AUSTRALIAN UNIX USERS GROUP NEWSLETTER

In This Issue

Another issue packed with goodies. In this issue you will find a few of the goodies held back last month, material from the Canadian newsletter (Vol I No I), local news and the inevitable mail. Recently I have not asked for contributions, but as the flow of material seems to be slowing, NOW IS THE TIME for all you people out there to START WRITING.

Reading the last issue of AUUGN, subscribers will have noticed the paper by Bill Joy called "Comments on the performance of UNIX on the VAX." This referred to another paper by David Kashtan from SRI, but at the time we were unable to obtain a copy of this paper. We have it now, so now the picture is complete.

If you have ever wished to marry a PDP11 to a VAX, then the paper 'A PDP-11 Front-end for a VAX-11/780' is for you. Sorry about the poor quality of the copy, but a good copy may be obtained from the DECUS conference proceedings.

If anyone out there is interested, we have an edited (note Ianj) transcript of Ian Johnstones view of the last US meeting.

Next User Group Meeting

As most of you will know, the next meeting will be held Wednesday July 2nd at the University Computing Centre, Sydney University. In case you lost the form already mailed to you, there is another on the back of this issue.

This is not to be confused with the world UNIX meeting in Melbourne in October. I will have much more detail on that meeting next issue.

UNIX Licenses

Prospective UNIX licensees often have numerous hurdles to cross on the way to their first 'login:'. Experiences at UNSW as well as elsewhere in Australia where people not entirely knowing what to ask for, who to ask it of, and how much to pay for it when they got it, have lead to a lot time wasting (even by air-mail) correspondence. Irma Biren, Bell's first line of defence in these matters, is as helpful as she can be from that distance. Indeed, Ms Biren has once or twice asked Ian Johnstone to ask the Australian User Group (by an international network mail link) to help a prospective user sort out what they want and the fastest way to get it.

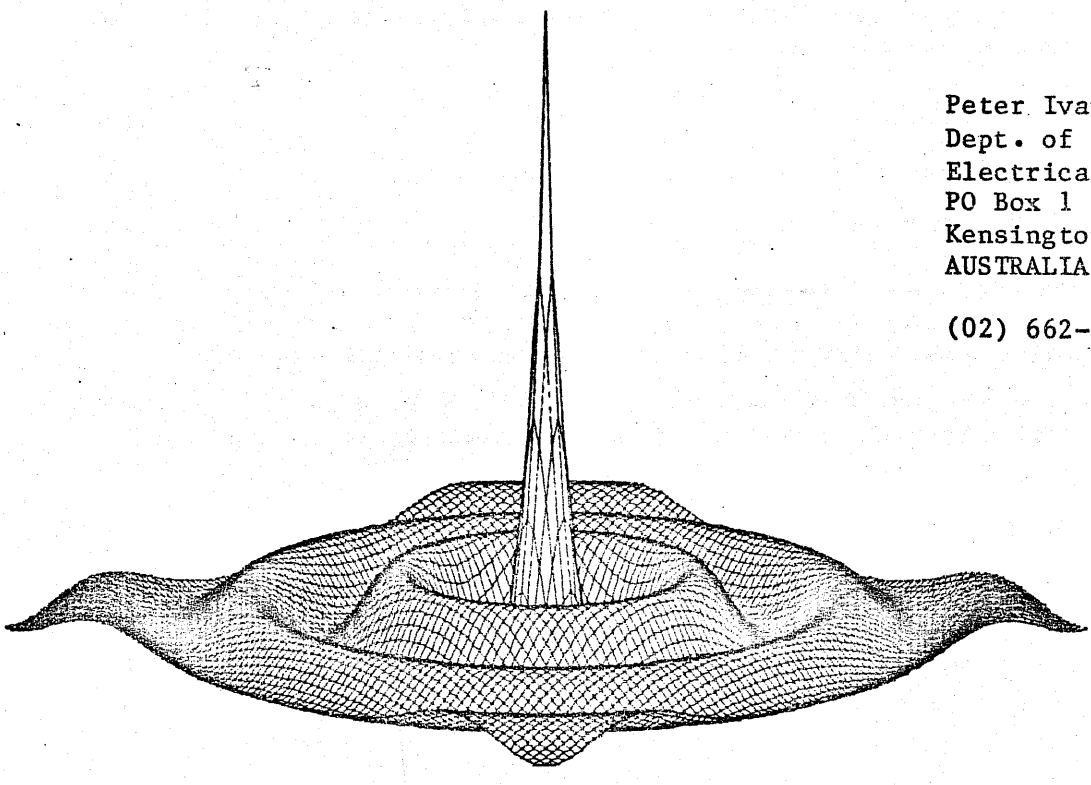
The last time I did this, Ms Biren commented that a wide circulation of letter might do a lot of good. Well, its in this issue. If you know someone who wants a UNIX license, particularly an 'Academic Use' one, then the letter may help smooth the way.

Bi-monthly Moan

The score so far is:

Readership	68
Site Surveys	10
Software Catalogue Offers	7

A Drop in the Ocean



Peter Ivanov
Dept. of Computer Science
Electrical Engineering
PO Box 1
Kensington 2033
AUSTRALIA

(02) 662-3781

GEORGIA TECH
SOFTWARE TOOLS SUBSYSTEM

NEW TOOLS

'Diff' file comparator

Can produce summary of differences between files, editor script to convert one file into another, or formatter input text to produce edition of a document with revision bars

'Xref' Ratfor cross-reference generator

'Rsa' public-key cryptosystem

An implementation of the Rivest-Shamir-Adleman system using exponentiation modulo a composite modulus

'Memo' memorandum handler

Allows the user to specify by a logical expression the times a memo will be displayed and the time at which it will be deleted.

'Stacc' compiler-compiler

Converts an extended BNF grammar with imbedded actions into a parser written in Ratfor.

'Rp' extended Ratfor preprocessor

'Field' columnar data manipulator

Cut-and-paste editing of data fields, insertion of arbitrary strings, etc.

'File' file attribute tester

Tests for existence, length, protection attributes, etc.

'Help' on-line documentation retriever

Select documentation by name of program or subroutine, by pattern match on reference manual index, or from reference manual table of contents.
Also allows retrieval of usage summary.

'Se' screen-oriented text editor

Upward compatible with the line editor from Software Tools. Allows vertical and horizontal cursor motion, character insertion, deletion, transliteration.

'Sh' extended command language interpreter (shell)

Pipeline syntax extended to handle nonlinear process networks; "function calls" allow output of programs to be substituted back into command lines dynamically; "distribution" allows common text to be factored out of many similar commands; "compound nodes" group a number of commands into a unit syntactically equivalent to a command (for use in pipelines, mostly); variables and control structures make extensive programming in the command language easy and effective.

NEW LIBRARY ROUTINES

Binary Input/Output

```
num_words_read    = readf (buffer, buffer_length, file)
num_words_written = writef (buffer, buffer_length, file)
```

File Position Control

```
current_position = markf (file)
success_or_error = seekf (file, position, relative_or_absolute)
```

Conversion Routines

```
length = ctop (string, start_pos, packed_string, maxlength)
value = gctoi (string, start_pos, default_base)
value = ctor (string, start_pos)
value = ctod (string, start_pos)
length = ctov (string, start_pos, char_var_string, maxlength)
length = ctoc (string1, string2, maxlength)

length = ptoc (packed_string, delimiter, string, maxlength)
length = gitoc (value, string, maxlength, ±base)
length = rtoc (value, string, width, decimal_places)
length = dtoc (value, string, width, decimal_places)
length = vtoc (char_var_string, string, maxlength)
```

RATFOR PREPROCESSOR

New Statements

'Select' statement for selection from multiple alternatives
'Linkage' statement to assist separate compilation
'String_table' statement for in-line generation of symbol tables
Multilevel 'break' and 'next'
'Return' statement allows expression to be returned as function value

Internal Procedures

Recursive internal procedures
Compound statements can begin new scopes with local variables

Lexical Changes

In-line character constants: EOS-terminated strings, packed strings, PL/I character-varying
String continuation across line boundaries
Case significant in identifiers
Long variable names with all characters significant
Assignment operators (+=, *=, &=, etc.)
Arbitrary radix integer constants in the style of Algol 68
Keywords are now reserved

Profiling and Tracing

Subroutine execution profile (no. of calls and timing)
Subroutine trace during execution
Statement-level execution profile

Miscellaneous

Macros with named arguments
Automatic inclusion of standard definitions
Short-circuited conditions (&&, ||) allowed anywhere Ratfor parses conditions
Logical expressions are simplified automatically

Implementation

Implemented with compiler-compiler written in Ratfor
Multiple output streams eliminate need for declaration ordering
New variables may be generated without fear of conflict with user variables

SAMPLE 'STACC' INPUT (FROM RATFOR PREPROCESSOR)

```
if_stmt ->
                                    ! integer lab, neglab
                                    ! integer labgen
                                    ! neglab = labgen (1)
                                    ! False_branch = neglab
   par_bool_expr
                                    ? call synerr ("missing condition.")
                                    ? state = ACCEPT
   { NEWLINE }
   ratfor_code
   [  ELSESYM
                                    ! lab = labgen (1)
                                    ! call outgo (lab)
                                    ! call outnum (neglab, CODE)
                                    ? call outnum (neglab, CODE)
      { NEWLINE }
      ratfor_code
                                    ! call outnum (lab, CODE)
      ]
   ;
```

SAMPLE 'SELECT' STATEMENT

```
# patsiz --- return size of pattern entry at pat (n)

   integer function patsiz (pat, n)
   character pat (MAXPAT)
   integer n

   select (pat (n))
      when (CHAR)
         return (2)
      when (BOL, EOL, ANY)
         return (1)
      when (CCL, NCCL)
         return (pat (n + 1) + 2)
      when (CLOSURE)
         return (CLOSIZE)
   else
      call error ("in patsiz:  can't happen.")

   return
   end
```

SAMPLE INTERNAL PROCEDURE

```
define (TREEDEPTH, 100)
define (LEFT (node), Mem (node))
define (DATA (node), Mem (node + 1))
define (RIGHT (node), Mem (node + 2))

procedure treeprint (node) recursive TREEDEPTH
   if (node ~= NIL) {
      treeprint (LEFT (node))
      call putdec (DATA (node), 0)
      call putch (NEWLINE, STDOUT)
      treeprint (RIGHT (node))
      }
```

LONG NAME HANDLING

If a name contains

- more than 6 characters
- an upper case letter
- a zero in position 6

then it must be "uniqued" as follows:

1. If the name has been seen before, simply return its replacement.
2. Pad the name to 6 characters with upper case "A".
3. Map lower case to upper case.
4. Replace the sixth character with a "0".
5. If the new name has never been seen or generated before, enter it as the replacement for the original name.
6. Otherwise, "increment" the second through the fifth characters and go to step 5.

A RUNNING EXAMPLE

input	output
longname	LONGN0
longname2	LONGO0
MEM	MEMAA0
bbbbb0	BBBBB0
longn0	LONGP0
longname	LONGN0

unique name table

longname	LONGN0
longname2	LONGO0
MEM	MEMAA0
bbbbb0	BBBBB0
longn0	LONGP0

must pool their expertise if they hope to keep pace with technical innova-

shows and conferences, meanwhile, will take place Aug. 19-21 at Stanford

... Membership, Com-puter Science Department, Stanford University, Palo Alto, Calif. 94305.

NAME

intro - introduction to TIPs

DESCRIPTION

TIPs (the Tilbrook Information Processing System) is a data base management and an output system for small text data bases. It was designed and created by David M. Tilbrook, while he was at Human Computing Resources, for Soft Arkiv (both in Toronto, Canada), as a demonstration at the 10th Sculpture Conference, which was held in June 1978 at York University, in Toronto.

This manual section will briefly describe TIPs data bases and software. More descriptions are available in the referenced manual sections. A description of the installation of the system is included in this document.

THE DATA BASES

A TIPs data base consists of a descriptive file (henceforth called the profile), and one or more data files. Normally, a data base is contained in a directory of its own. There will be three files called profile.c, .eprofile, and .fprofile. These are the original initialization, and the English and French profiles respectively. The data files should be names '*.d'.

The profile contains the names and structures of all classes of information in the data base. For a full description of profiles and how they are constructed, see Makeprof(TIPs).

The data files are normal UNIX text files (thus they can be edited and created with standard UNIX tools). They can contain one or more complete entries. An entry can be thought of as a set of records describing a single object. For example, in a data base of sculptors, an entry would be the bibliographic information about one artist.

All entries are divided into classes (i.e., records) of information. Each record contains information about one aspect of the entry. For example, in the artist data base, there could be records for the artist's birth date, address, or exhibitions. Each record has a unique two letter identifier (henceforth called the tag).

For the most part, an entry can contain zero or more occurrences of any record. There is no required order for the records. The exception to the above is the 'name' record (the tag is always 'na'). Every entry must begin with an 'na' record, and the occurrence of a subsequent 'na' line indicates the beginning of the next entry.

Records are entered as lines of text. The first four characters of the first record line are an '*', the tag followed by a tab character (ASCII 011). Such lines will hence forth be called 'tag lines.' The text for a record ends at the next tag line or the End of File.

There are three different classes of records. These are: 1) FIELDs; 2) LISTs; and 3) TEXT. The subentry's type is set in the profile.

FIELD records are used to contain information that consists of related parts. For example, in the artist data base, the artist's address was contained in a FIELD record. The fields were street, city, province or state, country, and postal code. In the data files, fields are separated from each other by tab characters, or NEWLINES (ASCII 012). The last field of a FIELDs record can be narrative text. It may contain literal tab characters. (See Makeprof(TIPs) for complete description of the text entry.)

LIST records are used to contain groups of equivalent units of information (e.g., the list of organizations an artist belongs to.) In the data files, a LIST record line can contain as many list elements, separated by tab characters, as desired.

TEXT records are used to contain text. The text begins immediately after the tab following the tag, and continues up to the next tag. Any contained tab characters are kept.

EXAMPLE DATA BASE DIRECTORY

The following could be a list of files in a sample TIPs data base.

.eprofile	English profile
.fprofile	French profile
data01.d	a data file
data02.d	YADF (Yet another data file)
desc	A short description of data base.
profile.c	Profile initialization

The 'desc' file is optional. Its contents are displayed by Query for informational purposes.

A alternative language profile is not required to run the system. The default profile is '.eprofile', thus '.fprofile' can be removed. (Sorry Rene.)

The 'profile.c' file is required by Makeprof(TIPs) only.

The files 'data*.d' are the files containing the TIPs entries. An example entry follows. For readability, tab

characters are shown as ' <TAB> '. Each record in the entry is numbered for reference.

```
1) *na <TAB> bmap <TAB> os/subr.c <TAB> 16
2) *us <TAB> os/rdwri.c <TAB> 46 <TAB> readi
   if ((bn = bmap(ip, lbn)) == 0)
3) *us <TAB> os/rdwri.c <TAB> 98
   writei <TAB> if ((bn = bmap(ip, bn)) == 0)
4) *ky <TAB> filesys <TAB> block
5) *un <TAB> 18-6 <TAB> 6415
6) *sy <TAB> bmap(ip,bn)
7) *de <TAB> bmap defines the structure of file
   system storage by returning the physical block
   number on a device given the inode and the
   logical block number in a file.
   When convenient, it also leaves the physical
   block number of the next block of the
   file in rablock for use in read-ahead.
8) *ky <TAB> extra
```

The 'na' record (#1) is the first record of the entry. The entry continues up to but not including the next 'na' record, or an End of file.

The 'na' record and the 'us' and 'un' records (#2, 3, 5) are all FIELD type records. Each field in the record holds a different piece of information about the item referred to in the record. For example, in the 'us' (Usage) records the contained fields are: the source file name, the source line number, the containing function, and the contents of the line, in that order. Record 3 actually consists of two text lines. The NEWLINE following the source line number field is interpreted as a TAB since the next line is not the start of a new record. It should be noted that there are two 'us' records. All records (with the exception of an 'na') can be repeated as many times as is required.

