled throughout this typical session were random sleeps that served to simulate the time the user spent thinking (or whatever students do when not typing).

---

* UNIX is a Trademark of Bell Laboratories.

By choosing a particular script and running many copies of it simultaneously we can test our system under simulated student load. Also, a CPU-bound idle process is run with a high 'nice', so that remaining CPU time can be accounted for. However, these scripts differ from a real load: there is no terminal I/O associated with running them, the job mix is highly artificial, the random sleeps and script timing spawn extraneous processes, and the jobs all require the same resources. We nevertheless feel that these scripts give a good indication of system performance under our workloads.

Results

The results for the first script, the Pascal compilation, are shown in figure 1. This summarises the results for 4.1 bsd and unix/aus. Each run of N simultaneous compilations is depicted in three points connected by a vertical bar. The lowest point is the minimum time, i.e. the compilation that completed first. The highest point of the vertical bar is the maximum time, and the remaining points, which are joined to form the graph, represent the average time for all the compilations.

Two anomalies in the graph deserve explanation. The results for 30 simultaneous compilations under unix/aus appear to be worse than those for 40. We believe that this is because our disk i-node free list emptied during this run and the system consumed an extra 10 seconds or so searching for more. We suspect that an analogous situation occurred during the run of 55 simultaneous compilations under 4.1 bsd unix. Some individual scripts on 4.1 bsd unix finished much more quickly than the average. This seems to be an artefact of the paging algorithms. As memory becomes overcommitted in this system, some processes get swapped out, and do not return to memory until the demand has lowered. Unix/aus may show similar behaviour with 75 scripts.

Figure 2 summarises the results from the runs with the second script. This script is a rough approximation to a typical student compile and execute cycle. In an attempt to simulate the 'thinking' time of a user, this script included many calls to a program that would sleep for a random amount of time. We kept track of the time each script spent sleeping, and the time plotted is the elapsed time minus the sleep time. The resulting time figure is roughly proportional to the response time of the system. However, it must be kept in mind that each run is simply one run in a sample space; it is possible to have a set of sleeps that make the response appear better or worse than the results we obtained. We confined ourselves to one run of each set of N simultaneous scripts. The point for unix/aus with 20 scripts (note 1 on graph) is probably due to similar system behaviour as noted above.

The last four runs, those for 60, 65, 70 and 75 simultaneous scripts (note 2) under unix/aus, were different from the others. This is because our system process tables were not large enough to handle the 5 processes per script that we had been using. We changed the script slightly to make it use only three processes, so that we could make these final measurements. By retaining the slope, and transferring the curve upward, we believe we have a fair prediction of how unix/aus would have performed had we increased the size of the system process tables.

We mention in passing that this test revealed a bug in 4.1 bsd unix: one script behaved incorrectly in each of the runs for 55, 60, 65, and 70 scripts; two scripts failed in the test of 75 scripts. We made no attempt to find the cause.

Figure 3 shows the overall times (including the random sleeps) for both systems as well as CPU time not consumed by the idle process. This non-idle time shows the increasing system overheads. The 4.1 bsd non-idle graph has an increasing gradient for larger numbers of scripts. This extra time must be increased overhead. Figure 4 shows disk page traffic generated by running the scripts (divided by 10). This seems fairly linear, indicating that neither disk saturation nor thrashing has set in. The number of swaps (abandonment of the process' page table) seems to be correlated with the average real time for the scripts. It would seem from this that bad swapping decisions are being made, or that swap-in overhead is particularly high; as mentioned above, processes which escape this swapping appear to finish much more quickly.

Conclusions

From the above graphs, it would seem that the paging UNIX is 50% slower in real time when under heavy loads. At about 70 users our current system is saturated; we expect that 4.1 bsd would be saturated at something less than 45 users. This roughly agrees with the stated maximum configurations recomended in the installation documents for 4.1 bsd, where the maximum is said to be 32 to 40 users. Of course, these users can run much larger programs than they could under unix/aus. However very few of our students feel handicapped by memory limits (we have 3.25 megabytes). Separate tests showed that the presence of a few large, low priority background processes do not significantly affect the performance of the 4.1 bsd system. However, such tasks would receive very little CPU attention during heavily loaded periods.

Appendix 1 - Hardware Configuration

VAX 11/780 CPU with Floating Point Accelerator
3.25 Megabytes of memory
1 TE16 45ips tape drive on MASSBUS
3 RM03 67 Megabyte disk drives on MASSBUS
1 CDC9766 256 Megabyte disk drive on EMULEX SC21V UNIBUS controller
7 DZ11E 16 line terminal multiplexors.
4 KMC11A UNIBUS microprocessor (for controlling DZ11s)

UNIBUS and MASSBUS are trademarks of Digital Equipment Corporation.

Appendix 2 - Script details

p-script

```
ed - errors.p <:errors1
rs 30 1
pi errors.p
rs 30 1
ed - errors.p <:errors2
rs 30 1
pi errors.p
px obj <postfix.data
times
```

'rs 30 1' sleeps for a time between 1 and 31 seconds (1 + r(30)).

:errors1

```
!rs 30 5
/program postfix/s/$/;/
!rs 30 5
/prosedure/s//procedure/
!rs 30 5
/op;/s//op:/
!rs 30 5
/{ begin/s//begin (/
!rs 30 5
/end./s//end/
!rs 30 5
/^end/s//&./
!rs 30 5
w
q
```

:errors2

```
!rs 30 5
/i:.*real/d
!rs 30 5
/fudge/d
!rs 30 5
w
q
```

Figure 1: Comparison of time taken for simultaneous pascal compilations

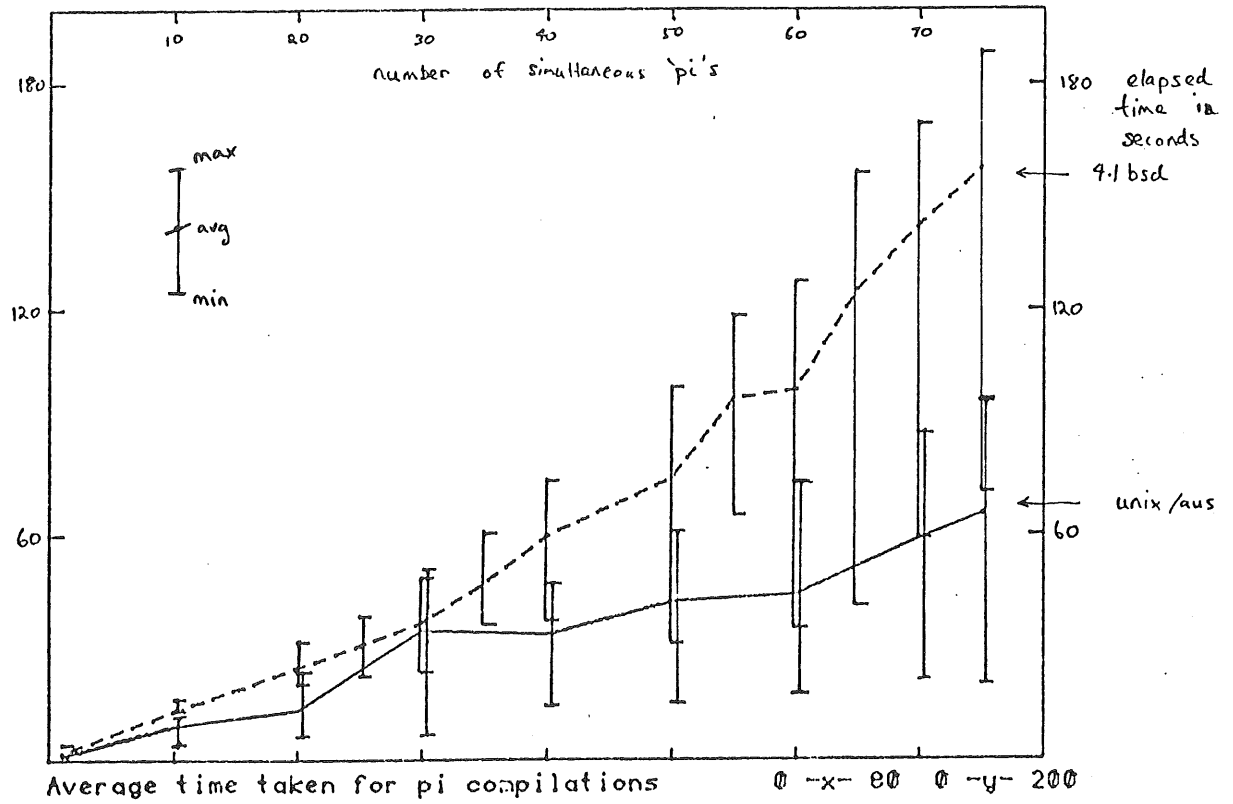
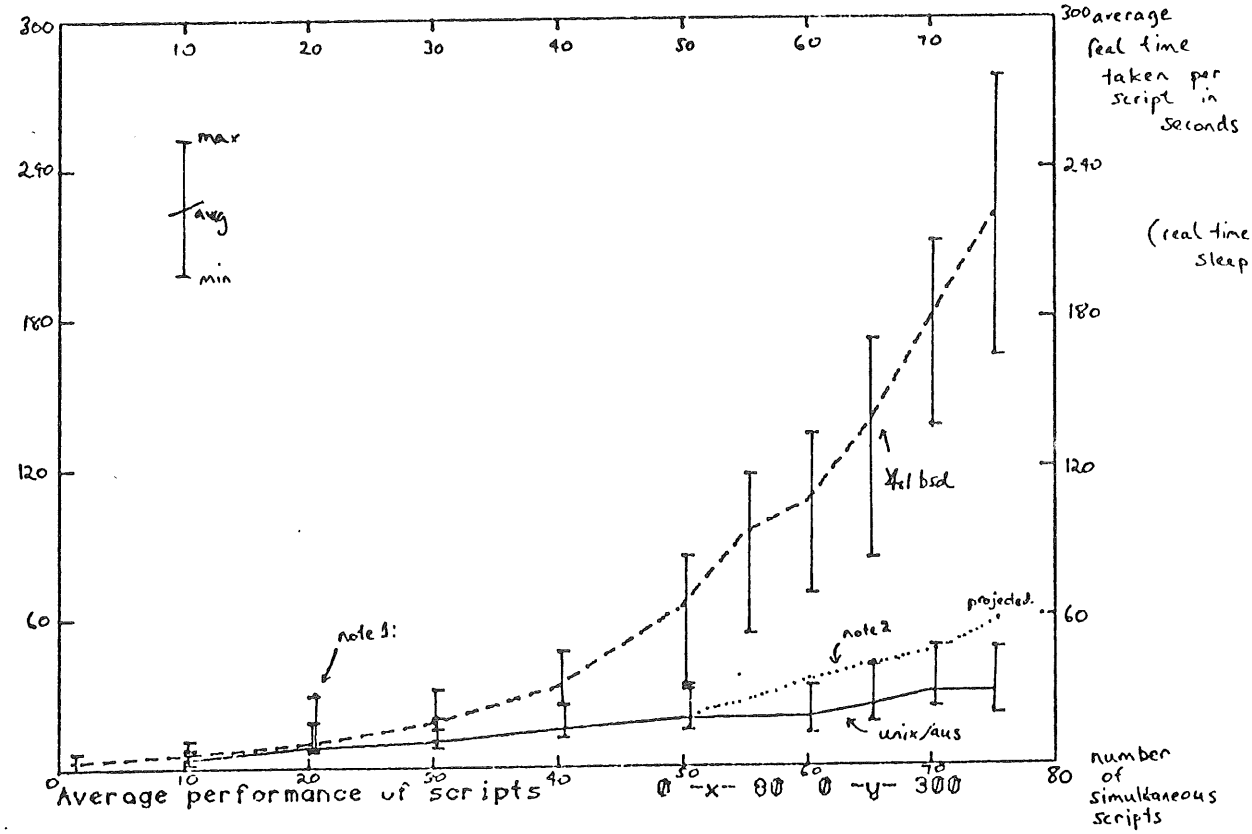