Records 4 and 8 are both 'ky' (keyword) records. The 'ky' record is a LIST type record. Each field of all the 'ky' records are considered as elements in the 'ky' list. LIST record elements can be entered as individual records or as part of the same record with separating tabs.

The 'sy' and 'de' records (#6 and #7) are TEXT type records. In such records, the content of the record is the text stream upto but not including the next tagged line. For example, record #7 is terminated by the occurrence of record #8. The tabs and newlines in the text are considered part of the text.

THE SOFTWARE

TIPs consists of eight major programs or procedures. These are Query, List, Profinfo, Dped, Mktpl, Tpl, Dumptpl, and

Makeprof.

Query allows a user to interactively scan a data base. It allows entry selection by a variety of techniques, and display of the text under selective control. For example, it is possible to scroll through the data base, selecting entries by query, index, or name, one by one, or a screenfull at a time, and then ask for records or fields to be listed automatically or by command. It should be noted that, in the spirit of Canadian unity, this program is bilingual. See Query(TIPs).

List allows the user to list a data base's contents in a variety of formats. An option allows listing of only entries that match a specified query. Another option checks the data base files for syntax errors. The formats offered by this program are fairly weak. The use of Tpl (see below) will allow the user more power and flexibility in structuring the output. (See List(TIPs)).

Makeprof is the procedure used to create a new profile for a data base. The profile is actually three tables, and a string area, that is written into a file by a program. To create a profile, the user must write the initialization of a C program structure. (A template is provided.) This structure is then compiled by the C compiler (see Cc(I)), and linked with two provided modules. When the resulting binary is executed, the tables are checked for validity, and the tables are written out into the profile files, one for English and another for French.

Profinfo will produce, on standard output, information about the profile. This information is available in a variety of formats. One of the formats is used by Dbed(TIPs) (the Qed TIPs entry editor).

Tpl is a report generating program. The user prepares a template which is compiled and then interpreted for each selected entry in the data base. (See Mktpl(TIPs) for description of template.) The template can contain strings, tag names, output controls, and flow constructs. This program is useful for creating other data bases, or preparing data for processing by other UNIX tools (e.g., Nroff(I)).

Mktpl is used by Tpl to compile the template. The resulting object is stored in a file that can be interpreted by Tpl. It may be run stand alone to syntax check templates.

Dumptpl is a program that will read a Tpl template object file and output the code in a readable form. Its major use was for debugging Mktpl; however, it has not been removed, because it may have a future use.

Dbed is a set of Qed (I) buffers that support the entry and editing of TIPs entries and files.

INSTALLATION

TIPs has been installed on a variety of UNIXs and PDP11s. It does not require any special drivers or system calls. Some difficulty may arise due to the version of Yacc being used, however, conversion between V6 and PWB Yaccs is fairly straight forward. Unfortunately, TIPs does not use Stdio, but it does use a fairly standard Libc. The known differences are that it uses four extentions to Libc. These are: Printd -- printf where first argument is the fid of the target output file; Stringf -- printf where the output is put into the string addressed by the first argument; and Derror -- a perror where the error message is provided, and if the third argument is non-zero exit is called; and a mod to Printf(III) -- a field width beginning with '0' is zero padded.

TIPs should have its own directory which contains directories 'help.d', 'source/mlib', and 'source/olib'.

When the query program is running, the 'help.d' directory should contain: 'short.e', 'short.f', 'commands.e', and 'commands.f'. These are the short and long command descriptions in English and French. Initially 'commands.[ef]' may not exist, but '[ef].comm' (which should) can be Nroff'ed to give them.

The path names for the 'help.d' directory is required in Query and List. All usages are constants which are defined in 'files.h' in the 'hd' directory.

'mlib' and 'olib' are necessary if data bases are going to be created or modified using Makeprof(TIPs). Makeprof is a shell file that links object modules found in these directories.

TIPs uses two magic file numbers (in the first word). These are 043344 in the '.[ef]profile' files, and 043345 in template object files. The appropriate modification to File (I) may be desirable.

SEE ALSO

Query(TIPs), List(TIPs), Makeprof(TIPs), Profinfo(TIPs), Dbed(TIPs), Tpl(TIPs), Mktpl(TIPs), Dumptpl(TIPs), ?/tips/help.d/commands.[ef].

FILES

?/tips/help.d/commands.[ef] - long command descriptions in English & French
?/tips/help.d/[ef].comm - Nroff(1) input for commands.[ef]
?/tips/help.d/short.[ef] - short command table in English &

French
name/profile.c, name/.[ef]profile (name is DB's directory name) - profiles in C, English, and French in the data base's directory.
name/*.d (name is DB's directory name) - data files for DB name.

AUTHOR

David Tilbrook

BUGS

One major problem, that has never been fixed (you know how it is), is that entries are stored in core, but as they are read in, there is no check for overflow. Thus a large entry may overwrite the in-core copy of the profile.

COMMENT

TIPs was done as a demonstration system for non technical people. As such, some parts of the implementation may seem inappropriate or not in keeping with the standard UNIX style. Furthermore, TIPs, like Topsy, just 'grewed', and thus a lot of planned features have never been fully implemented. Some hooks are provided and will be baited when time permits. Good luck, and constructive criticism will be appreciated.

David M. Tilbrook,
247 Brunswick Ave.,
Toronto, Ontario
Canada
M5S 2M6

Telephone: (416) 925-8168

OR

B-N Software Research Inc.,
14th floor,
522 University Ave.,
Toronto, Ontario
M5G 1W7

Telephone: (416) 598-0196

Invitation to a General Access UNIX* Network

Tom Truscott, Duke University

Invitation

A group of UNIX systems at Duke University and the University of North Carolina, Chapel Hill, have established a uucp-based computer communication network. Admission to the net is open to all UNIX licensees. In addition to providing the "uu" services available in the Seventh Edition of UNIX (remote mail, file transfer, job execution), it will provide a network news service. A prospective node must have a call-in facility, call-out facility, or some other means of communication with another UNIX net system. The node must have, or be able to legitimately obtain, uucp and related software.

Systems which do not call-out to the net must be polled occasionally. We will poll any system that so requests, and will bill the polled system for phone costs. The phone costs are expected to be $10-20/month. Requests for an application should be sent to

James Ellis
Department of Computer Science
Duke University
Durham, NC 27706
Telephone: (919) 684-3048

Services

The initially most significant service will be to provide a rapid access newsletter. Any node can submit an article, which will in due course propagate to all nodes. A "news" program has been designed which can perform this service. The first articles will probably concern bug fixes, trouble reports, and general cries for help. Certain categories of news, such as "have/want" articles, may become sufficiently popular as to warrant separate newsgroups. (The news program mentioned above supports newsgroups.)

The mail command provides a convenient means for responding to intriguing articles. In general, small groups of users with common interests will use mail to communicate. If the group size grows sufficiently, they would probably start an additional news group.

*UNIX is a Trademark of Bell Laboratories.

Complete programs and other machine readable text are inappropriate for transmission via (ordinary) news. On the other hand, if a news contributor announces a new version of "x.c", he could be innundated with requests such as "please mail dod!mum a copy of x.c." To avoid that, he could make his copy of x.c directly accessible to anyone via the uucp file copy program. In order to reduce uucp traffic, copies could be made at the more central nodes of the net. Traffic will be reduced further by extending news to support "news on demand." X.c would be submitted to a newsgroup (e.g. "netbulk") to which no one subscribes. Any node could then request the article by name, which would generate a sequence of news requests along the path from the requester to the contributing system. Hopefully, only a few requests would locate a copy of x.c. "News on demand" will require a network routing map at each node, but that is desirable anyway.

It is hoped that USENIX will take an active (indeed central) role in the network. There is the problem of members not on the net, so hardware newsletters should remain the standard communication method. However, use of the net for preparation of newsletters seems like a good idea.

Implementation

The hardware and software requirements for a system to join the net were mentioned above. The uucp system has been retrofitted to run on the Sixth Edition of UNIX, so both Sixth and Seventh Edition Systems can join. Each node must have a unique name, so all names must be cleared by the network administration. Duke will provide the initial administrative functions, and will also provide software for the network news to function.

Although we can supply a news program, individual sites may prefer to adapt their existing programs. This requires the ability to print and receive articles in the news transfer format. The format used to transmit an article between systems is given below. The transferred file consists of a sequence of 0 or more formatted articles, followed by a line consisting of the character `.'. The first character of the article identifies the format and will be used to simplify inevitable changes in article formats. The rest of the first line is a unique system-wide name, which also identifies the originating node. The article name is used to prevent the unlimited duplication of news articles that might otherwise occur. It has a side benefit of simplifying the implementation of a "news on demand" facility. Support for newsgroups (line 2) is required. All network newsgroups will have the prefix "net". Support for contributor name (line 3) and contribution date (line 4) are recommended. A `deletion date' is not supported; it is up to each node to delete ancient news. There is no article title per se, although the first line of the article text could be used for that purpose.

News Article Format

line	example	description
1	Aunc.173	Articles begin with the character `A´. The originating system is `unc´, which gave the article the unique name `unc.173´.
2	netsys7:netnit	Newsgroups to which this article belongs.
3	duke!unc!smb	The net path to the contributor. This line changes as the article passes from system to system.
4	Thu Jan 24 01:39:20 EST 1980	The date the article was submitted to news at the originating system.
5	Delete line 221 of cron.c:	The text of the news article.
6	*cp++ = ´\n´;	
7	Not needed. It confuses ps.	
8	..	Article terminator.

Article and newsgroup names are restricted to 14 or fewer characters. A news transfer file consists of a sequence of formatted news articles followed by `.´ alone on a line.

Questions Answered

1. Won't this be expensive?
Not at all. Night time phone costs are perhaps $0.50/3 minutes, in which time uucp could transfer perhaps 3000 bytes of data (300 baud). Daily polling would then cost $15.00/Month, which is half what Duke pays just for an office phone.

2. Could Duke really handle all the phone calls?
Sure. We have two call-out lines: at five minutes/call, we can handle 24 calls/hour. Other nodes can also volunteer to perform the call-out function.

3. What does Duke get out of this?
We avoid phone charges ourselves, and we get news sooner than anyone else.

4. What about abuse of the network?
In general, it will be straightforward to detect when abuse has occurred and who did it. The uucp system, like UNIX, is not designed to prevent abuses of overconsumption. Experience will show what uses of the net are in fact abuses, and what should be done about them.

   Certain abuses of the net can be serious indeed. As with ordinary abuses, they can be thought about, looked for, and even programmed against, but only experience will show what matters. Uucp provides some measure of protection. It runs as an ordinary user, and has strict access controls. It is safe to say that it poses no greater threat than that inherent in a call-in line.

5. Who would be responsible when something bad happens?
Not us! And we do not intend that any innocent bystander be held liable either. We are looking into this matter. Suggestions are solicited.

6. This is a sloppy proposal. Let's start a committee.
No thanks! Yes, there are problems. Several amateurs collaborated on this plan. But let's get started now. Once the net is in place, we can start a committee. And they will actually use the net, so they will know what the real problems are.

7. Okay, so a few systems get the net started. What next?

SOUTHERN ONTARIO UNIX USERS GROUP MEETING, 15 JAN 1980

Introduction

An informal meeting of UNIX users in the Southern Ontario region was held at the Defence and Civil Institute of Environmental Medicine (DCIEM) on 15 Jan 1980. It was attended by 33 people representing 17 sites distributed from Ottawa and Kingston to London, Ont. The major questions discussed were about relations with DECUS and with USENIX, as well as the organization of the local group. Some matters of substance were also discussed, including problems of Canadian software distribution, local standard utilities, conversion to Edition 7 UNIX, Networking, and problems of UNIX in a commercial environment.

ORGANIZATION AND RELATIONS WITH DECUS AND USENIX

There was considerable discussion as to how the local users would be best organized. To some extent, the discussion proceeded as if the group represented all Canadian users rather than just a local group. It is intended that these notes should reach all Canadian UNIX sites, and comments, criticisms, and offers of aid from those not present at the meeting will be much appreciated.

Some History: UNIX in Canada got off to a fairly slow start, and the relatively few users managed to interchange software and ideas through connections involving mainly the Universities of Toronto, Waterloo and Queens. At the February 1978 meeting of DECUS, 18 members attending a UNIX Birds-of-a-Feather session organized an ad-hoc UNIX DECUS SIG committee of about 5 members, chaired by Martin Taylor of DCIEM. In March 1979, the Canadian DECUS Executive Board recognised a UNIX SIG, but only on the condition that DECUS supply no financial resources. In June 1979, the USENIX meeting was held in Toronto under the auspices of the University of Toronto and DCIEM, but the DECUS SIG as such was not directly involved. To date, the original DECUS UNIX ad-hoc committee has not met, and may be considered moribund. It was to clear up this state of affairs, and to consider the value of working through DECUS and through USENIX that the DCIEM meeting was called.

Discussion: Three main functions of a user group were discussed: local and national meetings, software distribution, and news distribution. UNIX users have been reasonably well served by the twice-yearly USENIX meeting alternating between Eastern and Western North America. It was thought useful, however, to have a more local meeting, which might well be incorporated with the DECUS Canadian Symposium. There was discussion about the cost of attending the DECUS Symposium, contrasting the $125 cost for the 2.5 day meeting with the $20 cost for the 3.5 day USENIX meeting. Although the DECUS meeting has more attendees than the USENIX meeting, it is not significantly larger, and higher attendance should reduce rather than increase the cost per participant. In contrast to DECUS,

USENIX has recently initiated membership fees. USENIX is not subsidized, whereas DECUS is subsidized by DEC. USENIX charges $100 per installation and $12 per individual member. Much of this money goes to pay permanent staff equivalent to the DECUS Chapter and Headquarters staff who are largely on the DEC payroll.

The overall conclusion of the discussion was that there was probably some benefit to a UNIX presence at the DECUS Canada Symposium, but there did not seem to be great enthusiasm for attending at the price.

DECUS has facilities for publishing and distributing SIG Newsletters. Since this is the main means of communicating within a SIG, developing and distributing a newsletter is quite important. There exist UK and Australian UNIX newsletters in addition to ";login:", which goes to all USENIX members. These regional newsletters often carry duplicate items, but also contain matters of local and general interest that are not available elsewhere. It was decided that it would be valuable to enquire of DECUS whether they could distribute a Canadian UNIX SIG Newsletter, and if so, what would the cost be to the SIG. Mike Tilson (Human Computing Resources, 10 St Mary St, Toronto M4Y 1P9) volunteered to act as Newsletter editor on the basis that his job would be to collate camera-ready copy and send it in for distribution.

Installation membership in USENIX provides the member with one copy of each software distribution tape. Receipt of these tapes in Canada has proved awkward because of uncertainties about the behaviour of Canada Customs. It would be much better if the distribution tape went to just one centre and was circulated from there. DECUS in Kanata would be a natural place for this to occur. No UNIX licence would be required, since the distribution function consists in making a simple bit-for-bit copy of the original tape. No media conversion or interpretation is required. Again, it is necessary to determine whether DECUS Canada is willing to perform this function, and if so, at what cost to the SIG.

All attendees at this meeting used DEC computers for running UNIX, so that there would be no bar to their joining DECUS Canada. In the absence of a DECUS US UNIX SIG, US DECUS members could also join according to the DECUS Bylaws. They might cause some problems if included in the software distribution, because of the inverse border problem. Another potential problem for the future involves all the potential UNIX users not running on DEC equipment. Clearly a DECUS SIG cannot be the only grouping of UNIX users in Canada.

Resolution: Having considered all these arguments, the group had a straw vote as to whether to be primarily a USENIX subset or to put some strength into the existing DECUS UNIX SIG. A large majority was in favour of going with DECUS, and all who were not DECUS members agreed to submit applications for DECUS membership. The financial and other implications relating to USENIX and DECUS must be resolved by discussions with those organizations. Clearly, if DECUS agrees to perform the Canadian UNIX software distribution, it will create a financial burden on DECUS and relieve one on USENIX. Possibly the USENIX Board would agree to reduce installation membership fees for Canadian DECUS members. A possible

difficulty in this arises with the DECUS software distribution policy -- distribute on request for a specific fee. The UNIX SIG would not expect programs to be distributed, only complete distribution tapes. These would not ordinarily be distributed on request, although they should be available; they would be distributed to all installation members of the SIG. Some route would have to be found to deal with non-DEC UNIX users if and when they appear.

To consider the questions and to interact with DECUS and USENIX, the meeting selected the following ad-hoc committee:

- Chairman Dr M.M.Taylor, DCIEM, Box 2000, Downsview, Ont M3M 3B9
- Secretary M.I.Tuori, DCIEM, Box 2000, Downsview, Ont M3M 3B9
- Newsletter Editor M.D.Tilson, Human Computing Resources Corp., 10 St Mary St., Toronto, Ont M4Y 1P9
- Member D.Tilbrook, Bell Northern Software Research, 510 University Avenue, Toronto, Ont
- Member D Sherman, Computer Systems Research Group, University of Toronto, 121 St Joseph St., Toronto, Ont.

This list is heavily loaded from Toronto. Members from other parts of Canada are desired, and should anybody wish to belong to the committee or wish to nominate somebody, please let any of us know.

OTHER BUSINESS

Boulder Meeting of USENIX

Previous USENIX meetings have featured a software distribution centre. At the Toronto meeting in June 1979, the input and output demands overwhelmed the capabilities of the machines and volunteers to do the job in the time available. At Boulder, software will be accepted for distribution, and a 4th software distribution will be made in March or April. Greg Hill (U of Toronto) agreed to collate Toronto software for the conference. Greg currently holds copies of all available software distributions (e.g. USENIX 1, 2, and 3; UK, New South Wales, etc), and is willing to circulate them to an extent limited by licencing agreements and volunteer manpower availability.

Steve Pozgaj and Martin Tuori agreed to take notes at the Boulder meeting and report back to the local users. (Underlying this idea was the notion that ";login:" had not appeared for a long time and was not trusted as a medium of rapid communication).

V6 to V7 conversion

There was a short discussion on who was going to convert from V6 to V7, and on local standards for doing so. Most users intend to convert, Greg Hill at U of Toronto (a PWB site), and HCR being exceptions. As for standards, it was emphasised that distribution software should assume a vanilla V7, especially since the Shell is apparently embedded into a lot of V7 software. A lot of makefiles will have to be changed, and some cooperation is possible in this area. Networking should be via V7's *uucp* and *cu*, rather than via the local *yfr* and *cu*.

UNIX in a commercial environment

Steve Pozgaj, of Northern Telecom, raised the question of using UNIX in a commercial environment. A commercial environment is defined as one in which there are a large number of probably naive users, and in which unexpected service interruptions with loss of data are intolerable. The large number of users refers both to the user community and to the number of users on-line at any one time. Data security is also an important consideration, both from the point of view of unauthorized access, and safety when multiple authorized accesses occur simultaneously. To answer this last requirement, Northern Telecom has instituted a scheme for protecting file blocks at different levels. This involves a new system call.

Some response time problems had been observed, and monitoring system performance showed the paradoxical result that raw file I/O was slower than regular file I/O, due to the process being locked in core and limiting the swapping capabilities of the system. Purdue is known to be going for 80 on-line users to an 11/70, as opposed to the normally assumed saturation limit of around 45 users. A major requirement is some sensible replacement for "panic". Several users pointed out that automatic reboot and rebuild systems were running in various installations, so that "panic" usually meant a short interruption of service rather than a damaging delay. Under these circumstances data was seldom lost.

Southern Ontario UNIX Users Meeting

Defence and Civil Institute of Environmental Medicine
1133 Sheppard Ave West
Downsview, Ontario
Friday, April 25, 1980, 13:30 - 16:30

THEME
UNIX in Real Time: Applications and Modifications

Anybody who has anything to say about real-time practice and problems in the use of UNIX, or who has considered or implemented system modifications to improve its real-time performance, please come and talk about it. Please phone DCIEM (416) 633-4240 and ask for Martin Taylor (ext 280), Martin Tuori (ext 204) or Sandra Wright (ext 300), to let us know you are coming, and whether you want some time allotted for you to talk.

Main Speakers:

- *Doug Ross (Andyne Computing, Ltd., Kingston)* – **will discuss the modifications Andyne is developing under contract to DCIEM for time-shared real-time operation. Processes containing real-time phases must declare their parameters to a reservation system. The reservation system can then calculate the feasibility of providing the required responses. At run-time, a mechanism known as a "fuser" must ensure that the real-time process keeps its side of the bargain. Both reservation system and fuser seem to be essential components of a time-shared real-time system.**

- *David Tilbrook (Bell Northern Software Research, Toronto)* – **will talk about the implementation of MASCOT primitives within UNIX. MASCOT is a general system for describing and implementing cooperating parallel task structures, which may run under an operating system, or which may form an operating system on a bare machine.**

Other Business **will be covered at the wishes of the attendees. Let us know of anything you would like to have discussed.**

Getting to DCIEM

By car **– Starting from Highway 401 and the Allen (Spadina) Expressway, go North along the Allen and continue North along Wilson Heights Blvd from the end of the expressway. Turn left at the second traffic light, along Sheppard Ave. (If you are coming from the North or Northeast, get to Sheppard and Wilson Heights and go West on Sheppard). After about 500m Sheppard turns North, and you will see the DCIEM building on your left. Turn left and left again through the first open gate, and park where you can find a spot. The main door is on the West face of the building (facing away from Sheppard), and the meeting is in the room to the right once you get in the door.**

By TTC **– Take a Sheppard West bus from the Sheppard station (Yonge St subway line), or a Keele North or Downsview bus from the Wilson station (Spadina line). Get off when you see the DCIEM building on your left. (If the Sheppard West bus has a label stop at the gate to the DCIEM compound. Other buses stop on Sheppard.) Walk in the gate and find the main door as above.**

***** The minutes of this meeting will appear in the next issue. *****

[OF SOUUGN]

UNIX and VMS
Some Performance Comparisons

by
David L. Kashtan
SRI International

This report describes the results of running various benchmarks on a recently acquired VAX-11/780. The benchmarks were used to compare performance, in specific areas, of the two operating systems now available for the VAX, VAX/VMS version 1.6 and VM UNIX Berkeley version 2.1. The benchmarks were chosen to measure those aspects of the operating systems expected to influence the development and ultimate performance of the ARPA Image Understanding Testbed, for which SRI is the integrating contractor:

a) Access to large images will require both an efficient paging mechanism, for when the image is in memory, and an efficient file access system to bring that data into memory.

b) The current plans for the Image Understanding Testbed call for very heavy communication between many cooperating processes. This will require an efficient, high bandwidth Interprocess Communication facility and an efficient context switching mechanism.

The acceptance testing period for the VAX was a perfect time to attempt this performance evaluation. The system had no users to interfere with the measurements being taken or to complain about the extensive switching between the VMS and UNIX operating systems.

The same programs were run on both systems wherever possible. The differences in the implementations of some facilities neccessitated different programs on the two systems for some of the benchmarks. Where this occured much effort went into filtering out the effects of these differences so that an accurate picture of each system's performance could be formed.

The hardware configuration of the VAX used to run the benchmarks was as follows:

1 VAX-11/780 processor with Floating Point Accelerator
2 Megabytes of memory
1 RP06 disk with massbus adapter
1 TU45 magtape with massbus adapter
2 8-line DZ-11 terminal controllers

The UNIX operating system was run as configured on the distribution tape. There were few parameters in UNIX that could be tuned and the tuning required recompiling the operating system after any changes. It was decided to assume that the distributors knew what they were doing when they configured the system for distribution and that the various system parameters were not unreasonable.

The VMS operating system was run as configured by the software representative of the Digital Equipment Corporation when he installed VMS. One parameter was changed, but only for the paging benchmark, when it was discovered that it had significant effect on the system's performance. The following are some of the major VMS system parameters:

PFCDEFAULT = 64,	the number of pages VMS will attempt to read in a single I/O request.
SYSMWCNT = 64,	the number of pages VMS will use as the working set for the pageable system code and data.
BALSETCNT = 40,	the maximum number of processes VMS will try to keep in memory simultaneously.
VIRTUALPAGECNT=12288,	the largest virtual address space allowed per process (6 megabytes).
QUANTUM = 30 x 10 ms.,	the scheduling quantum (300 milliseconds).
MPW←WRTCLUSTR = 32,	the number of modified pages VMS will attempt to write to the disk in a single I/O request.
MPW←HILIM = 128.	the threshold at which the modified page writer begins writing pages to the disk. (this was changed to 4096 for the paging benchmarks)
MPW←LOLIM = 96,	the threshold at which the modified page writer stops writing pages to the disk.

Paging Performance:

UNIX and VMS have very different paging systems. It may be useful to briefly describe them. The UNIX paging scheme uses a clock algorithm to implement a Least Recently Used page replacement scheme. Page replacement decisions are made globally. Page I/O is performed on a "Page at a time" basis but the software has been used make the page size look like 1024 bytes. (All benchmarks assumed a page size of 512 bytes, which is the hardware page

size of the machine). The VMS paging scheme uses a First in First Out page replacement scheme local to each process (the pages of a process that are currently valid for it to reference is called its "working set"). To offset this, VMS maintains a list of free pages and a list of modified (i.e. must be written to the disk before they may be re-used) pages which are not in any process's working set. These two lists act as a global page cache from which processes may reclaim pages that have been thrown out of their working sets. There are various thresholds which control the sizes of these two lists. One threshold on the modified list triggers the modified page writer to begin writing pages from that list onto the disk and place the pages in the free list. A second threshold stops the modified page writer to keep it from entirely cleaning out the modified list. There is a threshold on the free list which also triggers the modified page writing/swapping mechanism when the available memory becomes low. Pages are regarded as being 512 bytes long but VMS will attempt (when possible) to read and write them in clusters in order to increase the page throughput to the disk. The modified page writer has a cluster parameter which tells it how many pages in the modified list to attempt to gather together in order to transfer them in a single disk request. When reading pages from the disk other cluster factors may be used (these may be specified at a level of detail down to various areas of program or data).

The paging benchmarks were an attempt to characterize how both systems performed when presented with programs that accessed large virtual address spaces (4Mbytes was chosen as the array size since we did not wish to re-compile the UNIX system and since 4Mbytes was considerably larger than the 2Mbytes of physical memory on the VAX). Three types of page access were chosen to represent various IU programs:

1) Sequential page access to a 4Mbyte array was used to simulate programs which make a linear pass over image data, such as simple transformations and picture display routines.

2) Random page access (with a uniform distribution) took

care of the worst case paging activity in which there was no correlation between the current page and the next page to be accessed. This did not measure the worth of the various page replacement policies but did force paging to occur in both systems so that system response to heavy paging could be measured.

3) Random page access (with a gaussian distribution about the current page) was used to simulate more general programs. A small standard deviation in the gaussian distribution simulated programs with considerable locality of reference and a large standard deviation simulated programs with little locality of reference (such as large lisp systems).

The sequential page access program was a loop that touched one byte on each page of the 4Mbyte array in order. This loop repeated 10 times to make sure that a steady state had been reached and that no zero-fill pages were being faulted into the program's physical memory.

VMS	UNIX
4:32.0 Real	20:16.0 Real
39.2 Cpu	1:43.0 Cpu

The stated UNIX times should, in fact, be slightly longer (21:40.0/1:48.0) since the full 4Mbyte array could not be used without re-compiling UNIX. This is the only case where different sizes of benchmarks were used. (The array used in UNIX was 7500 pages vs 8192 pages in VMS). The 5:1 difference in real times can be attributed to VMS making good use of its page clustering scheme. When a fault for a page necessitated reading it from the disk, some of the following pages were read in to memory during the same disk transaction in anticipation of their use. This seems to be a good policy, given that the disk seek and rotational latency times far exceed transfer times. It is worth lengthening the disk I/O request a few milliseconds to bring in extra pages in the hope that they will be needed. One has only to make a correct guess occasionally to make up for the extra time spent in the disk transaction.

The random page access program (linear distribution) was a loop, executed 30,000 times, which picked a page at random in the 4Mbyte array and

wrote a 1 into a location in that page.

VMS	UNIX
6:00.0 Real	17:24.0 Real
31.0 Cpu	1:09.4 Cpu

It was quite a suprise to find the 3:1 difference in execution times. It was expected that the extra work VMS did in bringing in additional pages would be a waste of time. No explanation has yet been found for these results.

!

The random page access program (gaussian distribution) used the same loop as the linear distribution program but called a gaussian distribution random number generator to decide how far from the current page the next page access would be.

In the course of these runs it was discovered that when the MODIFIED page list "high limit" threshold (the point at which the modified page writer is notified) was set higher than the physical memory available on the machine VMS's strategy became a global modified LRU page replacement strategy with results very similar to the paging strategy used on UNIX. This strategy was used for the entire run, although figures are presented (where available) for the low threshold run as a comparison. Two different working set sizes were used in the low threshold runs to see what effect the different working set sizes had on the page fault rate (and therefore the execution time).

An additional run was made with a standard deviation of 10 but with no zero-fill pages (to observe what happens when data must be paged in off the disk). This more closely models a program which pages in its data (e.g. an image).

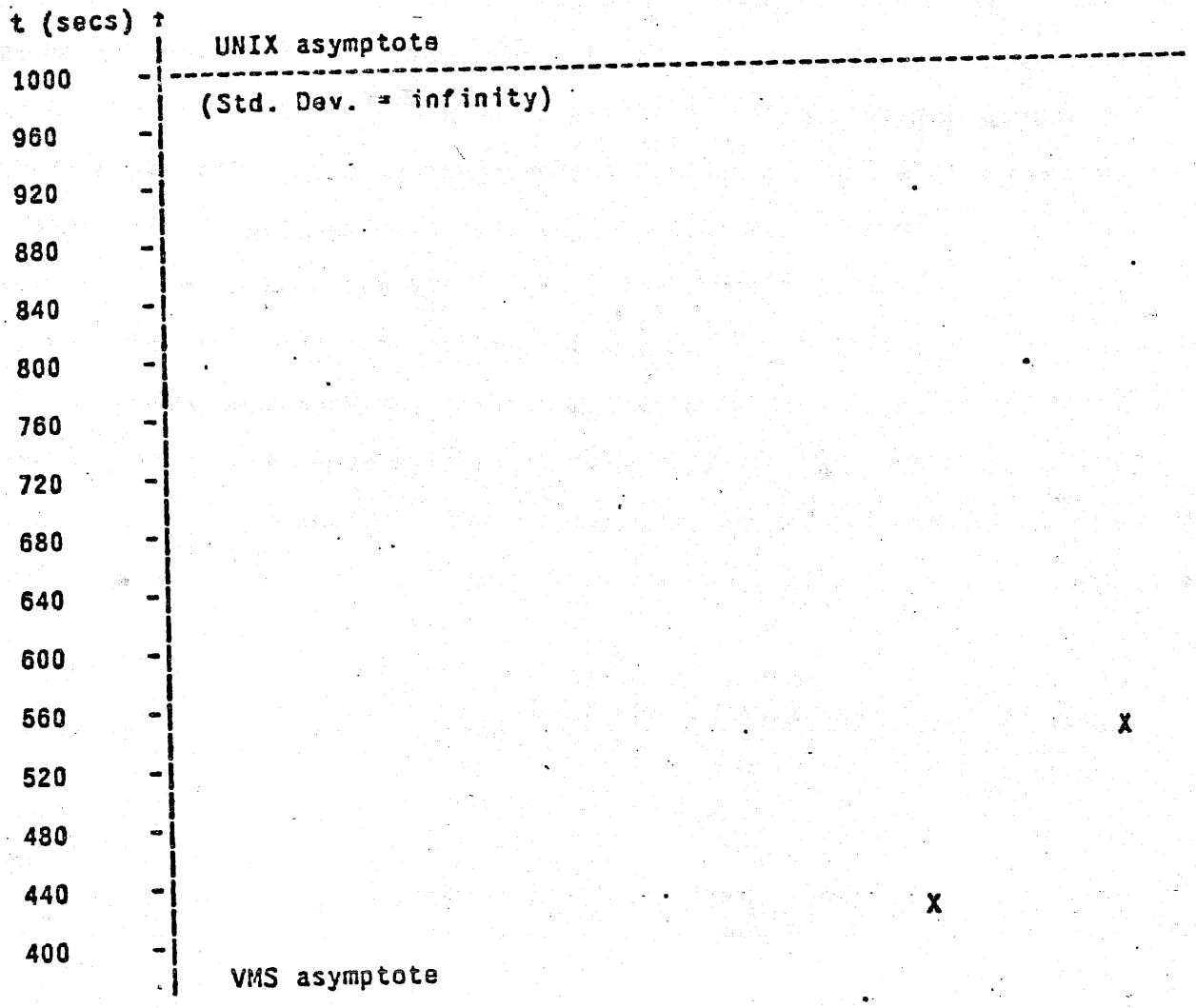
!