Figure 2: Pseudo 'response time' vs number of scripts



1: this point belongs to unix/aus.
2: this is projected, as real points measured with 2 fewer processes per scripts

Figure 3: Real time and 'non idle' time vs number of scripts

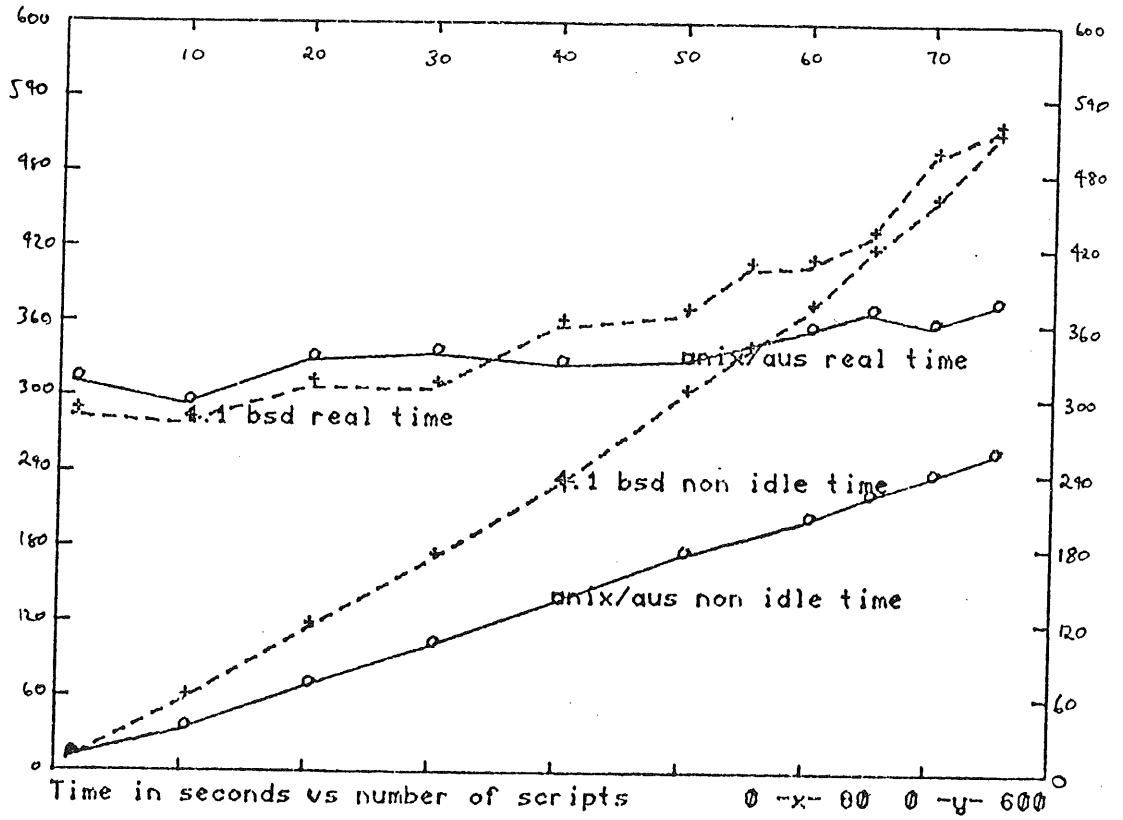
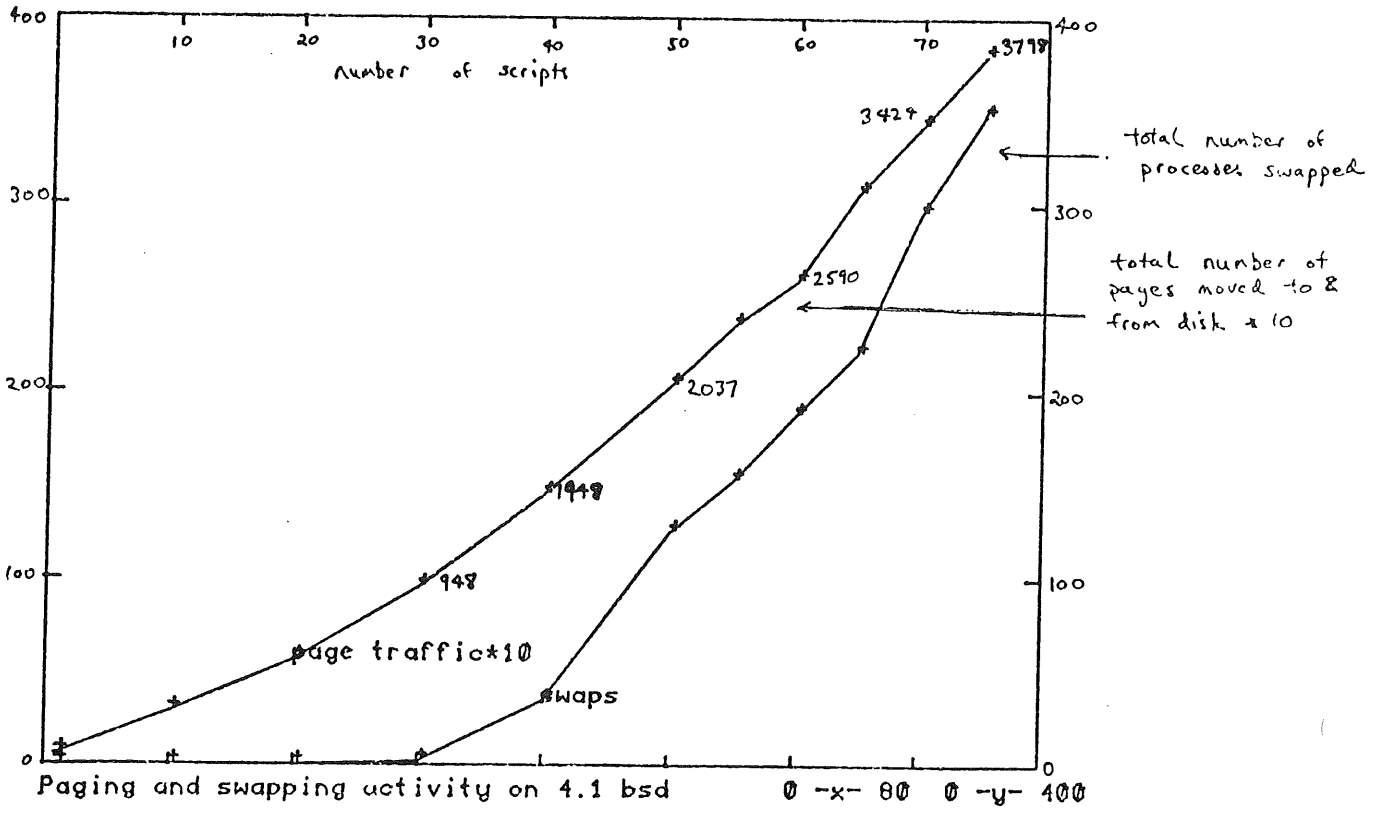


Figure 4: Paging & Swapping vs number of scripts

4.1 bsd only



Take the determinism out of your motds

Tired of that old

WELCOME TO LEVEL 7 UNIX

motd[1]? Or perhaps you're frightened that they've stopped reading those important messages about disk space. With little effort you can introduce variety, local content and spice into your motds.

A selection of motds and a crontab entry to cycle them is all it takes. Now it's safe to use jokes that would otherwise become stale and jaded through overexposure. Your users get into the habit of reading the damm thing and get the message when its important.

The shell script[2] cycle-motd given at the end of this note will copy the next motd in a sequence given in /etc/motds/motd* to /etc/motd. Place any really important messages in /etc/motds/motd. If this file is greater than 0 length it will overide the normal cycling mechanism.

Just make the directory /etc/motds. Put your motds in /etc/motds/motd0, /etc/motds/motd1 etc. Finally, add a crontab entry to run cycle-motd every hour say. It's nice if you keep the number of motds co-prime to the number of cycles per day to avoid boring those who log in at regular hours.

We started this at basservax after our 1247 students went on holidays and we could afford to be a bit frivolous. We like to give a new batch every week. The following are the motds we started with. The first set is from "The hitchhiker's guide to the galaxy" by Douglas Adams (Pan Books, stocks currently exhausted but expected in February). The speaker is, of course, Eddie, the Heart of Gold's shipboard computer. He was part of the Sirius Cybernetics Corporation's series of automata which featured GPP (Genuine People Personalities).

Why hello there.

Hi gang! This is getting real sociable isn't it?

I want you to know that whatever your problem, I am here to help you solve it.

---

1 Message Of The Day, as found in /etc/motd and printed by getty at login.
2 Level 7 sh version.

Hi there! This is Eddie your shipboard computer, and I'm feeling just great guys, and I know I'm just going to get a bundle of kicks out of any program you care to run through me.

Impact minus twenty seconds, guys.

Please call me Eddie if it will help you to relax.

When you walk through the storm, hold your head up high...

Now this is going to be your first day on a strange new planet, so I want you all wrapped up snug and warm, and no playing with any naughty bug-eyed monsters.

I can see this relationship is something we're all going to have to work at.

I can even work out your personality prolems to ten decimal places if it will help.

All I want to do is make your day nicer and nicer and nicer...

The next week we went on in a slightly more serious vain, quoting from "The Mythical Man-month" by F. P. Brooks, Jr. (Addison-Wesley).

"It is a very humbling experience to make a multimillion-dollar mistake, . . . I vividly recall the night we decided how to organize the writing of external specifications for OS/360."

"All programmers are optimists."

"Add one component at a time."

"To write a useful prose description, stand way back and come in slowly."

"I will contend that conceptual integrity is THE most important consideration in system design."
From "The Mythical Man-month" by F. P. Brooks, Jr.
Who learnt about conceptual integrity the hard way.

"Simplicity and straightforwardness proceed from conceptual integrity. Every part must reflect the same philosphies and the same balancing of desiderata."

". . . one finds that debugging has a linear convergence, or worse, where one expects a quadratic sort of approach to the end."

"Beyond craftmanship lies invention, and it is here that lean, spare, fast programs are born."

"Representation is the essence of programming."

"Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious."

"Plan to throw one away; you will, anyhow."

The following is the shell script used to crank onto the next motd:

```
mpath=/etc/motds

if [ ! -d $mpath ]
then
     exit 1
fi

if [ -s $mpath/motd ]
then
     cp $mpath/motd /etc/motd
     exit 0
fi

if [ -s $mpath/motd_number ]
then
     motd_number=`cat $mpath/motd_number`
else
     motd_number=0
fi

motd_number=`expr $motd_number + 1`
if [ ! -s $mpath/motd$motd_number ]
then
     motd_number=0
fi

cp $mpath/motd$motd_number /etc/motd
echo $motd_number > $mpath/motd_number
```

Proposed Changes to C

Tim Long
(timl:basservax)

October 23, 1981

Work on C at the Basser Department of Computer Science has spawned some ideas on possible improvements to the language's definition. These are our proposals. We solicit reader's thoughts on this matter.

1. Maxima and minima for arithmetic types

It is proposed that two unary operators be introduced. Their syntax is given by the following addenda to the definition of expression. To section 18.1 of The C Programming Language - Reference Manual add:

```
maxof expression
maxof ( type-name )
minof expression
minof ( type-name )
```

The syntax is identical to that of sizeof, but the type of the argument must be arithmetic. These expression elements are resolved at compile time into the maximum/minimum value attainable by the type of the argument. The type of this value is the same as the type of the argument.

2. Bit stream type

It is proposed that a new data type called bits be added, to supercede bit fields. It can also act as a representation for sets. To the type-specifiers in the syntax in section 18.2 of The C Programming Language Reference Manual add:

```
bits-specifier
```

with syntax

```
bits-specifier:
     bits { bits-decl-list }
     bits identifier { bits-decl-list }
     bits identifier

bits-decl-list:
     bits-declaration
     bits-declaration bits-decl-list

bits-declaration:
     range ;
     range identifier ;

range:
     constant-expression
     constant-expression .. constant-expression
```

The syntax of the bits-specifier and the bits-decl-list are analogous to the equivalent sections of struct, union and enum declarations. The declaration of a bit stream defines a type which is seen as a stream of at least as many bits as the maximum value found in the bits-declaration. For example:

```
bits charset
{
     '\0'       null;
     '0'..'9'   numbers;
     'a'..'z'   lowers;
     'A'..'Z'   uppers;
     maxof(char);
};
...
bits charset   a, *p;
```

Unlike Pascal, bit streams can not be operated on as a unit. There are two ways to reference the components of a bit stream. The first is the extraction into a long or an int (whichever is appropriate) of a named field of a bit stream. This is done in the same manner as struct, union and bitfield member references. For example:

```
a.null          /* will be 1 iff the '\0'th bit of a is set*/
p->numbers      /* will be non-zero iff *p has bits representing*/
          /* numbers set                              */
```

The second is the treatment of a bit stream as an array of bits with reference by an index. For example:

```
a.[ch]          /* will be 1 iff the ch'th bit of a is set*/
p->[ch]
```

3. Random initialisation

An additional form of compile time initialisation is proposed to allow the random initialisation of arrays and bit streams. To the initialisers in section 18.2 of The C Programming Language - Reference Manual add:

= set { random-init-list }

and to the initialiser-list add:

set { random-init-list }

where

```
random-init-list:
     random-initialiser
     random-initialiser , random-init-list

random-initialiser:
     range
     range <- constant-expression
```

a range is described as part of the syntax of a bits-specifier.

A random initialisation must apply to a type which can be indexed, such as an array or bit stream. Such an initialisation will cause all elements in the ranges given to be set to the coresponding constant expression, one if no expression if given. For example:

```
bits { maxof(char); } white_space =
set
{
     ' ', '\n', '\t', '\f',
};

int  a[10] =
set
{
     1..3 <- 5;
     0    <- 1;
     4..9 <- 10;
};
```

Profiling the VAX UNIX* kernel

Tim Long
Chris Maltby

This paper summarises some recent work done at the Basser Department of Computer Science in the tuning of our student UNIX system. Profiling of the operating system was carried out and some improvements have been made.

Introduction

When hardware for kernel profiling became available at Basser, we decided to investigate kernel CPU usage. Under heavy loads, our VAX-11/780 is CPU bound. We hoped to improve our machine's performance by channelling our efforts to trimming any overweight routines we might uncover.

Technique

The method and code were taken from work done by Ian Johnstone, Chris Maltby and Greg Rose+. A KW11-P real time clock was installed on the VAX UNIBUS. This clock was used as the system clock while the VAX interval clock was reassigned to produce profiling interrupts every 781 microseconds. Upon receipt of an interrupt from the VAX interval clock the PC is sampled and used as an index into a map of kernel code space. This is the same mechanism used by the profil system call. Interrupts taken while the processor was in user mode were counted on a global basis.

Results

The study was not intended as a formal evaluation of system performance but as a guide for improvement. The following table gives the top ten users of kernel CPU time measured over a period of several days. Unfortunately the machine load was lighter than normal during this period but steps were taken to simulate our normal heavy student load.

---

* UNIX is a Trademark of Bell Laboratories.

+ "A brief note on UNIX system performance" AUUGN Volume 1 number 4, May 1979.

% of all CPU time	% of kernel CPU time	function
1.77	9.22	_iget
1.01	5.25	_dzrint
1.00	5.21	_dzscan
0.91	4.75	_clock
0.91	4.71	_syscall
0.78	4.07	_resume
0.77	3.99	_mxstart
0.54	2.82	_copyout
0.53	2.78	_mxscan
0.44	2.29	_swtch

The three heavily used routines dzrint dzscan, and clock are invoked at regular intervals. They were found to consume about one percent of total CPU time each, independent of the system load. Another heavy consumer of CPU time was syscall, but this is because it is frequently used.

The most suitable case for treatment was the routine iget. This routine maps a device/i-number pair into the address of an in-core structure, fetching a disk i-node if necessary. This is clearly a common operation, performed at least once for each segment of every pathname scanned by the system.

Old trouble spots diagnosed in "A brief note on UNIX system performance", such as getblk, swtch and wakeup have already been under the knives of the authors.

Our response

The function iget linear seached the i-node table to determine if the desired i-node had to be fetched into core. Our i-node table had 500 entries. To reduce the time taken to find an i-node, we added a hash table indexed by a hash of device and i-number. The i-nodes were chained in doubly linked lists hanging off the table. A doubly linked free list is maintained by sewing a thread through the free i-nodes. As shown in the following table, this produced a significant improvement in the performance of iget.

% of all CPU time	% of kernel CPU time	function
1.45	5.32	_dzrint
1.24	4.55	_syscall
1.17	4.28	_resume
1.15	4.22	_putc
1.08	3.94	_clock
1.03	3.77	_dzscan
0.97	3.57	_getc
0.85	3.12	_mxstart
0.81	2.96	_ttyinput
0.73	2.67	_swtch
	...	
0.10	0.37	_iget

No attempt was made to reproduce the system activity of the first test; this table merely shows the dramatic drop in the time consumed by iget. Resume is now scheduled for treatment.

Proposals for a new UNIX* shell

Chris Berry

Basser Department of Computer Science
University of Sydney
(chrisb:graphics)

1. Introduction

This article outlines the reasons why a new shell should be written, and some details of the current proposal.

2. A new syntax

The main problem with the existing shells is that the shell programmer has difficulty remembering all the command structures. Those who do manage to write complex shell scripts are hindered by the fact that many commands are special cases. Most UNIX users never start to program in shell because the command language is too complex and hard to understand. There is no reason why the shell syntax on Level 6 UNIX systems should be totally different from that on Level 7. In fact, the syntax should be identical to minimise the trauma of a change to Level 7. A shell syntax should be consistent, with no special cases, enabling the user to easily remember the command syntax and thus manage to program in shell.

3. Small and efficient

The shell is probably the most commonly used program on UNIX systems, so its size and efficiency are important. As the shell has evolved it has grown in size, complexity and inefficiency. Existing Level 7 shells are too large to run on many small machines. The same shell should be able to run on any UNIX system no matter what the host machine.

4. Features

It is important to be able to fix up mistakes quickly in program development. This is especially true when programming in shell. In cshell‡ a record or history of the previous commands is kept and the user can edit and run them selectively. Although there are problems in the use of history in cshell it is a useful tool and should be included

---

* UNIX is a Trademark of Bell Laboratories.

‡ 'An introduction to cshell' - Bill Joy

in the UNIX shell. Another thing which should be included is a consistent mechanism for manipulating both shell and environment variables. No existing shell has the facility to remove anything from the environment. This is definitely needed. The alias facility (also introduced in cshell) enables the user to define and redefine his commands so that he can easily remember them and often do less typing.

5. Let's hear it for a new shell

A new shell is being written by Chris Berry and a host of others at the University of Sydney. Further documentation on syntax and semantics will be available when they are finished. Mail chrisb:graphics for developments. All criticisms of the existing shells and suggestions for the new shell are welcome. Speak now or forever hold your peace.

6. Shell proposed

The proposed shell flow control constructs are similar to C, such as for, while, if, else, switch. For consistency everything surrounded by '{' and '}' is considered to be a command, and everything surrounded by '(' and ')' is considered to be an expression. A sample command:

```
if (1 > 2)
{
        echo "(1 > 2)"
}
else
{
        echo "(1 > 2) is false"
        echo "try (2 > 1)"
}
```

A '{' '}' pair inside an expression has the value of the exit code of the command. For example

```
if ({mkdir fred} != 0)
{
        echo "Could not mkdir fred"
}
```

Expressions operate on strings, although some operators require that their arguments can be interpreted as integers.

7. Intrinsic functions

The following are functions being considered for addition in the new shell:

o string

The "o" command allows em style open mode operations on the previous command "!string" (as per history facility). The finished product is executed on exit.

nice -num command

The "nice" command executes the command at a lower priority. Nice with no command changes the priority of the shell.

time command

The "time" command executes the given command, and when it finishes a list of the user, system and real time used is listed. If no command is given the times for the shell are given.

set

The "set" command lists all the shell variables along with their value.

unset var

The "unset" command removes the variable from the shell variable list.

unenv var

The "unenv" command removes the variable from the environment (Level 7 only).

export var

The "export" command adds the variable to the environment (Level 7 only).

wait proc

The "wait" command waits for a particular process to finish. If no process number is given it waits until all the immediate children of the current shell have terminated.

alias

The "alias" command sets its first argument to be an alias for the rest. i.e. "alias l ls -l". If no arguments are given a list of all aliases is given.

unalias

The "unalias" command removes its first argument from the alias list.

history

The "history" command lists the currently saved history.

exit num

The "exit" command forces the shell to exit with the status num.

pushd word
popd
swapd word

The above commands maintain a stack of directory path-names. The "pushd" command, if word is present, pushes word onto the stack, otherwise it pushes the current directory on the stack. The current directory is kept in the shell variable "$place". The "popd" command pops the directory off the stack and 'cd's to it. The "swapd" command interchanges the current directory with the top of the stack.

signal val command

The "signal" command controls what happens when the shell receives signal val. Val may be either an integer or the strings "SIGINT", "SIGQUIT" etc. The command is executed every time the shell receives signal val. "SIGIGN" and "SIGDFL" can replace command resulting in the signal being ignored or default respectively.

timeout command

The "timeout" command causes the command to be executed when the shell times out.