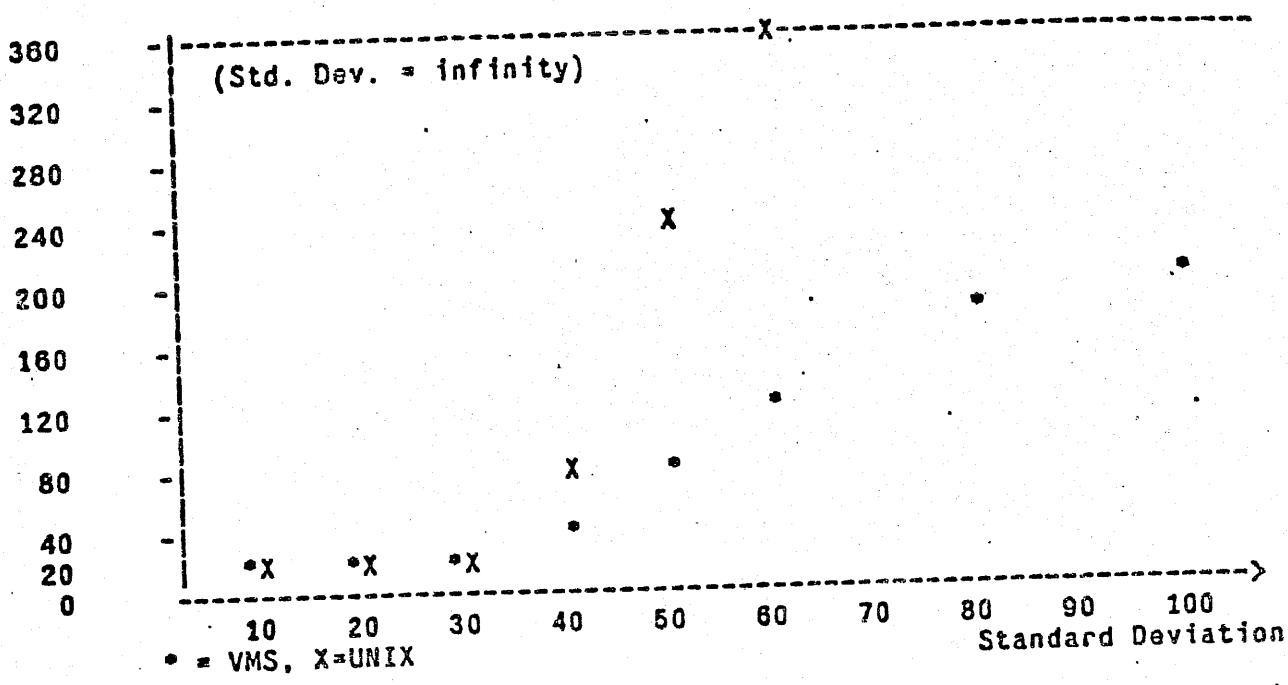
Std. Dev.	VMS(ws=256)	VMS(ws=512)	VMS with big modified list	UNIX
1	15.9 Real 15.5 Cpu	15.9 Real 15.5 Cpu	15.8 Real 15.5 Cpu	15.0 Real 14.8 Cpu
10	22.1 Real 19.8 Cpu	19.8 Real 18.3 Cpu	17.9 Real 17.8 Cpu	16.0 Real 15.6 Cpu
30	--	--	24.7 Real	18.0 Real

	--	--	24.3 Cpu	17.5 Cpu
40	--	--	47.31 Real	1:08.0 Real
	--	--	26.63 Cpu	21.0 Cpu
50	--	--	1:21.0 Real	3:54.0 Real
	--	--	30.0 Cpu	35.0 Cpu
60	--	--	1:53.3 Real	5:46.0 Real
	--	--	32.7 Cpu	39.6 Cpu
80	--	--	3:04.5 Real	6:38.0 Real
	--	--	36.7 Cpu	45.0 Cpu
100	4:24.0 Real	4:23.6 Real	3:27.4 Real	8:26.0 Real
	41.0 Cpu	39.0 Cpu	39.73 Cpu	51.5 Cpu

No zero fill				
10	24.6 Real	21.8 Real	--	26.0 Real
	19.7 Cpu	18.9 Cpu	--	16.4 Cpu

Completion Times vs Standard Deviation





One can compare the gaussian benchmark performance of the two systems under three distinct circumstances:

1) The amount of computation done between page faults is very large. Performance should be very close to the same in both systems when computation time dominates paging time.

2) The amount of computation done between page faults is moderate. We can see this from the completion times of the benchmark, since the new page location must be calculated before the page is accessed. The computation is moderate since a Square root, Cosine and Logarithm must be calculated for each page reference. In fact, the computation time in the benchmark is quite small (30000 accesses in 15 seconds = 500 microseconds/access).

3) The amount of computation done between page faults is very small. We can see this by subtracting off the completion times when the standard deviation is very small (since virtually no paging occurs). The rest of the time is the time spent paging.

Std. Dev.	little(zero) computation (0 microseconds/access) VMS	UNIX	Ratio(speed)	moderate computation (500 microseconds/access) Ratio(speed)
1	0.0	0.0	1:1	1:1
10	2.0	1.0	1:2	1:1.1
30	8.9	3.0	1:3	1:1.3

40	31.5	53.0	1.6:1	1.4:1
50	1:05.2	3:39.0	3:1	3:1
60	1:37.5	5:31.0	3:1	3:1
80	2:48.7	8:23.0	2:1	2:1
100	3:11.6	8:11.0	2:1	2:1
infinity	5:45.0	17:09.0	3:1	3:1

What might better illustrate the behaviour of these two systems is to look at the average time required for each page access given a specific standard deviation. This will show how much computation is required between accesses to make the access time look reasonable:

Std. Dev.	VMS	UNIX
1	0 microseconds/access	0 microseconds/access
10	66 microseconds/access	33 microseconds/access
30	300 microseconds/access	100 microseconds/access
40	1 milliseconds/access	2 milliseconds/access
50	2 milliseconds/access	6 milliseconds/access
60	3 milliseconds/access	10 milliseconds/access
80	5.5 milliseconds/access	12 milliseconds/access
100	6.5 milliseconds/access	16 milliseconds/access
infinity	12 milliseconds/access	34 milliseconds/access

The superior performance of UNIX when the standard deviations of the page references are between 10 and 30 is not quite as good as it may seem from the above tables. Even doing small amounts of computation, as in calculating the gaussian random number, between page accesses reduces UNIX's advantage in these areas to very small quantities (see the moderate computation column). Given a small standard deviation in the page references both systems, in fact, have about equal performance. Once the standard deviation goes up, VMS has a decided advantage. Its ability to sustain high page traffic to and from

the disk makes a great deal of difference in its performance.

UNIX's paging system seems to perform reasonably until it is required to move pages to and from the disk. Its excellent performance at a standard deviation of 30 became very poor when two such programs were run together. Since the standard deviation of 30 made the area of locality of reference about the same size as the physical memory on the machine, UNIX increased the amount of physical memory being used to support the virtual memory of this program and used most of the VAX's memory. Once an identical second program was added to the system, UNIX could no longer allocate all the memory to a single program and it began paging, with the result that the single program execution time rose from 18.0 to 3:51.0.

!
Paging Summary:

The results show that the VMS paging system performance varies from "about as good as" to "much better than" the UNIX paging system. For paging large chunks of data, such as images, into memory it can be about 5 times as fast and for paging programs with moderate to little locality of reference, such as large lisp systems, it can be 3 times as fast (since the paging times will dominate the computation times). When programs have significant locality of reference, both paging systems will perform about equally.

Prospects for further improving the VMS paging system look very good. The various parameters that control the paging system (e.g. the modified list upper and lower thresholds and write cluster size) can by dynamically varied while the system is running. In addition, much information about how well VMS is performing at any time is readily available. One can easily conceive of a process running under VMS which looks at the performance figures and dynamically varies system parameters to improve the system's performance as demands for resources change. There is also code in the system to bring some recency of use information into the working set page replacement policy (now FIFO) which is turned on by information in a process's "process control blk"

and which may be varied to control how much effort is to be spent by the system in acquiring and using recency of use information.

The most distinct difference in performance of the two systems is evident when heavy paging activity is occuring. UNIX's response to other users (other than the program(s) causing the heavy paging activity) can go from very poor (10-20 second response) to non-existant (users can be locked out for minutes at a time, and often until the program causing all the paging activity stops). When heavy paging occurs on VMS, response is only slightly degraded (what seemed like immediate response may lengthen to about 1 second). Very often one can not even notice when a heavily paging program is running.

File I/O Performance:

Since much of a system's performance can be impacted by its ability to move data to and from files, it is useful to measure the abilities of both VMS and UNIX in this regard. Two specific tests were performed, one was a sequential write test and one was a random write test.

The sequential test consisted of writing 4000 blocks of zeros into a new file. The test was run with 512 byte buffers (the size of a single disk sector) but when it was discovered that UNIX was using a block size of 1024 bytes a second run of the test was made with 1024 byte buffers. In addition, the VMS tests were run with various values for the RMS (Record Management Services) multi-block parameter to note its effect on file I/O performance. The RMS multi-block count instructs RMS on how many blocks it is to attempt to read or write in a single I/O operation.

One VMS run used the RMS sequential I/O services and a second run used a FORTRAN program to sequentially write into a random access file (although this introduced the overhead of the FORTRAN runtime system it did test the RMS relative access services).

512 byte buffers

RMS

Multi-Block	VMS (Sequential RMS)	VMS (Relative RMS)	UNIX
44	14.5 Real 11.5 Cpu	8.0 Real 5.0 Cpu	25.0 Real 16.0 Cpu
22	17.0 Real 11.5 Cpu	8.7 Real 5.3 Cpu	
0 (actually 4)	33.1 Real 14.0 Cpu	20.8 Real 7.6 Cpu	

1024 byte buffers

RMS Multi-Block	VMS (Relative RMS)	UNIX
44	8.3 Real 4.4 Cpu	13.0 Real 11.7 Cpu
22	9.6 Real 4.6 Cpu	
0 (actually 4)	23.5 Real 16.6 Cpu	

The multi-block count of 0 actually defaulted to the system multi-block count which was 4. The poorer performance of the RMS sequential I/O services seems to result from RMS placing additional data in the disk file (a record size count) which caused the 512 byte buffer size to actually be 514 bytes. This caused records to straddle sector boundaries. The RMS relative services were given fixed length 512 byte records and they went untouched onto the disk. It is clear that you want to use a reasonable RMS multi-block parameter when doing file I/O (22 seems a good figure as the performance increase at 44 is not very great). The results also show that performance is hardly impacted by the size of the user's buffer. If the buffer size is exceptionally small one would expect the CPU time to rise sharply due to the high number of calls to the RMS service routines.

UNIX greatly benefited from raising the user's buffer size to 1024 bytes. It seems, that to get the best peformance out of the UNIX file I/O

system one's program must be aware of the block size of the disk. UNIX uses a scheme of read ahead & write behind when sequential I/O is being performed. In addition, the disk is initialized to have its free block list in such an order that when blocks are allocated they leave some space between successive blocks in a file so that there is a high likelyhood that the disk head will be in the correct place when the user requires the next block of the file. This scheme does degrade when there are many allocations going on together for different files and when the disk becomes used (as blocks are returned to the free list in the order they are deallocated). All this still does not offset the advantage VMS has due to its multi-block I/O. The VMS run was still 50% to 300% faster. In addition, the heavy I/O to the disk in UNIX had the same bad effects on other users response times as did the heavy paging. These effects were not felt under VMS.

The random test consisted of picking a block at random from an 8,000 block file and writing a buffer to it. This was done 30,000 times. The VMS tests used the RMS relative services only, since the sequential services did not support random access. Two runs were made on each system, one with a block size of 512 bytes and one with a block size of 1024 bytes. The 1024 byte version only used a loop of 15,000 access due to general impatience at length of time the tests were taking to run.

512 byte blocks (30,000 accesses)

RMS Multi-Block	VMS	UNIX
22	11:37.5 Real 1:54.3 Cpu	22:49.0 Real 2:28.3 Cpu
0 (actually 4)	11:40.0 Real 1:57.1 Cpu	

1024 byte blocks (15,000 accesses)

RMS Multi-Block	VMS	UNIX

22	6:23.5 Real 1:14.5 Cpu	8:48.0 Real 1:21.5 Cpu
0 (actually 4)	6:23.0 Real 1:15.0 Cpu	

The gain to UNIX of using the 1024 byte block size was not that great (30,000 accesses = 17:36.0 vs 22:49.0) and the loss to VMS was also not that great (30,000 accesses = 12:46.0 vs 11:40.0). It is, once again, quite surprising that, as in the random paging test, the extra data written to and read from the disk by VMS did not impact its performance. Thus, VMS random access I/O performance can be from 40% to 200% better UNIX. The heavy disk I/O under UNIX had the same effect on other users as in the previous test but the VMS effects were very small.

!
File I/O summary:

The sequential and random I/O tests show that VMS will outperform UNIX in file I/O. In addition, VMS is considerably less sensitive to the size of the user's I/O buffer. It is advisable, though, to not use a system default RMS multi-block count of 4, 22 should provide much better performance. The greatest result in favor of VMS here is the fact that while it it is providing better file I/O performance,other users do not find that the system's response to their requests becomes excessively long when some program is doing heavy file I/O.

!
Context Switching Performance:

The context switching benchmark was meant to measure the performance of both systems when there are many cooperating processes which quickly pass control from one to another, as is expected to be the case for the ARPA Image Understanding Testbed project. The benchmark program was designed to measure how quickly each system's scheduler can save the context of one process, find the next process to be run and restore its context. It consisted of a process which signaled another process and waited for that process to signal it back

The second process waited for a signal from the first process and then sent the signal back. This was repeated 100,000 times on VMS but only 10,000 times on UNIX (the UNIX scheduler was slow enough that it became desireable to limit this loop in order to have the test finish in reasonable time). The intent of this was to force the scheduler to switch back and forth between the two processes as quickly as possible. Each pass through the loop required two switches by the system. The same program was run having it signal itself instead, to measure the time required to do the signaling so that scheduling time could be determined by subtracting the signaling time from the total time.

VMS

100,000 x 2 schedules in 1:43.8 = 2000 switches/second. When run alone, the program took 34.1 seconds. The two programs together took 1:08.2 seconds, leaving 35.8 seconds spent in the scheduler. This is equivalent to a maximum switching rate of 5600 switches/second.

UNIX

10,000 x 2 schedules in 1:35.0 = 210 switches/second. When run alone, the program took 24.0 seconds. The two programs together took 48.0 seconds, leaving 47 seconds spent in the scheduler. This is equivalent to a maximum switching rate of 425 switches/second.

UNIX, as currently implemented, has to do considerably more work when scheduling a process. In addition, UNIX must do a context switch to process number 0 in order to make the decision as to which process is to be run next. Even this cannot explain the greater than 10 to 1 difference in the performance of the two systems.

Interprocess Communication Performance:

With cooperating processes, there must be some way for them to communicate with each other. In UNIX this is done by pipes and in VMS it is done with mailboxes. The benchmarks were used to test how quickly the two systems could move data through their interprocess communication facilities. All tests on both systems were run with two different message sizes, 512 bytes and 4 bytes, to determine if the size of the message would affect the throughput of the system.

The first test consisted of a single process using the IPC facility to write a message to itself, which it would then read. The process wrote to itself in order to keep any system scheduling overhead out of the results. The IPC write/read combination was repeated 10,000 times in each run.

	VMS	UNIX
512 bytes	27.6 Real (370/sec) 27.3 Cpu	30.0 Real (333/sec) 29.6 Cpu
4 bytes	22.4 Real (440/sec) 22.0 Cpu	27.0 Real (370/sec) 25.3 Cpu

The second test consisted of two processes in which one process wrote using the IPC facility and the second process read the data from the 1st one, discarding it. This was used to measure the effectiveness of any buffering schemes in the IPC facility.

	VMS	UNIX
512 bytes	39.4 Real (253/sec) 20.4 Cpu for the writer	39.0 Real (256/sec) 18.2 Cpu for the writer
4 bytes	33.6 Real (297/sec) 18.0 Cpu for the writer	34.0 Real (294/sec) 16.9 Cpu for the writer

The third test had the second process send the message back rather than discard it. Each process, in turn, read and then wrote that same

message to the other process. This forced the system to switch contexts after each write, since each process waited for the message to return after it wrote it. The test was designed to show what would happen when the IPC facility could no longer buffer the data, a situation expected to crop up frequently with a set of processes which cooperate by sending messages.

The very poor performance of both systems prompted a 1/2 hour attempt at constructing an IPC system using the shared page and "common event flag" facilities of VMS.

	VMS (mailboxes)	VMS (shared pages)	UNIX (pipes)
512 bytes	1:07.3 Real 35.8 Cpu (298/sec)	13.0 Real 7.0 Cpu (1540/sec)	1:21.0 Real 37.9 Cpu (246/sec)
4 bytes	55.3 Real 29.2 Cpu (363/sec)	10.0 Real 5.0 Cpu (2000/sec)	1:11.0 Real 32.7 Cpu (281/sec)

The performance of the systems is almost identical (with a slight advantage given to VMS when the messages cannot be buffered), both poor. Surprisingly, the amount of data being pushed through the IPC mechanism on each call has little effect on the number of IPC calls per second both systems will support. It is clear that the kind of IPC facility required by the ARPA Image Understanding testbed project is not currently available on either system.

The performance of the quickly constructed shared page facility, on the other hand, was quite spectacular. This system (as shown in the above table) can outperform both the standard IPC mechanisms in VMS and UNIX by a factor of from 5:1 to 7:1, depending on the message size. The same facility could not be constructed on the current version of UNIX since it lacked both a shared page facility and a true interprocess signaling facility.

Conclusion

The benchmark results show that VMS has a considerable edge over

UNIX in the areas of system performance that are likely to affect the ARPA Image Understanding Testbed. When heavy paging occurs VMS is likely to be between 3 and 5 times faster than UNIX. When heavy file I/O occurs VMS is likely to be between 1.4 and 3 times faster than UNIX. This advantage is augmented by the tendency of UNIX to lock out other users when the disk is in heavy demand. The order of magnitude difference in context switching times and the promise of vastly superior interprocess communication throughput seems to make VMS preferrable for the kind of environment expected of the Image Understanding Testbed.

APPENDIX

BENCHMARK SOURCE PROGRAMS

This appendix contains the source text for the various benchmark programs. A set of numeric functions from the unix library was transferred to VMS to guarantee that the random number sequences were identical. Note that some of the VMS programs were written in Fortran rather than C. This is not because any of the required UNIX system calls (except for fork) would have been difficult to directly map into VMS calls through an appropriate subroutine library, but because we wanted to measure performance using the direct VMS system calls, and because of time pressure to complete the benchmarks. Also note that some of the C programs for the VMS benchmarks make direct calls to VMS system services. It is not intended that portable code directly use such calls, but instead use a subroutine library which implements a much simplier environment.

We intend to repeat these benchmarks using the identical C source code on both systems using a more complete UNIX emulation package for VMS. However, we expect no measurable change in the benchmark timings.

```
!UNIX - sequential page access
------------------------------
main()
{
static char *i;
static char *j;
char *valloc();
int loopcount;
j=valloc(7500*512);
 for (loopcount=0;loopcount<10;loopcount++)
{
  for(i=j;i<j+(7500*512);i=i+512)
        *i=1;
 }
}
```

VMS - sequential page access

```
main()
{
static char memory[8192][512] ;
static char *i;
static char *j = memory;
int loopcount;
 for (loopcount=0;loopcount<10;loopcount++)
{
  for(i=j;i<j+(8192*512);i=i+512)
        *i=1;
 }
}
```

!

UNIX - random page access

```
main()
{
static char *i;
static char *j;
int rand();
char *valloc();
int loopcount,seed=1;
j=valloc(7801*512);
 for (loopcount=0;loopcount<30000;loopcount++)
        *(j+(512*(rand(seed)/275318)))=1;
}
```

VMS - random page access

```
main()
{
static char *i;
static char *j;
int rand();
char *valloc();
int loopcount,seed=1;
j=valloc(7801*512);
 for (loopcount=0;loopcount<30000;loopcount++)
        *(j+(512*(rand(seed)/275318)))=1;
}
```

!

UNIX - gaussian page access

```
#define SD 30.0
main()
{
register int pagenumber,delta;
float gauss();
register char *j;
char *valloc();
register int loopcount;
 j=valloc(7801*512);
 pagenumber=0;
 for (loopcount=0;loopcount<30000;loopcount++)
        {
        delta=gauss(SD,0.0);
```

```
        while ((pagenumber+delta < 0) || (pagenumber+delta >7800))
                delta=gauss(SD,0.0);
        pagenumber=pagenumber+delta;
        *(j+(int)(pagenumber*512))=1;
        }
}
float gauss(SD,MEAN)
float SD,MEAN;
{
float rnd();
float sqrt(),log(),rnd(),cos();
register float qa,qb;
        qa=sqrt(log(rnd())*(-2.0));
        qb=3.14159*rnd();
        return(qa*cos(qb)*SD+MEAN);
}


float rnd()
{
static int seed=1;
static int biggest=0x7fffffff;
int rand();
        return((float)rand(seed)/(float)biggest);
}

!
VMS - gaussian page access
--------------------------
#define SD 30.0
main()
{
register int pagenumber,delta;
float gauss();
static char memory[7800*512];
register char *j=memory;
register int loopcount;
 pagenumber=0;
 for (loopcount=0;loopcount<30000;loopcount++)
        {
        delta=gauss(SD,0.0);
        while ((pagenumber+delta < 0) || (pagenumber+delta >7800))
                delta=gauss(SD,0.0);
        pagenumber=pagenumber+delta;
 /*     printf("page #%d\n",pagenumber);*/
        *(j+(int)(pagenumber*512))=1;
        }
 }
 float gauss(SD,MEAN)
 float SD,MEAN;
 {
 float rnd();
 float sqrt(),log(),rnd(),cos();
 register float qa,qb;
        qa=sqrt(log(rnd())*(-2.0));
        qb=3.14159*rnd();
        return(qa*cos(qb)*SD+MEAN);
 }

 float rnd()
```

```
{
static int seed=1;
static int biggest=0x7fffffff;
int rand();
        return((float)rand(seed)/(float)biggest);
}


!
UNIX - sequential file I/O
--------------------------
main()
{
char buf[512];
int outfile,loopcount;
        outfile=creat("test",0755);
        for(loopcount=0;loopcount<4000;loopcount++)
           write(outfile,buf,512);
}

VMS - sequential file I/O
-------------------------
        byte buf(512)
        open(unit=1,name='test',type='new',access='direct',
     1       initialsize=4000,recordsize=128)
        do 10 i=1,4000
10      write(1'i)buf
        stop
        end
!
UNIX - random file I/O
----------------------
main()
{
long pos;
static char buf[512];
int rand();
int loopcount,seed=1,fd;
 fd=open("test",1);
 for (loopcount=0;loopcount<30000;loopcount++)
 {
     pos=(512*(rand(seed)/275318)) ;
     lseek(fd,pos,0);
     write(fd,buf,512);
     }
}


VMS - random file I/O
---------------------
        byte buf(512)
        integer rand,seed
        data seed /1/
        open(unit=1,name='test',type='old',access='direct',
     1       initialsize=8000,recordsize=128)
        do 10 i=1,30000
        ipos=(rand(seed)/275318)/2+1
10      write(1'ipos)buf
        stop
        end
```

```
!
UNIX - context switching
------------------------
int otherpid;
main()
{
int fork(),getpid(),sigsub();
int (*func)()=sigsub;
int pid,i;
        signal(15,func);
        otherpid=getpid();
        pid=fork();
        if(pid != 0) {otherpid=pid; kill(otherpid,15);}
        for (i=0;i<10000;i++)
        {
        pause();
        }

}

sigsub()
{
static int (*func)()=sigsub;
int signal();
        signal(15,func);
        kill(otherpid,15);
}
UNIX - signal without context switch
------------------------------------
int otherpid;
main()
{
int getpid(),sigsub();
int (*func)()=sigsub;
int pid,i;
        signal(15,func);
        otherpid=getpid();
        kill(otherpid,15);
}

sigsub()
{
static int (*func)()=sigsub;
static int i=0;
int signal();
        if (i++ < 5000) {
        signal(15,func);
        kill(otherpid,15);
        }
}
!
VMS - context switching
-----------------------
        call sys$ascefc(%val(64),'test',,)
        do 10 i=1,100000
        call sys$setef(%val(65))
```

```
        call sys$waitfr(%val(64))                    ! Process 1
        call sys$clref(%val(64))
10      continue
        stop
        end

        call sys$ascefc(%val(65),'test',,)
10      continue
        call sys$waitfr(%val(65))
        call sys$clref(%val(65))                     ! Process 2
        call sys$setef(%val(64))
        goto 10
        stop
        end
```

VMS - signal without context switch

```
        call sys$ascefc(%val(64),'test',,)
        do 10 i=1,100000
        call sys$setef(%val(64))
        call sys$waitfr(%val(64))
        call sys$clref(%val(64))
10      continue
        stop
        end
!
```

UNIX - IPC (write to self)

```
#define n 512
main()
{
char buf[n];
int i,fd[2];
        fd[0]=10000;
        fd[1]=10000;
        pipe(fd);
        for (i=0;i<10000;i++)
        {
                write(fd[1],buf,n);
                read(fd[0],buf,n);
        }
}
```

VMS - IPC (write to self)

```
#define n 512
main()
{
extern sys$crembx(),sys$qiow(),io$←readvblk,io$←writevblk;
char buf[n];
int i,chan;
struct {int nchar;
        char *ptr;
        } name;
        name.nchar=2;
        name.ptr="IN";
        sys$crembx(0,&chan,512,0,0,0,&name);
        for (i=0;i<10000;i++)
        {
```

```
            sysSqiow(0,chan,&io$←writevblk+48,0,0,0,buf,n,0,0,0,0);
            sysSqiow(0,chan,&io$←readvblk,0,0,0,buf,n,0,0,0,0);
          }
}
!
UNIX - IPC (write and discard)
------------------------------
#define n 512
main()
{
char buf[n];
int fork();
int i,fd[2];
        fd[0]=10000;
        fd[1]=10000;
        pipe(fd);
        if (fork() == 0)
                { for (i=0;i<10000;i++)  read(fd[0],buf,n);}
            else
                { for (i=0;i<10000;i++)  write(fd[1],buf,n);}
}

VMS - IPC (write and discard)
-----------------------------
#define n 512
main()
{
extern sysSassign(),sysSqiow(),io$←writevblk,io$←readvblk;
char buf[n];
int i,chan;
struct {int nchar;
        char *ptr;
       }name;
        name.nchar=2;
        name.ptr="IN";
        sysSassign(&name,&chan,0,0);              /* Process 1 */
        for (i=0;i<10000;i++)
         sysSqiow(0,chan,&io$←writevblk+48,0,0,0,buf,n,0,0,0,0);
}

#define n 512
main()
{
extern sysScrembx(),sysSqiow(),io$←writevblk,io$←readvblk;
char buf[n];
int i,chan;
struct {int nchar;
        char *ptr;
       }name;
        name.nchar=2;
        name.ptr="IN";
        sysScrembx(0,&chan,512,0,0,0,&name);     /* Process 2 */
        for (i=0;i<10000;i++)
         sysSqiow(0,chan,&io$←readvblk,0,0,0,buf,n,0,0,0,0);
}
!
UNIX - IPC (write back)
-----------------------
#define n 512
main()
```

```
{
char buf[n];
int fork();
int i,fd[2],fd2[2];
        pipe(fd);
        pipe(fd2);
        if (fork() == 0)
                { for (i=0;i<10000;i++)  {read(fd[0],buf,n);
                                          write(fd2[1],buf,n);
                                          }
                }
            else
                { for (i=0;i<10000;i++)  {write(fd[1],buf,n);
                                          read(fd2[0],buf,n);
                                          }
                }
}
```

VMS - IPC (write back)

```
#define n 512
main()
{
extern sys$assign(),sys$qiow(),io$←writevblk,io$←readvblk;
char buf[n];
int i,chan1,chan2;
struct {int nchar;
        char *ptr;
       }name1;
struct {int nchar;
        char *ptr;
       }name2;
        name1.nchar=2;
        name1.ptr="IN";
        name2.nchar=3;
        name2.ptr="OUT";
        sys$assign(&name1,&chan1,0,0);          /* Process 1 */
        sys$assign(&name2,&chan2,0,0);
        for (i=0;i<10000;i++)
         {
         sys$qiow(0,chan1,&io$←writevblk+48,0,0,0,buf,n,0,0,0,0);
         sys$qiow(0,chan2,&io$←readvblk,0,0,0,buf,n,0,0,0,0);
         }
}
#define n 512
main()
{
extern sys$crembx(),sys$qiow(),io$←writevblk,io$←readvblk;
char buf[n];
int i,chan1,chan2;
struct {int nchar;
        char *ptr;
       }name1;
struct {int nchar;
        char *ptr;
       }name2;
        name1.nchar=2;
        name1.ptr="IN";
        name2.nchar=3;
        name2.ptr="OUT";
```

```
        sys$crembx(0,&chan1,512,0,0,0,&name1);  /* Process 2*/
        sys$crembx(0,&chan2,512,0,0,0,&name2);
        for (i=0;i<10000;i++)
         {
         sys$qiow(0,chan1,&io$←readvblk,0,0,0,buf,n,0,0,0,0);
         sys$qiow(0,chan2,&io$←writevblk+48,0,0,0,buf,n,0,0,0,0);
         }
}
-------
```

A SUMMARY OF THE VAX/VMS VERSUS VAX/UNIX DEBATE

G. H. MacEwen
Queen's University
Kingston, Ontario

D. F. Athersych, D. J. Ross
Andyne Computing Limited
Kingston, Ontario

ABSTRACT

In many research institutions where the VAX-11/780 is being considered, there has been much discussion on the relative merits of two available operating systems: VAX/VMS and VAX/UNIX. The debate centres on issues of support, functionality, flexibility, and maintainability. This paper focuses on the arguments presented for and against each system, and attempts to summarize current thinking.