8. This is the first and final call for. . .

We want to hear your ideas on the new proposals and any needs you have perceived. But hurry! Implementation will commence soon.

Book review: "Zen and the art of software maintenance"

Jason Catlett

Basser Department of Computer Science
University of Sydney
Australia 2006
(on SUN: jason:basservax)
(in USA: mhtsa!australia!jason)

After the familiar, dry textbooks on programming by Europeans such as Wirth and Dijkstra, "Zen and the art of software maintenance" provides a refreshing insight into the minds of Californian programmers. Instead of dull, technical chapters like "Euclid's algorithm revisited", we find titles such as "I/O: random, sequential and oral", "Gestalt modularization", and "Relationships". The work is probably the first to relate the Californian life-style and leisure ethic to the creative process, treating areas including "Interrupt awareness levels", "Primal scream debug output" and "Cosmic design methodologies".

Australian gurus who maintain Berkeley programs will be relieved to find out that many routines were placed in programs as koans, those thought-provoking, often totally pointless Zen stories through which one attains divine enlightenment ("What is the sound of one hand coding?"). The book reveals certain inconsistencies in Californian tastes; their passion for organic data structures contrasts with the fast food compiler-compiler output which they seem to relish. One of the more extreme theses of the book argues that a perception of the infinite can only be obtained with virtual memory, and that nirvana can be achieved by "thrashing", apparently some form of flagellation practiced by truly hermetic system supervisors. Two appendices, "Cocaine coding: a user's guide" and "Hot tub program verification" end a very worthwhile text.

P.S. This book has not yet been written. The title and chapter headings are copyright by Jason Catlett. Anyone interested in contributing material should mail the author at the above address.

New product release - tpr

tpr is really 'roff' rewritten in C. Those of you who remember this fast, no-frills formatter may also realise that as it was written in PDP-11 assembler, it was not portable to the VAX, and modifications were not viable. Most of the familiar roff directives are supported with the major exception of hyphenation. (Hackers take note.) A few experimental directives have been added. No guarantee is made that they will perform exactly as they did in roff (the roff manual was not sufficiently lucid as to the effects of some).

Since this formatter runs much faster than nroff/troff, the edit/format cycle becomes more bearable for small jobs, e.g. letters to officialdom. For more details, man tpr.

The name? Just wanted to get away from dogged names.

Ken Yap

AS you like it.

Haven't you sometimes thought the software you use is perhaps sub-standard? Do you wonder about those long-winded compilations, those mega-makes which go on, and on, and on like a member of the Conservative Party? I know I have.

Well, I wondered about the VAX assembler, that long known trusted friend 'as'. Back in level 6 days on our PDP-11 there was this really fast program called 'as'. Now we have a VAX, a much faster machine, and everything seemed a lot quicker... or did it? as seemed to be pretty slow, quite a significant part of the big 'cc's that make up those mega-makes.

So I got in boots and all.

Arrgghhhhh! That code! Those kludges! Those contorted curly brackets! Why do people write code like that? How can they write such compiler dependent code? Why can't these people adopt the Australian C Formatting Standard (AS-1756B)? Those error diagnostics!!! What does "RP expected" mean? And a couple of instructions are left out (in particular the exotic REMQTI - pronounced rem-cutie).

So I profiled it. Some things stuck out like a 1 in a MBZ field. Over 20% of the assembler's time is spent in the symbol table lookup routine. Oh dear, it's the 'linked list of half-full hash tables' again! I ripped that out and replaced it with the standard horrendously efficient hash table of binary trees.

Things were a bit better now. That squeezed 10% out of it. Another profile revealed sorting of the symbol table was causing infringements of the NSW electricity restrictions. The sorting was done so that the neat Unix pseudo-branches (JBR, JEQL etc) could be resolved.

Rather than just sorting the branches and their targets the silly thing sorted the whole table!

Things were looking a bit confusing at this stage so I applied cosmetic surgery to the source, bringing it in line with ACFS and fixed lots of inefficiencies along the way. The pseudo-diagnostics (like "RP expected") were replaced by diagnostics in English, and miscellaneous other vermin were removed.

I then fixed the symbol table sorting, and SHAZAM, another 10% shaved off.

A further profile revealed a bottleneck in writing out the symbol table. Those keen programmers at UCB had discovered this too and had replaced "stdio" calls with their own I/O routines. Unfortunately they had underestimated function call overhead. I wrote some buffering macros for this I/O with quite a substantial CPU saving. We were now saving from 25% for small files to well over 30% for bigguns.

Well, I was happy now. I could see that the code could still be improved (for example lexical analysis buffering is really absurd) but I think it deserves a rewrite rather than another patch!

Fortune tells me now and then not to patch bad code but to rewrite it - one of those silly textbook writers said that.

My mega-makes are now a bit quicker. Except for that horrible ccom thing... but there is no hope for that.

Bruce Ellis
brucee:basservax

From Datamation November 1981

The system design is elegant but the user interface is not.

THE TROUBLE WITH UNIX

by Donald A. Norman

UNIX is a highly touted operating system. Developed at the Bell Telephone Laboratories and distributed by Western Electric, it has become a standard operating system in universities, and it promises to become a standard for micro and mini systems in homes, small businesses, and schools. But for all of its virtues as a system—and it is indeed an elegant system—UNIX is a disaster for the casual user. It fails both on the scientific principles of human engineering and even in just plain common sense.

If UNIX is really to become a general system, then it has got to be fixed. I urge correction to make the elegance of the system design be reflected as friendliness towards the user, especially the casual user. Although I have learned to get along with the vagaries of UNIX's user interface, our secretarial staff persists only because we insist.

And even I, a heavy user of computer systems for 20 years, have had difficulties: copying the old file over the new, transferring a file into itself until the system collapsed, and removing all the files from a directory simply because an extra space was typed in the argument string. The problem is that UNIX fails several simple tests.

Consistency: Command names, language, functions, and syntax are inconsistent.

Functionality: The command names, formats, and syntax seem to have no relationship to their functions.

Friendliness: UNIX is a recluse, hidden from the user, silent in operation. The lack of interaction makes it hard to tell what state the system is in, and the absence of mnemonic structures puts a burden on the user's memory.

What is good about UNIX? The system design, the generality of programs, the file structure, the job structure, the powerful operating system command language (the "shell"). Too bad the concern for system design was not matched by an equal concern for the human interface.

One of the first things you learn when you start to decipher UNIX is how to list the contents of a file onto your terminal. Now this sounds straight-forward enough, but in UNIX

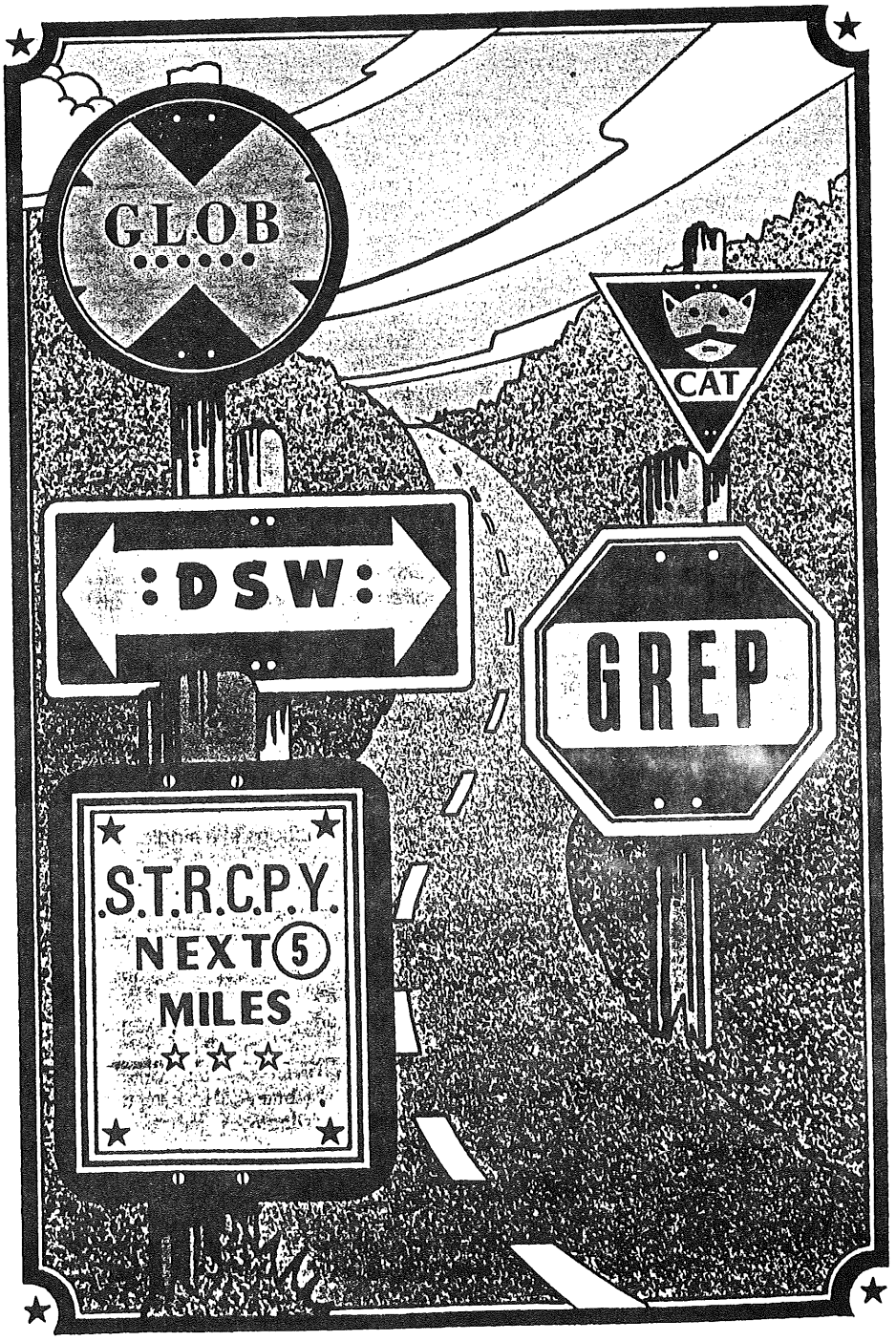


ILLUSTRATION BY CHRIS SPOLLEN

WHAT IS UNIX?