Introduction

Many research and development laboratories and universities have found the PDP-11/UNIX operating system to be a friendly environment for software development and specialized application work. Many of these same institutions are considering, or have already moved to, the VAX-11 as a hardware base. A major consideration for the research and development community is access to software developed elsewhere and the complementary ability to make one's own software generally available. As a consequence it is of some importance, in the view of many, to have a common operating system in use.

The production of VMS as a time-sharing system for the VAX-11 raises the question as to whether or not it is a viable replacement for UNIX, or more precisely for VAX/UNIX, which has been released by Bell Laboratories. This question has aroused some vigourous debate, especially among installations on the ARPA network, much of which consists of subjective opinions rather than carefully considered evaluation. Nevertheless, some important questions have been raised and debated in informal unpublished documentation.

The unbiased observer (if one can indeed qualify as such) is confronted with masses of technical documentation which requires much time to study, opinions stated from biased viewpoints, and unanswered questions, the solutions to which are hidden somewhere within the documentation, the supplier's organization, or perhaps only in the systems themselves.

This paper represents an attempt to summarize the questions that have been raised in the current VMS/UNIX debate. It is based on copies of generally available informal documentation, limited access to VMS documentation, and rather better access to UNIX documentation. Our hands-on experience with UNIX has been moderate but not extensive and one of us has had a very brief encounter with VMS.

Organization of the Paper

The remainder of the paper is divided into sections each of which examines one aspect of the systems under study. In each section we try to establish some requirements for the particular environment in which we are interested, namely the research and development lab. We then make some comments on VMS and UNIX. Since some of our information has not been confirmed by direct examination, much of the following material should be taken as questions for consideration rather than as conclusions. We try carefully to indicate specifically where our information is accurate.

Maintainability and Modifiability

The ability to examine a system and carefully alter or extend its function is probably the single most important requirement for specialized application and development work. Indeed, it is unlikely that any research facility is not heavily modified. For this to be possible a system must be written in a high-level language and designed in a well modularized structure. Device driver insertion effort is a common measure of this ability.

VMS is written largely in assembler, a surprising fact to almost everyone familiar with current software technology. The system is large although reported to be relatively well structured. Manuals are generally perceived to be verbose and rather varying in level of presentation. Although opinion varies on the effort to install device drivers the weight is on the side of difficulty. It is possible, however, to install a driver dynamically (without rebuilding the entire system).

UNIX is written in C, a relatively unrestraining, but nevertheless high-level language. This single fact has probably

given UNIX its current acceptance. It is relatively small which mitigates against its lack of modularity. As a consequence most people find it quite understandable after a short time. Device driver insertion is not difficult, but does require rebuilding the system.

Utilities

The program used directly more than any other is probably the editor. A good editor can contribute enormously to productivity. Although other utilities are worth considering in an evaluation the editor certainly dominates. A full screen editor with the ability to save the state and invoke the command interpreter is a minimum requirement.

Neither UNIX nor VMS comes equipped with such an editor. Most UNIX installations have written their own and VMS installations are probably doing so. The failure of DEC to provide this capability is also surprising although it may have been influenced by terminal handling within the hardware architecture. Recent hardware announcements may have removed this problem.

Input/Output

In an R&D installation a requirement for I/O support is that it be simple and that it not impose unwanted structure or efficiency constraints. The ability to construct appropriate access methods on top of the standard I/O schema should satisfy a divergent set of application requirements. Viewing an external process as a random addressable sequence of bytes (sequential where required) provides a device independent base for other I/O mechanisms. The option of synchronous or asynchronous I/O operations is desirable.

There must also be the ability for any non-privileged process to have full control over its associated terminal. The variation amongst available terminals and the function available in the more intelligent of these demands such a capability. This includes, of course, a raw untranslated 8-bit byte-stream mode.

The most unfortunate design decision in VMS is the placing of the Record Management System (RMS) at a low level thus requiring its use by all processes regardless of need. RMS imposes a line-oriented access mode on all users and consequently all DEC utility software uses it. Although it can be bypassed with difficulty it remains as the base of almost all VMS software. VMS is claimed by some to be slow for simple tasks (e.g. copying data). It is possible to do asynchronous I/O. Complete terminal control requires a privilege level which also gives access to the entire system including the system disk.

UNIX provides one basic byte-stream access mode and allows full terminal control by unprivileged processes. Asychronous I/O primitives are not provided although some concurrency can be achieved. Lacking, however, are more structured access modes which are useful in certain applications.

Both VMS and UNIX are designed on the principle that there be a common file access mode for all software thus unifying software processing and communication interfaces. The problem for specialized work is that this must be a quite primitive mode.

File Structure

The kinds of installations we are discussing have users with overlapping projects and much interchange and cooperation in software development. A file structure to model this environment can have few structural limitations. The ability to create symbolic links between directories, to share sub-directories, and to associate users with a "home" directory distinct from the current directory are minimal needs. A well defined search sequence for named files that are not found in the current directory allows public software and private software to be uniformly available. Finally, for such dynamic structures reference counting of files should be automatic with space reclaimed upon the reference count reaching zero or the directory being deleted.

File directories can be created to eight levels in VMS. This is probably sufficient although it seems an arbitrarily low number. There are no symbolic references possible between users' structures, and names not found in a user's directory receive no further processing. There are at least two disconcerting problems in addition to these structural problems: A deleted directory results in the file space being lost and new versions of files (all files are maintained as a sequence of historical versions requiring explicit deletion to reclaim space) do not inherit the protection attributes of the old version.

A simple hierarchical file structure characterizes UNIX. A search sequence for named files makes public and sharable software uniformly available although there is no concept of "home" directory. No symbolic directory references or shared sub-directories are possible.

All physical resources can be accessed via the file naming structure giving a very nice uniformity to program interfaces in UNIX. This feature is one of the major reasons for the accepted view of UNIX as a cleanly designed system.

Process Control and Communication

The ability of one process to exercise control over another in a debugging mode is needed for development work. This includes the ability to save a process image as a file and restart it some time later.

For more general structuring of systems as communicating processes the ability to create subprocesses and to communicate with any process on a global name basis is required. Process creation should not be an expensive operation.

Named mailboxes and global events in VMS provide a communication means between any pair of processes. Process creation specifies the image to be executed by the new child.

In UNIX pipes and signals can only be used between a child and its parent and/or its siblings. Pipes, however, appear as files and so a very clean I/O redirection mechanism results. Also, a child can inherit open files making this indirection transparent to a process. Process creation copies the image of the invoking process for execution by the new child.

Neither UNIX nor VMS possesses particularly good parent/child control as suggested above. In each, a particular flavour of process creation has been chosen suggesting that the method be an option to the programmer. A problem in UNIX particularly is that the copying is often unnecessary as the new process typically executes a different program.

Command Language

The user of a multi-process system needs control commensurate with the underlying available mechanism. In particular, a user should be able to exercise active control and communication with a tree of processes. Avoiding waiting time at the terminal can be achieved while at the same time carrying on related tasks each running as a separate process.

The VMS command interpreter, DCL, provides no explicit concurrency mechanism although the system primitives seem to provide sufficient support. The shell in UNIX, on the other hand does provide for "piping" between processes and the creation of multiple processes. The necessary communication channels to and from such processes are not provided however. Although some find the shell to be too cryptic (What does cat mean?) it is a cleanly designed and powerful tool.

System Calls

The interface to operating system routines must, above all, be clean and easy to use. Of almost as much importance is the provision of a uniform interface from a variety of languages.

UNIX has a relatively small set of clean primitives implemented as calls from C. VMS provides a uniform calling sequence from all languages and what some experienced programmers feel is a good error return mechanism. The call interface, however, is very complex and, consequently, difficult to learn and use. In addition, the set of primitives is very large.

Sharing

Processes should be able to share code and data segments. The latter can be provided by mapping files into process address space. For the former, library files should be automatically sharable.

VMS allows sharable code segments, sharable libraries, and permits files to be mapped into user space. UNIX is much more limited allowing only a single sharable code segment per process, no library sharing, and no file mapping into user space.

Protection

Development environments tend to comprise overlapping projects. Consequently, an overlapping protection group structure is needed to model the situation. Users should be able to establish membership in multiple groups. It should also be possible to establish at least one level of privilege hierarchy within a group. Many situations require a supervisory structure within groups.

Both VMS and UNIX provide a strict user/group/global structure with VMS providing one additional more privileged system level. Neither meet our requirements here.

Resource Control

Resource quotas are not essential for most development environments since quotas always represent arbitrary limitations, and administrative controls are usually effective. They are, however, convenient to have and in some situations, such as a mixed teaching and research facility, absolutely essential.

Available documentation indicates that VMS has a resource quota mechanism. UNIX certainly does not. However, at least one VMS installation is known to be operating without any resource control and is of the opinion that VMS cannot provide it.

Summary

We do not intend to state any conclusions with respect to the superiority of VMS or UNIX for a development environment. Any such pronouncement, in addition to being subjective, would not be credible if based on such an informal study. Rather our intent has been to try and look at some

characteristics of the "forests" in question without being drawn into discussions about the "trees".

While not stating conclusions, we do have opinions. We find the use of assembler in VMS very surprising and disappointing. (It has been suggested that this was a marketing decision.) UNIX wins in this regard and also in its I/O and file accessing facilities. Placing RMS in the path of users seems unnecessary. On the other hand VMS seems to have rather more general sharing and inter-process communication. Finally, the VMS system call interface has not been received well by those who have been exposed to it.

Given our choice we would lean to UNIX, primarily because it represents less of an obstacle and more of a friendly tool.

April 1980
(M)

A PDP-11 FRONT-END FOR A VAX-11/780*

M. J. Browne, Charles Granieri, D. J. Sherden, Leon J. Weaver
Stanford Linear Accelerator Center
Stanford, California

US DECUS 1980 SPRING MEETING

ABSTRACT

An unpublicized feature of the VAX-11/780 is the provision for attaching a PDP-11 to the VAX UNIBUS Adapter. This can give significantly improved I/O performance for applications which are limited by overhead in the VAX I/O driver rather than by the transfer speed of the UNIBUS itself. We have implemented such a system using a PDP-11/04 as a "front-end" to a CAMAC data acquisition system. Both the PDP and the VAX have full access to the UNIBUS. That portion of the PDP address space which does not have UNIBUS memory can be mapped to buffers in the VAX memory, allowing the PDP to access VAX memory and to initiate DMA transfers directly to the VAX. The VAX also has full access to the PDP memory, providing a convenient means for developing and downloading the PDP software.

INTRODUCTION

As online computers have progressed from simple to more complicated machines such as the VAX-11/780 many of the more difficult tasks have been made easy, but many of the simple tasks have been made difficult. One such area is in the reading of time-critical data. While modern computers offer useful facilities for user protection, DMA capability, and I/O queueing and buffering, the software overhead in the interrupt servicing and I/O systems has in some cases outpaced the increasing speed of the computers. Fortunately, such problems are well suited to micro-computer applications.

In this paper we describe a means of connecting a PDP-11 computer (in our application a PDP-11/04) to the VAX-11/780, which allows the two to interact in a very flexible fashion. We note that this scheme is an unpublicized design feature of the VAX, and hence credit for the system should go to Digital, although we accept responsibility for any misstatements in this paper, and, should it make anyone feel better, blame for any of its failures.

In our particular application a VAX-11/780 is used for the acquisition and analysis of data for elementary particle physics experiments in End Station A of the Stanford Linear Accelerator Center. Communication with the experimental equipment is accomplished through a CAMAC [1] system using three Jorway Model 411 Branch Drivers [2]. Time critical data are acquired at rates of up to 360 "events" per second. While the amount of data read for each event is small (typically several hundred bytes), the high repetition rate, combined with the fact that each event requires several separate PDT (single word transfer) and DMA (block transfer) operations, introduces significant software overhead.

As the system was originally set up, the event data were read by the VAX using a locally written CAMAC I/O software driver [3]. To minimize software overhead this system provides for list-driven multiple CAMAC operations and multiple interrupt servicing within a single QIO system service call (note that the software overhead of a single QIO call is comparable to the time between events). Independent of the overhead involved in the QIO call itself (~2 msec), each CAMAC PDT operation requires ~30 usec, each DMA operation requires ~300 usec, and each event interrupt requires ~300 usec. With this system, the reading of event data required ~25% of the total CPU power of the computer. Particularly when compared to the time available for the analysis of events, this is a quite significant overhead, which, with the present scheme, is almost entirely eliminated through the use of a "front-end" PDP-11/04 to read the data.

THE SECRET PORT

In the most naive picture, one imagines the UNIBUS adapter (UBA) hanging from the SBI with the UNIBUS emanating from the UBA, as shown in Figure 1. The UNIBUS arbitration functions are mentally associated with the UBA. There are two separate UNIBUS arbitration functions: Non-Processor Request (NPR) arbitration and Bus Request

*Work supported by the Department of Energy, contract DE-AC03-76SF00515.

Presented at the 1980 Spring DECUS U. S. Symposium, Chicago, Illinois, April 22 - 25, 1980.

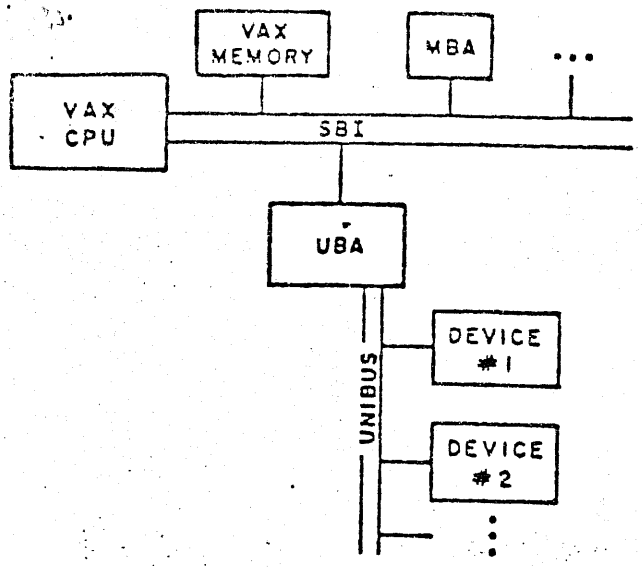


FIGURE 1

A naive view of the UBA.

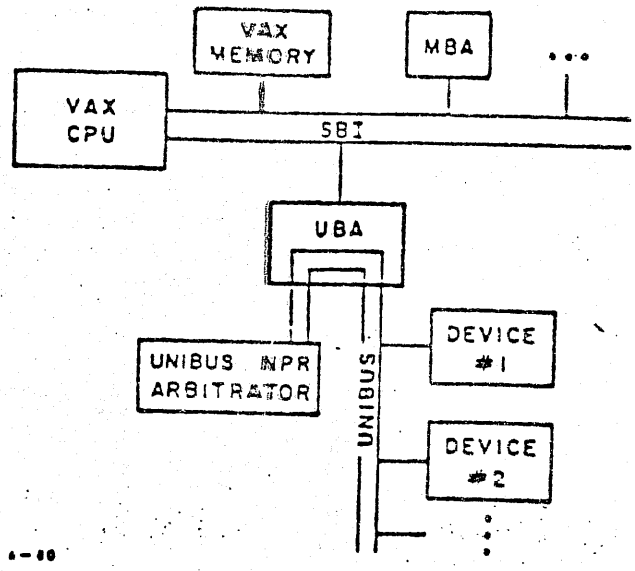


FIGURE 2

A slightly less naive view showing the separation of UBA and NPR arbitrator.