UNIX is an operating system developed by Dennis Ritchie and Ken Thompson of Bell Laboratories. UNIX is trademarked by Bell Labs and is available under license from Western Electric. Although UNIX is a relatively small operating system, it is quite powerful and general. It has found considerable favor among programming groups, especially in universities, where it is primarily used with DEC computers—various versions of the DEC PDP-11 and the VAX. The operating system and its software are written in a high level programming language called C, and most of the source code and documentation is available on-line. For programmers, UNIX is easy to understand and to modify.

For the nonexpert programmer, the important aspect of UNIX is that it is constructed out of a small, basic set of concepts and programming modules, with a flexible method for interconnecting existing modules to make new functions. All system objects—including all I/O channels—look like files. Thus, it is possible to cause input and output for almost any program to be taken from or to go to files, terminals, or other devices, at any time, without any particular planning on the part of the module writer. UNIX has a hierarchical file structure. Users can add and delete file directories at will and then "position" themselves at different locations in the resulting hierarchy to make it easy to manipulate the files in the neighborhood.

The command interpreter of the operating system interface (called the "shell") can take its input from a file, which means that it is possible to put frequently used sequences of commands into a file and then invoke that file (just by typing its name), thereby executing the command strings. In this way, the user can extend the range of commands that are readily available. Many users end up with a large set of specialized shell command files. Because the shell includes facilities for passing arguments, for iterations, and for conditional operations, these "shell programs" can do quite a lot, essentially calling upon all system resources (including the editors) as subroutines. Many nonprogrammers have discovered that they can write powerful shell programs, thus significantly enhancing the power of the overall system.

By means of a communication channel known as a pipe, the output from one program can easily be directed (piped) to the input of another, allowing a sequence of programming modules to be strung together to do some task that in other systems would have to be done by a special purpose program. UNIX does not provide special purpose programs. Instead, it attempts to provide a set of basic software tools that can be strung together in flexible ways using I/O redirection, pipes, and shell programs. Technically, UNIX is just the operating system. However, because of the way the system has been packaged, many people use the name to include all of the programs that come on the distribution tape. Many people have found it easy to modify the UNIX system and have done so, which has resulted in hordes of variations on various kinds of computers. The "standard UNIX" discussed in the article is BTL UNIX Version 6 (May 1975). The Fourth Berkeley Edition of UNIX is more or less derived from BTL UNIX Version 7 (September 1978), with considerable parallel development at the University of California, Berkeley and some input from other BTL UNIX versions. I am told that some of the complaints in the article have been fixed; however, Version 6 is still used by many people.

The accompanying article is written with heavy hand, and it may be difficult to discern that I am a friend of UNIX. The negative tone should not obscure the beauty and power of the operating system, file structure, and the shell. UNIX is indeed a superior operating system. I would not use any other. Some of the difficulties detailed result from the fact that many of the system modules were written by the early users of UNIX, not by the system designers; a lot of individual idiosyncrasies have gotten into the system. It is my hope that the positive aspects of the article will not be overlooked. They can be used by all system designers, not just by those working on UNIX. Some other systems need these comments a lot more than does UNIX.

—D.A.N.

even this simple operation has its drawbacks. Suppose I have a file called "testfile." I want to see what is inside of it. How would you design a system to do it? I would have written a program that listed the contents onto the terminal, perhaps stopping every 24 lines if you had signified that you were on a display terminal with only a 24-line display. UNIX, however, has no basic listing command, and instead uses a program meant to do something else.

Thus if you want to list the contents of a file called "HappyDays," you use the command named "cat":

cat HappyDays

Why cat? Why not? After all, as Humpty Dumpty said to Alice, who is to be the boss, words or us? "Cat," short for "concatenate" as in, take file1 and concatenate it with file2 (yielding one file, with the first part file1, the second file2) and put the result on the "standard output" (which is usually the terminal):

cat file1 file2

Obvious, right? And if you have only one file, why cat will put it on the standard output—the terminal—and that accomplishes the goal (except for those of us with video terminals, who watch helplessly as the text goes streaming off the display).

The UNIX designers believe in the principle that special-purpose functions can be avoided by clever use of a small set of system primitives. Why make a special function when the side effects of other functions will do what you want? Well, for several reasons:

- Meaningful terms are considerably easier to learn than nonmeaningful ones. In computer systems, this means that names should reflect function, else the names for the function will be difficult to recall.
- Making use of the side effects of system primitives can be risky. If cat is used unwisely, it will destroy files (more on this in a moment).
- Special functions can do nice things for users, such as stop at the end of screens, or put on page headings, or transform nonprinting characters into printing ones, or get rid of underlines for terminals that can't do that.

Cat, of course, won't stop at terminal or page boundaries, because doing so would disrupt the concatenation feature. But still, isn't it elegant to use cat for listing? Who needs a print or a list command? You mean "cat" isn't how you would abbreviate concatenate? It seems so obvious, just like:

FUNCTION	UNIX COMMAND NAME
c compiler	cc
change working directory	chdir
change password	passwd
concatenate	cat
copy	cp
date	date
echo	echo
editor	ed
link	ln
move	mv
remove	rm
search file for pattern	grep

Notice the lack of consistency in forming the command name from the function. Some names are formed by using the first two consonants of the function name. Editor, however, is "ed," concatenate is "cat," and "date" and "echo" are not abbreviated at all. Note how useful those two-letter abbreviations are. They save almost 400 milliseconds per command.

Similar problems exist with the names of the file directories. UNIX is a file-oriented system, with hierarchical directory structures, so the directory names are very important. Thus, this paper is being written on a file named "unix" and whose "path" is /cs1/norman/papers/CogEngineering/unix. The name of the top directory is "/", and cs1, norman, papers, and CogEngineering are the names of directories hierarchically placed beneath "/". Note that the symbol "/" has two meanings: the name of the top level directory and the symbol that separates levels of the directories. This is very difficult to justify to new users. And those names: the directory for "users" and "mount" are called, of course,

After all, as Humpty Dumpty said to Alice, who is to be the boss, words or us?

"usr" and "mnt." And there are "bin," "lib," and "tmp" (binary, library, and temp). UNIX loves abbreviations, even when the original name is already very short. To write "user" as "usr" or "temp" as "tmp" saves an entire letter: a letter a day must keep the service person away. But UNIX is inconsistent; it keeps "grep" at its full four letters, when it could have been abbreviated as "gr" or "gp." (What does grep mean? "Global REgular expression, Print"—at least that's the best we can invent; the manual doesn't even try. The name wouldn't matter if grep were something obscure, hardly ever used, but in fact it is one of the more powerful, frequently used string processing commands.)

LIKE CAT? THEN TRY DSW

Another important routine goes by the name of "dsw." Suppose you accidentally create a file whose name has a nonprinting character in it. How can you remove it? The command that lists the files on your directory won't show nonprinting characters. And if the character is a space (or worse, a "*"), "rm" (the program that removes files) won't accept it. The name "dsw" was evidently written by someone at Bell Labs who felt frustrated by this problem and hacked up a quick solution. Dsw goes to each file in your directory and asks you to respond "yes" or "no," whether to delete the file or keep it.

How do you remember dsw? What on earth does the name stand for? The UNIX people won't tell; the manual smiles the wry smile of the professional programmer and says, "The name dsw is a carryover from the ancient past. Its etymology is amusing." Which operation takes place if you say "yes"? Why, the file is deleted of course. So if you go through your files and see important-file, you nod to yourself and say, yes, I had better keep that one. You type in "yes," and destroy it forever. There's no warning; dsw doesn't even document itself when it starts, to remind you of which way is which. Berkeley UNIX has finally killed dsw, saying "This little known, but indispensable facility has been taken over . . ." That is a fitting commentary on standard UNIX: a system that allows an "indispensable facility" to be "little known."

The symbol "*" means "glob" (a typical UNIX name: the name tells you just what it does, right?). Let me illustrate with our friend, "cat." Suppose I want to collect a set of files named paper.1 paper.2 paper.3 and paper.4 into one file. I can do this with cat:

```
cat paper.1 paper.2 paper.3 paper.4>
        newfilename
```

UNIX provides "glob" to make the job even easier. Glob means to expand the filename by examining all files in the directory to find all that fit. Thus, I can redo my command as

```
cat paper*>newfilename
```

where paper* expands to {paper.1 paper.2 paper.3 paper.4}. This is one of the typical virtues of UNIX; there are a number of quite helpful functions. But suppose I had decided to name this new file "paper.all"—pretty logical name.

```
cat paper*>paper.all
```

Disaster. In this case, paper* expands to paper.1 paper.2 paper.3 paper.4 paper.all, and so I am filling up a file from itself:

```
cat paper.1 paper.2 paper.3 paper.4
        paper.all>paper.all
```

Eventually the file will burst. Does UNIX check against this, or at least give a warning? No such luck. The manual doesn't alert users to this either, although it does warn of another, related infelicity: "Beware of 'cat a b > a' and 'cat b a > a', which destroy the input files before reading them." Nice of them to tell us.

The command to remove all files that start with the word "paper"

```
rm paper*
```

becomes a disaster if a space gets inserted by accident:

```
rm paper *
```

for now the file "paper" is removed, as well as every file in the entire directory (the power of glob). Why is there not a check against such things? I finally had to alter my version of rm so that when I said to remove files, they were moved to a special directory named "deleted" and preserved there until I logged off, leaving me lots of time for second thoughts and catching errors. This illustrates the power of UNIX: what other operating system would make it so easy for someone to completely change the operation of a system command? It also illustrates the trouble with UNIX: what other operating system would make it so necessary to do so? (This is no longer necessary now that we use Berkeley UNIX—more on this in a moment.)

THE SHY TEXT EDITOR

The standard text editor is called Ed. I spent a year using it as an experimental vehicle to see how people deal with such confusing things. Ed's major property is his shyness; he doesn't like to talk. You invoke Ed by saying, reasonably enough, "ed." The result is silence: no response, no prompt, no message, just silence. Novices are never sure what that silence means. Ed would be a bit more likable if he answered, "thank you, here I am," or at least produced a prompt character, but in UNIX silence is golden. No response means that everything is okay; if something had gone wrong, it would have told you.

Then there is the famous append mode error. To add text into the buffer, you have to enter "append mode." To do this, you simply type "a," followed by RETURN. Now everything that is typed on the terminal goes into the buffer. (Ed, true to form, does not inform you that it is now in append mode: when you type "a" followed by "RETURN" the result is silence.) When you are finished adding text, you are supposed to type a line that "contains only a . on it." This gets you out of append mode.

Want to bet on how many extra periods got inserted into text files, or how many commands got inserted into texts, because the users thought that they were in command mode and forgot that they had not left append mode? Does Ed tell you when you have left append mode? Hah! This problem is so obvious that even the designers recognized it, but their reaction, in the tutorial introduction to Ed, was merely to note wryly that even experienced programmers make this mistake. While they may be able to see humor in the problem, it is devastating to the beginning secretary, research assistant or student trying to use UNIX as a word processor, an experimental tool, or just to learn about computers.

How good is your sense of humor? Suppose you have been working on a file for an hour and then decide to quit work, exiting Ed by saying "q." The problem is that Ed would promptly quit. Woof, there went your last hour's work. Gone forever. Why, if you had wanted to save it you would have said so, right? Thank goodness for all those other people across the country who immediately rewrote the text editor so that we normal people (who make errors) have some other choices besides Ed, editors that tell you politely when they are working, that tell you if they are in append or command mode, and that don't let you quit without saving your file unless you are first warned, and then only if you say you really mean it.

As I wrote this paper I sent out a message on our networked message system and asked my colleagues to tell me of their favorite peeves. I got a lot of responses, but there is no need to go into detail about them; they all have much the same flavor, mostly commenting about the lack of consistency and the lack of interactive feedback. Thus, there is no standardization of means to exit programs (and because the "shell" is just another program as far as the system is concerned, it is very easy to log yourself off the system by accident). There are very useful pattern matching features (such as the "glob" * function), but the shell and the different programs use the symbols in inconsistent ways. The UNIX copy command (cp) and the related C programming language "string-copy" (strcpy) reverse the meaning of their arguments, and UNIX move (mv) and copy (cp) operations will destroy existing files without any warning. Many programs take special "argument flags" but the manner of specifying the flags is inconsistent, varying

Ed's major property is his shyness; he doesn't like to talk.

ANOTHER VIEW

Prof. Norman praises the UNIX system design but makes a number of caustic remarks about command names and other aspects of the human interface. These might be ignored, since he has no experimental tests to justify them; or they might even be taken as flattery of UNIX, since he does not name any system he likes better; but some of his comments are worth discussing.

Most of the command names Norman points to are indeed strange; some, such as dsw, were removed several years ago (by the way, to repair the discourtesy of the manual, dsw meant "delete from switches"). However, it is not clear that it makes much difference what the command names are. T. K. Landauer, K. Galotti, and S. Hartwell recently tried teaching people a version of the editor in which "append," "delete," and "substitute" were called "allege," "cypher," and "deliberate." It didn't seem to have much effect on learning time, and afterwards the users would say things like "I alleged three lines and deliberated a comma on the last one" just like subjects who had learned the ordinary version of the editor ("A Computer Command By Any Other Name: A Study of Text Editing Terms," available from the authors at Bell Labs.)

In addition to the amusing but secondary discussion of command names, Prof. Norman does raise some significant issues: (1) whether systems should be verbose or terse; (2) whether they should have a few general commands or many special-purpose ones; and (3) whether they should try to anticipate typical mistakes. Experimental results on these issues would be welcome; meanwhile, the armchair evidence is not all on one side.

UNIX is undoubtedly near an extreme of terseness, partly because it was originally designed for slow hardcopy terminals. However, the terseness is very valuable when connecting processes. If the command that lists the logged-on users prints a heading above the list, you can't tell how many users are on by feeding the command output to a line counter. If the editor types acknowledgments now and then, its output may not be directly usable as input somewhere else. Of course, you could feed it through something which strips off the extra remarks, but presumably that program would add its own chatty messages.

Prof. Norman complains about using "cat" for a command which prints files, rather than having a special-purpose command for the purpose (there is one, by the way: "pg"). Having a few general-purpose commands is a definite aid to system learning. In practice, it is not the novices who use the alternatives to "cat"; it is the experts, who want something better adapted to their special needs and are willing to learn another command. In general, people are quite good at recognizing special uses of commands in context, probably because it is a lot like things they have to do every day in English. To take an analogy from programming languages, one doubts that Prof. Norman would advocate a separate operator for "+" in integer arithmetic and "+" in floating point arithmetic. There are many advantages to a small, general-purpose set of commands. Having only one way to do any given task minimizes software maintenance while maximizing the ability of two users to help each other with advice. But this implies that whenever a general command and a specific command do the same thing, the specific command should be removed. It would be a definite service if the "cognitive engineers" could tell us how many commands are reasonable, to give some guidance on, for example, whether "merge" should be a separate command or an option on "sort" (on UNIX it is a sort option) and whether the terminal drivers should be separate commands or options on a graphics output command (on UNIX they are separate). The best rule of thumb we have today is that designing the system so that the manual will be as short as possible minimizes learning effort.

Prof. Norman seems to think that the computer should try to anticipate user problems, and refuse commands that appear dangerous. The computer world is undoubtedly moving in this direction; strong typing in programming languages is a good example. The "ed" editor has warned for some years if the user tries to quit without writing a file. The "vi" editor has an "undo" feature, regardless of the complexity of the command which has been executed. Such a facility is undoubtedly the best solution. It lets the user recognize his mistakes and back out of them, rather than expecting the system to foresee them. It is really not possible to anticipate the infinite variety of possible user mistakes; as every programmer who has ever debugged anything knows, it is hard enough to deal with the correct inputs to a program. Human hindsight is undoubtedly better than machine foresight.

A large number of Prof. Norman's comments are pleas for consistency. UNIX has grown more than it has been built, with many people from many places tossing software into the system. The ability of the system to accept commands so easily is one of its main strengths. However, it results in command names like "finger" for what Bell Labs called "whois" (identify a user) and "more," "cat," or "pg" for what Prof. Norman would rather call "list." The thought of a UNIX Command Standardization Committee trying to impose rules on names is a frightening alternative. Much of the attractiveness of UNIX derives from its hospitality to new commands and features. This has also meant a diversity of names and styles. To some of us this diversity is attractive, while to others the diversity is frustrating, but to hope for the hospitality without the diversity is unrealistic.

—Michael Lesk
Bell Labs
Murray Hill, N.J.

from program to program.

The version of UNIX I now use is called the Fourth Berkeley Edition for the VAX, distributed by Joy, Babaoglu, Fabry, and Sklower at the University of California, Berkeley (henceforth, Berkeley UNIX). This is both good and bad.

Among the advantages: History lists, aliases, a richer and more intelligent set of system programs (including a list program, an intelligent screen editor, an intelligent set of routines for interacting with terminals according to their capabilities), and a job control that allows one to stop jobs right in the middle, start up new ones, move things from background to foreground (and vice versa), examine files, and then resume jobs. The shell has been amplified to be a more powerful programming language, complete with file handling capabilities, if—then—else statements, while, case, and other goodies of structured programming (see box, p. 00).

Aliases are worthy of special comment. Aliases let users tailor the system to their own needs, naming things in ways they can remember; names you devise yourself are easier to recall than names provided to you. And aliases allow abbreviations that are meaningful to the individual, without burdening everyone else with your cleverness or difficulties.

To work on this paper, I need only type the word "unix," for I have set up an alias called "unix" that is defined to be equal to the correct command to change directories, combined with a call to the editor (called "vi" for "visual" on this system) on the f

```
alias unix "chdir /cs1/norman/papers/
    CogEngineering; vi unix"
```

These Berkeley UNIX features have proven to be indispensable: the people in my laboratory would probably refuse to go back to standard UNIX.

The bad news is that Berkeley UNIX is jury-rigged on top of regular UNIX, so it can

There are lots of aids to memory that can be provided, but the most powerful of all is understanding.

only patch up the faults: it can't remedy them. Grep is not only still grep, but there is an egrep and an fgrep.

And the generators of Berkeley UNIX have their problems: if Bell Labs people are smug and lean, Berkeley people are cute and overweight. Programs are wordy. Special features proliferate. The system is now so large that it no longer fits on the smaller machines: our laboratory machine, a DEC 11/45, cannot hold the latest release of Berkeley UNIX (even with a full complement of memory and a reasonable amount of disk). I wrote this paper on a VAX.

LEARNING IS NOT EASY

Learning the system for setting up aliases is not easy for beginners, who may be the people who need them most. You have to set them up in a file called .cshrc, not a name that inspires confidence. The "period" in the filename means that it is invisible—the normal method of directory listing programs won't show it. The directory listing program, ls, comes with 19 possible argument flags, which can be used singly or in combinations. The number of special files that must be set up to use all the facilities is horrendous, and they get more complex with each new release from Berkeley.

It is very difficult for new users. The program names are cute rather than systematic. Cuteness is probably better than standard UNIX's lack of meaning, but there are limits. The listing program is called "more" (as in, "give me more"), the program that tells you who is on the system is called "finger," and a keyword help file—most helpful, by the way—is called "apropos." I used the alias feature to rename it "help."

One reader of a draft of this paper—a systems programmer—complained bitterly: "Such whining, hand-wringing, and general bitchiness will cause most people to dismiss it as over-emotional nonsense. . . . The UNIX system was originally designed by systems programmers for their own use and with no intention for others using it. Other hackers liked it so much that eventually a lot of them started using it. Word spread about this wonderful system, and the rest you probably know. I think that Ken Thompson and Dennis Ritchie could easily shrug their shoulders and say 'But we never intended it for other than our personal use.' "

This complaint was unique, and I sympathize with its spirit. It should be remembered, though, that UNIX is nationally distributed under strict licensing agreements. Western Electric's motives are not altogether altruistic. If UNIX had remained a simple experiment on the development of operating systems, then complaints could be made in a more friendly, constructive manner. But UNIX is more than that. It is taken as the very model of a proper operating system. And that is exactly what it is not.

In the development of the system aspects of UNIX, the designers have done a magnificent job. They have been creative, and systematic. A common theme runs through the development of programs, and by means of their file structure, the development of "pipes" and "redirection" of both input and output, plus the power of the iterative "shell" system-level commands, one can easily combine system level programs into self-tailored systems of remarkable power. For system programmers, UNIX is a delight. It is well structured, with a consistent, powerful philosophy of control and structure.

Why was the same effort not put into the design at the level of the user? The answer is complex, but one reason is the fact that there really are no well known principles of design at the level of the user interface. So, to remedy the harm I may have caused with my heavy-handed sarcasm, let me attempt to provide some positive suggestions based upon research conducted by myself and others into the principles of the human information processing system.

Cognitive engineering is a new discipline, so new that it doesn't exist, but it ought to. Quite a bit is known about the human information processing system, enough that we can specify some basic principles for designers. People are complex entities and can adapt to almost anything. As a result, designers often design for themselves, without regard for other kinds of users.

The three most important concepts for system design are these:

1. Be consistent. A fundamental set of principles ought to be evolved and followed consistently throughout all phases of the design.

2. Provide the user with an explicit model. Users develop mental models of the devices with which they interact. If you do not provide them with one, they will make one up themselves, and the one they create is apt to be wrong.

Do not count on the user fully understanding the mechanics of the device. Both secretaries and scientists may be ignorant of the difference between the buffer, the working memory, the working files, and the permanent files of a text editor. They are apt to believe that once they have typed something into the system, it is permanently in their files. They are apt to expect more intelligence from the system than the designer knows is there. And they are apt to read into comments (or the lack of comments) more than you have intended.