(BR) interrupt processing. While the UBA does contain a BR interrupt processor, the NPR arbitrator is separate from the UBA, as shown in Figure 2. The UBA may, for most purposes, be considered as an NPR device on the UNIBUS. The NPR arbitrator is functionally identical to that in any PDP-11, so one can simply replace the arbitrator card with a standard UNIBUS cable to a PDP-11 without adverse effect, as shown in Figure 3.

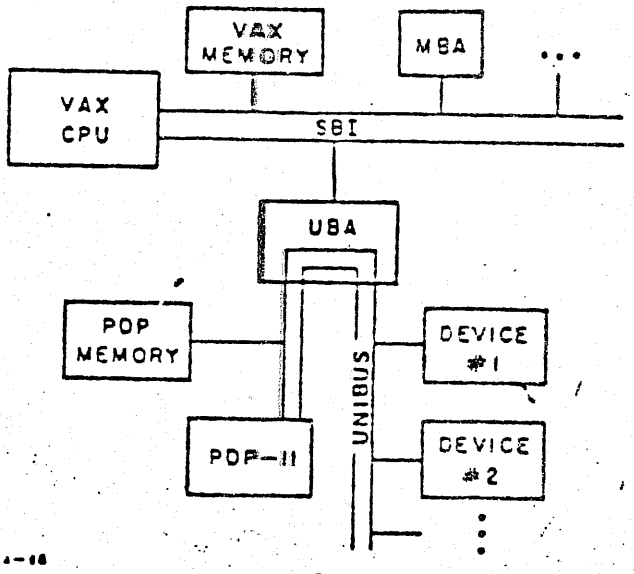


FIGURE 3

UBA with NPR arbitrator replaced by PDP-11

THE UNIBUS ADAPTER

Almost all of the information in this section is readily available from the VAX-11/780 Hardware Handbook. We nonetheless present the information here since it is relevant to understanding the VAX/PDP interaction.

One of the most important functions of the UBA is to map the UNIBUS address space 000000-757777(8) to VAX memory. For this purpose, the UBA contains 496 map registers, allowing one to map each page of UNIBUS address space to VAX memory. Because the PDP has its own memory on the UNIBUS, one needs a mechanism for disabling the corresponding map registers in the UBA. For this purpose (or to accommodate external UNIBUS memories, in general) DEC provides the Map Register Disable field (bits 26:30) of the UNIBUS Adapter Control Register (UACR). This field may be loaded with the number of 4K word blocks of external UNIBUS memory, which must begin at UNIBUS address 0. With the corresponding map registers disabled, the VAX has complete access to the UNIBUS memory; the disabling simply prevents the UBA from attempting to associate these pages with VAX (SBI) memory. The fact that the VAX can access the PDP memory directly provides, among other things, an extremely simple method of downloading programs to the PDP.

The PDP can be given access to VAX memory using the UBA map registers. Here one simply maps some portion of PDP address space without UNIBUS memory to the desired pages of VAX memory. Note that for PDP models without memory management, the sum of PDP memory plus PDP-accessable VAX memory is thus limited to 28K words. The UNIBUS address space allocation for our configuration (PDP-11/04 with 8K words of memory) is illustrated in Figure 4.

As long as the map registers remain unchanged, the PDP may treat the associated VAX memory as though it were its own. Since the PDP also has access to the UNIBUS I/O space (760000-777777(8)), it may initiate I/O from a UNIBUS device directly to or from VAX memory.

Because the UBA has its own BR interrupt processor, it intercepts BR interrupts from all UNIBUS devices downstream of the UBA

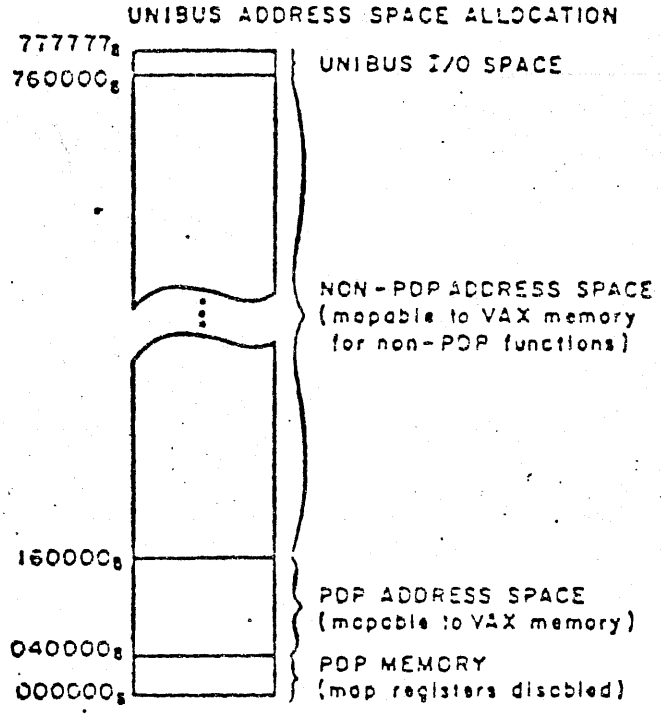


FIGURE 4

(the "VAX side" of the UNIBUS) and interrupts the VAX, not the PDP. BR interrupts from devices upstream of the UBA (the "PDP side" of the UNIBUS) are received by the PDP and not the VAX. This arrangement can be altered by clearing (or, more precisely, by not setting) the Interrupt Field Switch (Bit 5) of the UACR. In this case the UBA will pass all BR interrupts to the PDP.

HARDWARE

Hardware associated with the PDP-11/04 in our system is shown schematically in Figure 5. The PDP is powered up and down with the VAX CPU. When the system is powered up, PDP control is transferred to the ROM of the M9301-YA bootstrap module. Since the UBA is initially unmapped, it is important to prevent the PDP from accessing its own memory until the UACR Map Register Disable Field has been properly initialized. (Note that the M9301 ROM is in the I/O space of the UNIBUS, which is always known to be external by the UBA.) Hence it was desireable to provide a means of communication between the VAX and the PDP which did not require the use of PDP memory. For this purpose a simple interface module was build using an MDB-1710 foundation module [4] plugged into the PDP side of the UNIBUS so that the PDP rather than the VAX receives its interrupts. The interface module consists of a control register (CSR), two data registers, and two interrupts. One register is used by the VAX to initiate program operation, while the second is left free for user applications. The first interrupt is used to signal an event; while the second is used by the VAX to terminate PDP program operation. (As designed, either interrupt can be fired from hardware or from

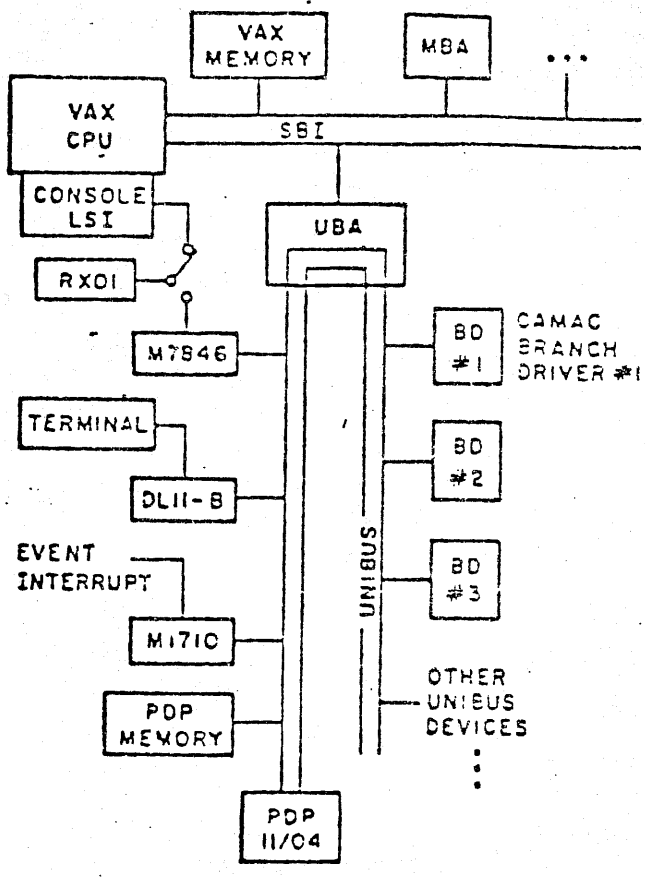


FIGURE 5
PDP-associated hardware for our system.

software.) To provide for PDP program downloading a simple modification was made to the M9301 ROM program. The program initially loads an odd number into the data register of the interface module. In addition to looking for input from the DL-11B (i.e. the normal console emulator routine) the program also looks at the data register. After the VAX has downloaded the program into the PDP memory, it places the (even) starting address in the data register. This is recognized by the PDP and used to begin program execution. Sufficient unused space exists in the M9301-YA that the downloading feature could be added without elimination of any features of the standard ROM.

In the standard ROM console emulator routine, a START command causes the ROM program to execute a RESET before transfering control to the requested address. Since this instruction resets all devices on the UNIBUS, its execution after the VAX has been bootstrapped would have disastrous consequences. Hence the RESET instruction was eliminated from the ROM program.

In addition to the VAX/PDP interface, the PDP also has a DL-11B serial line interface and an M7846 floppy disk controller, both of which are used solely for diagnostic purposes. The DL-11B can be connected to one of the terminals normally used by the VAX, and the M7846 can be plugged into the

CRYC normally used by the VAX console LSI. As an indication of the reliability of the system we note that the terminal has not been connected to the PDP since the initial debugging of the PDP program, and, apart from verifying that the M7846 module worked, the floppy disk has never been connected to the PDP.

The only unexpected problem which arose in bringing up the system occurred in the bootstrap sequence. With the switches of the M9301 bootstrap module properly set, the power up sequence should cause the PDP to interrupt to the M9301 ROM. This indeed occurred when power was initially applied to the CPUs. However, the power fail and power up sequence can also be generated from VAX software by: (i) SBI UNJAM, (ii) setting the Adapter Init field (Bit 0) in the UACR, or (iii) setting (and resetting) the UNIBUS Power Fail field (Bit 1) in the UACR. At least one of these methods is employed by the VAX bootstrap procedure. It was found that for the software generated power up sequence, the AC LO signal was deasserted simultaneously with DC LO rather than the prescribed >5 usec later. This caused the PDP to trap to location 24 (power fail) rather than to the bootstrap ROM. While a less brutal approach is probably possible, we cured the problem by cutting a trace on the M9301 module.

SOFTWARE

A VAX/VMS I/O driver was written to support the "front-end" PDP-11/04. While not technically necessary, the I/O driver format was chosen because it offered a convenient and well documented means of accessing both VMS system routines and UNIBUS addresses. There is, however, one special feature about this driver. In order to allocate the UBA map registers specifically associated with the PDP, the PDP driver must be loaded before the drivers of any other devices which access the UNIBUS.

When it is loaded, the PDP driver initialization routine performs the following functions:

1. It permanently allocates within VMS the first 28K words of UNIBUS addresses (i.e. the PDP address space).

2. It sets the Map Register Disable field of the UACR for the 8K words of PDP UNIBUS memory which is attached.

3. It sets up VMS system page table entries so that the 8K of UNIBUS memory may be read and written directly from the VAX and saves the generated virtual addresses for later use.

VAX user programs may access the PDP through the PDP driver by using the standard QIO system service. The PDP driver supports the following functions:

1. Write to PDP memory.

2. Read from PDP memory.

3. Read VAX/PDP interface registers.

4. Write VAX/PDP interface registers.

5. Interrupt the PDP by writing into the VAX/PDP interface CSR register.

6. Set up a "never-ending" QIO which maps UNIBUS addresses between 8K and 28K into VAX memory to allow the PDP program to read data directly into or from the VAX memory.

Down-loading of programs from the VAX to the PDP is accomplished by a user level routine using the QIO facility described above. PDP programs are prepared, assembled, and linked using the compatibility-mode RSX-11M facilities of the VAX. The down-loading routine reads the load module to determine the program length, first address, and starting address, as well as the program itself. Using the QIO facility, the program is written to PDP memory, and the starting address is written to the VAX/PDP interface module data register to initiate program operation.

OPERATIONAL EXPERIENCE

In our particular application one of the three CAMAC branch drivers is dedicated exclusively to the reading of event data. This restriction was present in the original VAX-based system and has been carried over to the PDP system. The other two branch drivers are used for I/O which occurs at repetition rates significantly lower than that of the event data, and remain driven by the VAX rather than the PDP. The isolation of the event branch was adopted to avoid handshaking and interlocking problems between the VAX and the PDP.

The event interrupt was moved from the VAX to the PDP. The event branch driver, however, was left on the VAX side of the UNIBUS so that diagnostic programs can be run from the VAX when the PDP is not running its normal event reading program. This prohibits the PDP from receiving DMA completion interrupts from the branch driver. However, since CPU time on the PDP is not at a premium, it is a simple matter for the PDP to monitor the CSR of the branch driver until the DMA operation is completed.

Upon receipt of an event interrupt, the PDP reads event data along with status information from the branch driver directly into a circular buffer in VAX memory. The buffer is large enough to contain roughly 20 events. A VAX program, which is activated roughly 10 times per second, retrieves data from the buffer for analysis and logging on magnetic tape.

Because the PDP could be programmed to read data in a more brute force fashion than the list-driven VAX I/O system, the PDP was able to duplicate the functions of the VAX-based system using less real time. In practice,

the PDP program was expanded to provide error checking with retry capability, and to format the data in a form more convenient to the specific experiment than that of the more general VAX-based system. The final PDP program reads events in approximately the same time as did the original VAX system. While most of the PDP program is specific to the current experiment, it was written with sufficient generality that reprogramming for an auxiliary experiment was accomplished in less than a day.

The PDP system has been in use for six months, and has not encountered problems in that time.

CAVEATS

While we have been extremely satisfied with the system, we must add that it is not all things to all people.

1. The system is effective in eliminating software overhead but does not improve hardware performance. In particular, since the PDP requires UNIBUS cycles to access its own memory, UNIBUS performance will be degraded rather than enhanced. Note, however, that the PDP itself has the lowest UNIBUS priority and the WAIT instruction can be used to inhibit PDP activity during idle periods.

2. For PDP models (including the PDP-11/04) which use the DATIP-DATO (read-modify-write) sequence to write to memory, the UBA direct data path must be used, resulting in heavier traffic on the SBI. Even for models using the DATO sequence, the use of a UBA buffered data path would destroy the feature of 16 bit random access to VAX memory by the PDP, although one could have 32 or 64 bit access. Similarly, for any model, the PDP could initiate DMA transfers from I/O devices to the VAX through a buffered data path if the block size were always in integral units of 32 or 64 bits, or if the VAX were interrupted to purge the data path after DMA completion.

3. VAX buffers which are to be used by the PDP must be page aligned in VAX memory since the UBA map registers can only map a page of VAX memory to a page of UNIBUS address space.

4. A HALT instruction on some PDP models (including the PDP-11/04) hangs the UNIBUS causing the VAX to crash. Don't do that.

5. Access by the PDP to unmapped (and not disabled) UNIBUS addresses also causes the VAX to crash. Don't do that either.

6. One of the VAX micro-diagnostics gives an error condition when the PDP is connected. We have not investigated this further, but simply disconnect the PDP before running the micro-diagnostics.

7. Crash recovery is not an important aspect of our system and we have not paid detailed attention to the relative power up and down sequences of the VAX CPU, PDP CPU, and UNIBUS adapter.

8. As previously mentioned, no general scheme exists for the sharing of interrupts.

9. While it is in principle possible to share a common I/O device between the VAX and the PDP, provision must obviously be made in hardware and/or software to handle the associated handshaking and lock-out problems.