Feedback is of critical importance in helping establish the appropriate mental model and in letting the user keep its current state in synchrony with the actual system.

3. Provide mnemonic aids. For most purposes it is convenient to think of human memory as consisting of two parts: a short-term memory and a long-term memory (modern cognitive psychology is developing more sophisticated notions, but this is still a valid approximation). Five to seven items is about the limit for short-term memory. Thus, do not expect a user to remember the contents of a message for much longer than it is visible on the terminal. Long-term memory is robust, but it faces two difficulties: getting stuff in so that it is properly organized, and getting stuff out when it is needed. Learning is difficult, unless there is a good structure and it is visible to the learner.

There are lots of sensible memory aids that can be provided, but the most powerful and sensible of all is understanding. Make the command names describe the function that is desired. If abbreviations must be used, adopt a consistent policy of forming them. Do not deviate from the policy, even when it appears that a particular command warrants doing so.

System designers take note. Design the system for the person, not for the computer, not even for yourself. People are also information processing systems, with varying degrees of knowledge and experience. Friendly systems treat users as normal, intelligent adults who are sometimes forgetful and are rarely as knowledgeable about the world as they would like to be. There is no need to talk down to the user, nor to explain everything. But give the users a share in understanding by presenting a consistent view of the system. Their response will be your reward. *

Partial research support was provided by Contract N00014-79-C-0323, NR 157-437 with the Personnel and Training Research Programs of the Office of Naval Research, and was sponsored by the Office of Naval Research and the Air Force Office of Scientific Research. I thank the members of the LNR *research group for their helpful suggestions and descriptions of misery. In particular, I wish to thank Phil Cohen, Tom Erickson, Jonathan Grudin, Henry Halff, Gary Perlman, and Mark Wallen for their analysis of* UNIX. *Gary Perlman and Mark Wallen provided a number of useful suggestions.*

Donald A. Norman is professor of psychology and director of the program in cognitive science at the University of California, San Diego. He has degrees in electrical engineering from MIT and the University of Pennsylvania, and a doctorate in psychology from the University of Pennsylvania. He is the author of seven books, including *Human Information Processing*, Academic Press, N.Y., 1977.

BABBAGE – THE LANGUAGE OF THE FUTURE

There are few things in this business that are more fun than designing a new computer language, and the very latest is Ada—the Department of Defense's new supertoy. Ada, as you know, has been designed to replace outmoded and obsolete languages such as COBOL and FORTRAN.

The problem is that this cycle takes 20 to 30 years and doesn't start until we're really convinced present languages are no good. We can short-circuit this process by starting on Ada's replacement right now. Then, by the time we decide Ada is obsolete, its replacement will be ready.

The new generation of language designers has taken to naming its brainchildren after real people rather than resorting to the usual acronyms. Pascal is named after the first person to build a calculating machine and Ada is named after the first computer programmer. As our namesake, we chose Charles Babbage, who died in poverty while trying to finish building the first computer. The new language is thus named after the first systems designer to go over budget and behind schedule.

Babbage is based on language elements that were discovered after the design of Ada was completed. For instance, C.A.R. Hoare, in his 1980 ACM Turing Award lecture, told of two ways of constructing a software design: "One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies." The designers of Babbage have chosen a third alternative—a language that has only obvious deficiencies. Babbage programs are so unreliable that maintenance can begin before system integration is completed. This guarantees a steady increase in the dp job marketplace.

Like Pascal, Ada uses "strong typing" to avoid errors caused by mixing data types. The designers of Babbage advocate "good typing" to avoid errors caused by misspelling the words in your program. Later versions of Babbage will also allow "touch typing," which will fill a long-felt need.

A hotly contested issue among language designers is the method for passing parameters to subfunctions. Some advocate "call by name," others prefer "call by value." Babbage uses a new method—"call by telephone." This is especially effective for long-distance parameter passing.

Ada stresses the concept of software portability. Babbage encourages hardware portability. After all, what good is a computer if you can't take it with you?

It's a good sign if your language is sponsored by the government. COBOL had government backing, and Ada is being funded by the Department of Defense. After much negotiation, the Department of Sanitation has agreed to sponsor Babbage.

No subsets of Ada are allowed. Babbage is just the opposite. None of Babbage is defined except its extensibility—each user must define his own version. To end the debate of large languages versus small, Babbage allows each user to make the language any size he wants. Babbage is the ideal language for the "me" generation. The examples that follow will give some idea of what Babbage looks like.

Structured languages banned GOTOs and multiway conditional branches by replacing them with the simpler IF-THEN-ELSE structure. Babbage has a number of new conditional statements that act like termites in the structure of your program:

WHAT IF—Used in simulation languages. Branches before evaluating test conditions.

OR ELSE—Conditional threat, as in: "Add these two numbers OR ELSE!"

WHY NOT?—Executes the code that follows in a devil-may-care fashion.

WHO ELSE?—Used for polling during I/O operations.

ELSEWHERE—This is where your program really is when you think it's here.

GOING GOING GONE—For writing unstructured programs. Takes a random branch to another part of your program. Does the work of 10 GOTOS.

For years, programming languages have used "FOR," "DO UNTIL," "DO WHILE," etc. to mean "LOOP." Continuing with this trend, Babbage offers the following loop statements:

DON'T DO WHILE NOT—This loop is not executed if the test condition is not false (or if it's Friday afternoon).

DIDN'T DO—The loop executes once and hides all traces.

CAN'T DO—The loop is pooped.

WON'T DO—The cpu halts because it doesn't like the code inside the loop. Execution can be resumed by typing "May I" at the console.

MIGHT DO—Depends on how the cpu is feeling. Executed if the cpu is "up," not executed if the cpu is "down" or if its feelings have been hurt.

DO UNTO OTHERS—Used to write the main loop for timesharing systems so that they will antagonize the users in a uniform manner.

DO-WAH—Used to write timing loops for computer-generated music (Rag Timing).

Every self-respecting structured language has a case statement to implement multiway branching. ALGOL offers an indexed case statement and Pascal has a labeled case statement. Not much of a choice. Babbage offers a variety of case statements:

The JUST-IN-CASE Statement—For handling afterthoughts and fudge factors. Allows you to multiply by zero to correct for accidentally dividing by zero.

The BRIEF CASE Statement—To encourage portable software.

The OPEN-AND-SHUT CASE Statement—No proof of correctness is necessary with this one.

The IN-ANY-CASE Statement—This one always works.

The HOPELESS CASE Statement—This one never works.

The BASKET CASE Statement—A really hopeless case.

The Babbage Language Design Group is continuously evaluating new features that will keep its users from reaching any level of effectiveness. For instance, Babbage's designers are now considering the ALMOST EQUALS SIGN, used for comparing two floating point numbers. This new feature "takes the worry out of being close."

No language, no matter how bad, can stand on its own. We need a really state-of-the-art operating system to support Babbage. After trying several commercial systems, we decided to write a "virtual" operating system. Everybody has a virtual memory operating system so we decided to try something a little different. Our new operating system is called the Virtual Time Operating System (VTOS). While virtual memory systems make the computer's memory the virtual resource, VTOS does the same thing with cpu processing time.

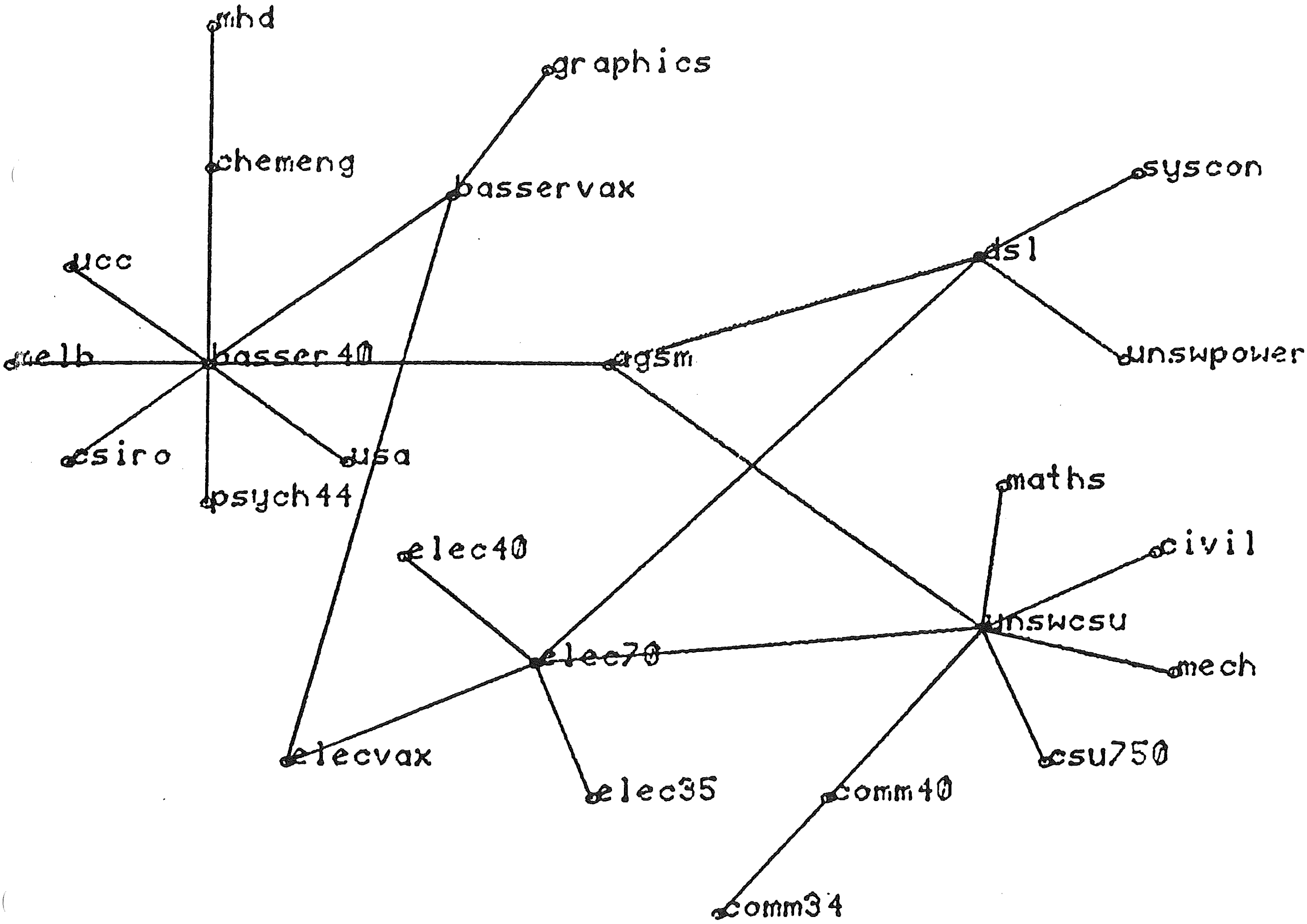
The result is that the computer can run an unlimited number of jobs at the same time. Like the virtual memory system, which actually keeps part of the memory on disk, VTOS has to play tricks to achieve its goals. Although all of your jobs seem to be running right now, some of them are actually running next week.

As you can see, Babbage is still in its infancy. The Babbage Language Design Group is seeking suggestions for this powerful new language and as the sole member of this group (all applications for membership will be accepted), I call on the data processing community for help in making this dream a reality.

—Tony Karp
Jamaica, New York

DATAMATION OCTOBER 1981

NETMAIL NETMAIL NETMAIL NETMAIL NETMAIL NETMAIL NETMAIL NETMAIL NETMAIL NETMAIL



From mhtsa!ianj Thu Dec 3 16:39 EST 1981 netmail from usa
echo HOORAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAH
IANJ

Gooday, Goodevenin, bonjour and Welcome
Break out the champagne
Lite up the Barbee

ie NI NI NI

This is the first try of the French UNIX link via SydneyNet at LERS, Transpac (the French packet switching network, Oh la la), and TELENET.

We don't have an auto dialler, so I have to do that bit by hand. I have the CSIRO source to 'tm'. The only problem to solve now is the re-distribution. I remember clearly your explanation of how this was done. All I need is the details of what to put in the telemail and a copy of the shell scripts you used would be handy.

Useful LERS logins include 'adrian', 'ian' (the boss)
Note: FROGXMIT is "#to.france"
I hope all goes well with you, as it were. How was the champagne??

I await excitedly, the first reply
(hoping that this message didn't stuff things up too much)

ADrian F

echo YIIIIIIIIIIIIIIIIIIIIIIIPPEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE

From peteri Sat Jan 9 14:55:35 1982 netmail from elecvax
To: auugn:basservax
Subject: of interest to our readers?

From mhtsa!duke!decvax!peter Tue Dec 29 19:37:20 1981 netmail from usa

hello peter:
Here is a short summary of the /usr/group meeting held in Boston on the 10th and 11th of December:

- usr/group will hold its next meeting in Boston, this summer with Usenix (in July?) by popular demand from members. The usr/group part of the meeting should occupy the beginning of the week, with Wednesday as a possible overlapping day.
- usr/group has produced a Unix software/services/hardware catalogue that costs $50 for non-members.
- as of 7 Dec 81, usr/group has 519 members.
- the standards committee will attempt to put together a minimal Unix-compatible standard by 12/82 that will at least be V7 and possibly System 3. There may be a couple of enhancements, with the most probable being a file locking sys call, according to one of the steering committee members.
- Doug McIlroy of Bell Labs spoke about getting a ANSI standard committee for C started.
- the summer conference will have considerably more space devoted to exhibitors' booths and could have around a thousand people attending it and Usenix.
- the licensing committee reports that AT&T will now be responsible for Unix licensing, but that the same people will do the job.
- a technical talk was given by Prof. Robert Fabry, of UC Berkeley, about the work they did with Unix; Steve Saperstein of Amdahl talked about Unix on the Amdahl (saying there was a 3 fold productivity increase for programmers who use Unix, using a lines of code per year metric)
- Prof. Fabry also said that they(Berkeley) would be coming out with a distribution based on System 3 fairly soon (implying that they had a prerelease?)
- representatives from Charles River Data Systems, Whitesmiths, and Mark Williams Co., selling ?, Idris, and Coherent, respectively as Unix-compatible products gave technical detail talks about their products. (aside, the Charles River Data Systems product is MC68000 based)
- there were also talks given by people about Ethernet, Office Automation, Networking, and Source Code Control products for Unix.
- the Hilton in Boston is a pretty good hotel!

In other news, Lucasfilm (of Star Wars fame) has a version of Unix running on the MC68000 based sun workstation (from Stanford Uni) and couple of hardware firms have contracted to get themselves versions of Unix for the sun based systems they will sell. The Lucasfilm system was demonstrated at the Comdex show in November.

If you need the technical and licensing details for System 3 Unix let me know and I'll send them along.
Regards to you and your family ... Peter
PS ... we finally got our Berkeley tape ... I got running yesterday with no problems ... half our hardware is still missing! DEC hardware deliveries have promised us the whole thing 3 times and nothing new has arrived since October.
ttfn

Date: Mon Nov 23 10:52:26 1981
From: peteri at elecvax
To: Bob Kummerfeld <bob at basser40>
Subject: some funny mail

From mhtsa!ikeya!alice!dmr Wed Nov 18 02:47 EST 1981 netmail from usa
r a AM-Brights 11-17 0662
?^AM-Brights,650<
^Bright and Brief<

LIVERPOOL, England (AP) _ Tarquin Fintimlinbinwhinbimlim Bus Stop-F'Tang-F'Tang-Ole-Biscuit-Barrel is causing headaches for polling officials at Crosby in Liverpool in a vital special parliamentary election.

The 22-year-old student was John Desmond Lewis until he legally changed his name for 50 pence (96 cents) _ as all Britons are entitled _ to run in the election as a joke candidate representing the pranksters of Cambridge University Raving Looney Society.

There are eight other candidates in the Nov. 26 election, which has attracted nationwide attention because the favorite to take the seat from the ruling Conservatives is Shirley Williams, joint leader of Britain's new Social Democratic Party.

But Tarquin said: ''I am a non-political candidate. I am, simply, very silly.''

Mayor William Bullen, who will have to read the full election results before millions of television viewers, said, ''This is ridiculous. He may think it's a joke but an election is a very serious matter.''

Polling officials, who already have had to use special small print to squeeze Tarquin's name on the ballot, are now scanning election rules to see if there is any way Bullen can avoid having to read out the candidate's full name.

___=

EASTON, Md. (AP) _ Kemal Cem Sekip Deniz isn't happy with his name, so the 21-year-old plastics worker at the local Black and Decker manufacturing plant is seeking to change it to Turkmenogluen Yilmaz Barburoglu Saras Deniz.

And he says he's not worried that few people will be able to pronounce it, just so long as the new name reflects his cultural heritage.

Deniz was born in Seaford, Del., in 1960, of a Turkish neurosurgeon now living in Turkey and a Russian woman who now is a housewife in New York City.

Deniz said his current name, which is Turkish, does not reflect his Russian heritage, so he filed a petition in Talbot County Circuit Court last week asking to drop all but his last name and add four new ones, two of which are Russian.

The petition said Deniz wants the change ''because he favors the policies of the Soviet Union,'' but he said his heritage was really the issue.

''I take pride in my mother's side of the family,'' he said, ''and I take pride in my father's side of the family.''

Turkmenogluen Yilmaz, the first two names he wants to adopt, are Turkish, Deniz said, and the next two are Russian. They are first and middle names used by aunts and uncles, he said.

From keith Wed Dec 16 10:21:12 1981
To: abby
Subject: Unix Counsellor

> From abby Tue Dec 15 21:12:40 1981
> To: paul keith doug paull judy agb
> Subject: Unix Counsellor
>
> I am Abby, of "Dear Abbey" fame, instituted to provide an interesting column in this month's AUUGN (Unix Users Group newsletter). I would be pleased to receive any letters from you, or from anyone you may care to direct to write to me. Emotional letters prefered. Please include some reference to Unix. Mail abby:basservax.

Dear Abby,

I received your system wide message yesterday, and now today a personal letter! What prompted such a well known media personality such as yourself to write to little ol' me? Not that I'm not grateful, mind you - I haven't received much mail since all our darling students left us for the long vacation.

regards, keith.

PS. Is it true that daylight saving fades curtains?

PPS. This letter was prepared on a VAX-11/780 under Unix Level 7.

PPPS. Unix is a trade mark of Bell Laboratories, Murray Hill, N.J.

Dear Keith,

Firstly, on the most important part of your letter. Daylight saving does NOT fade curtains. University tests have proved this conclusively, and researchers are now working on the problem of whether it increases the metabolic rate of suburban grasses. The system wide message was an administrative error, but I hoped to increase my chances of getting a letter from you by addressing you personally. I was disappointed to find the references to Unix in your letter nothing more than cursory. Please check your address and Mail again, this time including some of the outstanding questions received from your hoards of students.

Abby

From doug Thu Dec 17 00:06:45 1981
To: abby
Subject: my life

My life has become an empty shell since I met Unix,
please what can I do?

Dear Doug,

I know a lot of system programmers who would give their flowchart templates to be in your place. You should realise that a user/machine interface is a relationship which both parties have to continuously work at. There's more to command languages than the glossy illustrations in the VMS manual. Sure, you might have started off attracted by Shell's trim, elegant 20 page figure, but even as she aged and started to tend towards OS/360's proportions you must surely be aware that there was some greater commitment, as you together increased your software investment in shell scripts. Doug, why not try harder next time you log in. Use new features, add different options. Go over a few history files together. But remember, operating systems aren't built on REMQTIs or ADAWIs. It's the simple, daily autoincrement MOVLs and SOBGTRs that keep the disks turning. So don't yearn for TOPS-10 or mope around until the level 8 shell is released, try going back home and make the partnership work.

Yours Truly,
Abby

From chris Tue Dec 15 20:38:23 1981
To: abby
Subject: Share and share alike...

From abby Tue Dec 15 20:09:27 1981
To: judy chris
Subject: Share and share alike...

From richardg Tue Dec 15 17:35:59 1981
Dear Abby,

Prove the correctness of the Share scheduling algorithm. In your proof, outline any assumptions you make, in particular detail the metric of fairness used. Also indicate what relationship the percentage share figure has with the real proportion of the machine you are using. For the sake of brevity, could you limit the answer to 20000 words please? I await your pearls pregnantly...
Richard.

Dear Richard, It's perfectly normal at your age to be worried by questions such as the division of wealth, social justice, and the correctness of programs from other universities. I have relayed your question to responsible people. I hope to be able to publish a fuller answer in the next issue of AUUGN.
Abby

Could I just say that this is the first time I've ever been called a responsible person?
Chris

No, I'm sorry, there isn't space.

AUUGN is produced by volunteers from the Australian Unix Users Group. To sustain the fine standard of journalism which our discriminating clientele have come to expect (or will soon, anyway), we solicit material from readers. This means YOU! Yes YOU! Send your material to the editors at the addresses given below. We prefer to be netsent the unformatted file, but hard copies are gratefully accepted and reproduced intact. Also, material should be in the public domain.

Now about that other thing. Money. We need to increase our readership, so get anyone you can to join the Australian Unix Users Group. For a mere $24 you will receive six issues of AUUGN over a period of one year. Please send your cheques in Australian currency. Do not send purchase orders. So mail your subscription fee to this address:

AUUGN
c/o Bob Kummerfeld
Basser Department of Computer Science
Madsen Building F09
University of Sydney
NSW 2006
Australia

Electronic corresponence should be addressed to:

auugn:basservax (on the SUN)
mhtsa!australia!auugn (in USA)