SUMMARY

The system described provides a simple means of interfacing a PDP-11 computer to a VAX-11/780, and offers the following features:

1. Complete access to the UNIBUS by both PDP and VAX computers.

2. Complete access to PDP memory by the VAX.

3. Limited access to VAX memory by the PDP.

4. Initiation by the PDP of I/O directly from UNIBUS devices to VAX memory.

5. Availability of RSX-11M facilities on the VAX, providing a convenient means of PDP program development.

ACKNOWLEDGEMENTS

We would like to thank Mr. Rick Casabona of the DEC VAX-11/780 Engineering Group, f his advice in this project.

This work was supported by the Depart of Energy under contract nu DE-AC03-76SF00515.

REFERENCES

[1] Computer Automated Monitorin Control (CAMAC), IEEE Standard 583-1

[2] Jorway Corporation, Westbury, N

[3] Charles Granieri and Karl REAL-TIME DATA COLLECTION USIN VAX/VMS SYSTEMS, Proceedings of Equipment Users Society, Vol. April 1979.

[4] MDB Systems Inc., Orange,

Macros for Analyzing C Program Arguments

J. Lions

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Two sets of macros are described for use in the analysis of the arguments to C programs. The advantage of their use should be a considerable reduction of programmer effort and improved comprehensibility of programs.

1. Introduction

One of the areas of program implementation in UNIX† systems that does not seem to have progressed very far in recent times is the analysis of the arguments passed to a program by its precursor (usually the *shell*). This paper describes two sets of macros, each of which provides a mechanism for defining and setting internal flags and variables based on the arguments passed to the program. The aims of the present proposal are:

1. above all, simplicity and ease of use;
2. sufficient range and flexibility to handle the requirements of most existing UNIX commands.

A survey of existing UNIX commands reveals many diverse styles of syntax and semantics for command arguments. However, amongst all these, there are a number, including many of the most important and heavily used commands, that conform to a fairly standard approach. The essential features of this approach are:

1. The division of arguments into "key" and "non-key" arguments, with the latter most frequently being the names of files.
2. The use of '−' or '+' to introduce "key" arguments.
3. The use of single character "keys", possibly followed by a number, another character, a string, or, occasionally, some combination of these.
4. The independence of the various keys, that may appear in arbitrary combinations and orders, and may be divided in various ways among one or more key arguments.

There is a relatively a small number of existing commands whose current argument usage is sufficiently exotic that they will most probably never be included in a general argument analysis scheme, and that certainly fall outside the scheme just described. This comment applies particularly to commands that are sensitive to the ordering of several, syntactically similar arguments, e.g. *sort,* or commands with a strong tradition such as *dd* and *find,* that use multi-character keywords.

The existence of a set of macros of the type described here does not of itself impose a standard, but it should discourage the creation of non-conforming commands in the future. Of the existing commands, those that are clearly non-conforming are likely to remain so; those that are already conforming may still gain in clarity and ease of maintenance by being rewritten to take advantage of the macros.

At the time the initial version of these macros was being written, it was brought to the writer's attention that there existed a proposed standard for standardizing the format of UNIX commands. (See *UNIX Command Syntax,* by A. S. Cohen, S.B. Olsson and G. C. Vogel, internal BTL memorandum.) The proposed standard is intended to impose a much stricter view on the

† UNIX is a trademark of Bell Laboratories.

way arguments to commands may be presented than the *laissez faire* approach of the past and proposes a drastic simplification of practice and the elimination of many present idiosyncrasies. It has the merits of simplicity and rigidity and holds the promise of simplifying the lives of less experienced users. It is also fair to say that it is not without its controversial aspects.

On the other hand, the proposed standard is not entirely compatible with current practice, so that the macros that implement it differ in some respects from the set which was originally designed. Accordingly, as a way of accommodating a diversity of opinions, two sets of macros have been formulated. These two are largely compatible in appearance and usage. The second set, which is found in the file *nstdargs.h* in Appendix B, is considerably larger and has considerably more options (reflecting current practice) than the first set. The latter, which implements the proposed standard command syntax, is found in the file *stdargs.h* in Appendix A. Since *stdargs.h* is the simpler, it is the first to be described in detail below.

2. The Flavour

The intent of the present proposal is that the programmer should be able to replace the section of his or her program that analyzes *argc* and *argv* with something like

```
.
.
ARGBEGIN('-')
        FLAG   ('f', fflag);
        FLAG   ('t', tflag);
        NUMBER('w', wval);
        STRING('h', hptr);
ARGEND;
.
.
```

The function of the above set of instructions is to analyze the argument list, looking for arguments that begin with '−'. Each of these is analyzed character by character:

1. if an 'f' is encountered, the variable *fflag* is incremented.
2. if a 't' is encountered, the variable *tflag* is incremented.
3. if a 'w' is encountered, the variable *wval* is given the numeric value of the unsigned numeric string immediately following (if the string is null, a fatal diagnostic is generated).
4. if an 'h' is encountered, the pointer *hptr* is set to point to the remainder of the current argument, or, if this is null, to the next argument in the argument list (and this would not be further examined during this phase of argument analysis).

An important feature of the present proposal is that after the ARGEND statement, any arguments that were not processed (i.e., in the above example, that did not begin with '−', and were not "captured" by an 'h') are available via a compacted list in the usual *argc/argv* format.

Under the proposed standard, and with the use of the present proposal, if the program *xyz* incorporates the above code, it would find the following lists identical and acceptable:

```
−t −f −w10 −h junk able elba
−t −w10 −h junk able −f elba
−hjunk able −tf elba −w 10
```

and it would increment each of *fflag* and *tflag* once, set *wval* to 10, would set *hptr* to point to "junk", would leave *argc* with a value of 3, and would leave *argv* pointing to the strings "xyz", "able", "elba", 0.

3. The Proposed Standard

Because the proposed standard has not yet been widely promulgated, it will be described here first before the macros that implement it are introduced. A BNF description of the proposed standard format of commands (other than those recognized directly by the shell) is as follows:

command	= commandname options names
commandname	= {name of an executable file}
options	= {null} | options {white space} option
names	= {null} | names {white space} name
option	= flagoption | stringoption
flagoption	= '−' flagkey | flagoption flagkey
stringoption	= '−' stringkey separator string
flagkey, stringkey	= {letter}
separator	= {null} | {white space}
name	= '−' | {string not beginning with '−'}
string	= {character string containing no unquoted white space}

Under this proposed standard the following forms of the *ptx* command would be acceptable and equivalent:

```
ptx −ft −w 70 infile outfile
ptx −f −t −w 70 infile outfile
ptx −ft −w70 infile outfile
ptx −f −w70 −t infile outfile
ptx −f −w 70 −t infile outfile
```

while the following are not acceptable:

```
ptx −ftw 70 infile outfile
ptx −fw70t infile outfile
ptx −f −w −t infile outfile
```

From the point of view of implementation, the most controversial feature of the proposed standard is the optional white space separator within a string option, because such white space is normally "processed" by the shell before the command is invoked.

4. The Standard Macros

In the descriptions that are given below, c, y, and p stand for a character value, an integer variable, and a character pointer, respectively. The two macros, ARGBEGIN and ARGEND, form the head and tail respectively of a *switch* statement, within which FLAG, STRING and NUMBER represent individual cases.

MAIN	Has the value "main (argc, argv) char *argv[];"
USAGE	Declares a "usage" message to be displayed before termination if argument errors occur. Must be used between MAIN and ARGBEGIN to be effective.
ARGBEGIN(c)	Sets up code to search the argument list, looking for arguments whose first character is *c*. Set up code to search the second and remaining characters of such arguments via a switch statement, keyed on individual characters.
ARGEND	Terminates the switch statement opened by ARGBEGIN and terminates processing if errors have occurred.

FLAG(c,y)	If the option key *c* occurs, increments *y*.
STRING(c,p)	If the option key *c* occurs, sets *p* to point to the string of characters remaining in the current argument, or if this is null, to the next entire argument in the list. In the latter case, the argument will be removed from the list and not scanned.
NUMBER(c,y)	If the option key *c* occurs, takes the string that follows (located according to the same rules as for STRING described above) and evaluates it as a decimal number. Reports an error if the string contains any non-numeric characters.

The following macros, for ARGINIT, ARGCOUNT, NEXTARG and ARG, together with the one for MAIN already given, are somewhat controversial, depending on whether one believes that the variables *argc* and *argv* should, or should not, be explicitly recognized by the application programmer.

ARGINIT	Initializes variables for NEXTARG when ARGBEGIN and ARGEND are not used,
ARGCOUNT(c1,c2)	Checks the number of arguments, usually after the options have been analyzed. If the number of arguments in the *argv* list is less than *c1*, or greater than *c2* (given that *c2* > *c1*), then the usage message is displayed and the program terminates. Note that *argv[0]*, the command name, counts as one argument here.
NEXTARG	Gives a pointer to the next argument from among those arguments which remain in argv[] after earlier processing.
ARG	Denotes the value previously returned by NEXTARG (initially *argv[0]*).

The relevant portions of a version of the command *ptx* that would conform to the proposed standard are contained in the following short C program:

```
#include <stdargs.h>
int tflag, fflag, wnum;
MAIN {
        USAGE "-ft -w[ ](n) [ infile [ outfile ]]";
        ARGBEGIN('-')
                FLAG   ('f', fflag);
                FLAG   ('t', tflag);
                NUMBER('w', wnum);
        ARGEND;
        ARGCOUNT(1, 3);
        fin  = fopen (NEXTARG, "r");
        fout = fopen (NEXTARG, "w");
}
```

The result after this program was passed through the C preprocessor (to expand the macros) and *cb* (to clean up the result), is given in Appendix C. The following features may be noted:

1. *stdargs.h* will include *stdio.h* and *ctype.h* if these have not already been included.
2. A semicolon may not appear immediately after ARGBEGIN, but is optional after ARGEND.
3. The method for argument analysis which is used does not enforce a clear separation between "options" and "names", i.e., the two can be inter-mixed.

5. The Regular Macros

The file *nstdargs.h* includes all the macros mentioned in *stdargs.h* plus several additional ones. One small difference exists between the two sets. With the regular macros, an argument consisting of a single '−' will not necessarily be classed as a "name" but may be treated as an "option" (see OCCURS and STANDIN below).

As before, in the descriptions that follow, c, y, p, and a stand for a character value, an integer variable, a character pointer and an array of integers respectively.

ARGELSE(c)	Formally equivalent to ARGEND; ARGBEGIN(c)
SUFFIX(c,p)	If the option key *c* occurs, sets *p* to point to the remainder of the current argument and does not scan the current argument further.
NAME(c,p)	If the option key *c* occurs, sets *p* to point to the next entire argument in the list. This argument will not be scanned subsequently.
NUM(c,y)	If the option key *c* occurs, evaluates the numeric string that follows immediately in the current argument, and assigns its value to *y*. Continues analysis at the next non-numeric character of the current argument. A null string has the value −1.
NUMLIST(c,a)	If the option key *c* occurs, takes the remainder of the current argument as a list of unsigned integers separated by strings of non-numeric characters. The numeric strings are converted to integers and stored in successive elements in the integer array denoted by *a*. The list entry beyond the last valid entry is set to −1.
CHARNUM(c,d,y)	If the option key *c* occurs, looks for a single, optional non-numeric character, followed by an optional unsigned numeric string. The optional character is assigned to d, and the value of the string to *y*.
VALUE(y)	If the current argument contains an "unattached" numeric string, assign its value to *y*.
OCCURS(y)	If the current argument consists only of the single lead character, e.g. '−', increments the value of *y*.
STANDIN	If the current argument consists only of the single lead character, e.g. '−', treats it not as an option but as a name (i.e., one which is left in the *argv* list).

Note that OCCURS and STANDIN are mutually exclusive, and that the latter is assumed implicitly in "standard" macros; also that NUM is a restricted version of NUMBER that will not reject null digit strings, and that the territory covered by STRING is divided between SUFFIX and NAME.

6. Example

The following is a larger example that illustrates the coding needed to handle the argument lists for the *pr* command:

```
/* sample for pr command */
#include <stdargs.h>
#include <stdio.h>
      .
      .
int firstp, ncols, aflag, mflag, ...
int xval = -1;
char *eptr, *hptr, *nptr, *sptr;
      .
```

```
MAIN {
        .
        .
        USAGE "+k −k −adm −eCK −nCK −wK −oK −lK −h N −pfrt −sC" ;
        ARGBEGIN ('+')
            VALUE (firstp);       /* first page      */
        ARGELSE('−')
            VALUE (ncols);        /* k column output*/
            FLAG  ('a', aflag);   /* print across page*/
            FLAG  ('m', mflag);   /* merge  */
            FLAG  ('d', dflag);   /* double spacing*/
            SUFFIX('e', eptr);    /* tab settings   */
            SUFFIX('n', nptr);    /* line numbering*/
            NUM   ('x', xval);    /* old line numbers*/
            NUM   ('w', wval);    /* page width     */
            NUM   ('o', oval);    /* offset   */
            NUM   ('l', lval);    /* page length    */
            NAME  ('h', hptr);    /* header   */
            FLAG  ('p', pflag);   /* pause for each page*/
            FLAG  ('f', fflag);   /* use form-feeds*/
            FLAG  ('r', rflag);   /* no diagnostic reports*/
            FLAG  ('t', tflag);   /* no header or trailer*/
            SUFFIX('s', sptr);    /* column separator*/
        ARGEND;
        .
        .
        /* analyze sptr, nptr, eptr */
        .
        .
        /* process files */
        if (argc > 1)
        while (fopen (NEXTARG, "r") != NULL) {
            .
        }
}
```

Note that variables, such as

firstp, ncols, aflag, ..., hptr,...

in the example, must be predeclared, preferably as external variables. For most of these, the default initialization to zero will be adequate, but for some, explicit initialization may be needed.

Because the standard macros do not include processing of the peculiar kind required by the *n* and *e* options, the suffixes to these characters have been separated for later processing. In the case of the 's' option, it is possible to replace the line relating to *s* by something like

```
case 's': sval = *(++p); continue
```

This assumes a variable *sval,* rather than a pointer *sptr,* and some knowledge of the internal workings of the macros. The pointer *p* is a local variable in the block initiated by ARGBEGIN.

7. A Comment

It has been suggested that the present proposal, by disturbing the initial *argv[]* structure, will interfere with the "normal" expectations of the command *ps.* However there is no long-established tradition that commands do not alter argv[], and in fact, some commands (e.g. *passwd, crypt*) modify their arguments as quickly as possible for security reasons. Moreover, under the present proposal, while the dope vector that points at argument strings may be

modified, the value of *argv[0]* and the argument strings themselves are not modified, so that the problem for *ps* does not become impossible.

8. Action and Reaction

The circulation of early drafts of this memorandum produced two reactions. The first was the suggestion of many more features that could be supported by suitable expansions of the macro package. (The formulation of these presented a challenge which could be accepted with pleasure.) The second, in many ways a delayed reaction to the first, was to decry the proliferation of macros and to argue for severe economy. Of late, the author has inclined towards the second view. The macros in the file "stdargs.h" are available through the UNIX stockroom. The others have been invented and hence exist. They can be reinvented and reinstalled when, and if, popular demand dictates.

9. Acknowledgments

The assistance of Aaron Cohen, Ted Dolotta, Dick Haight, Andy Hall, Andy Koenig, Doug McIlroy, Bert Olsen, Larry Rosler and Jerry Vogel in reviewing and commenting on earlier drafts of this memorandum is gratefully acknowledged.

Appendix A. stdargs.h

```
#ifndef FILE
#include <stdio.h>
#endif
#ifndef isdigit
#include <ctype.h>
#endif

int     A_errcnt = 0;
int     A_index = 0;
int     A_done = 0;
char    *A_usage;
char    *A_rgv0;

int A_atoi(d) char **d; { /* convert string to number */
        register int i;
        register char *p;
        p = *d;
        if (p==NULL || *p=='\0') i = -1;
        else {
                while (*p==' ') p++;
                i = ((isdigit(*p)) ? atoi(p) : -1 );
                while (isdigit(*p)) p++;
        }
        *d = --p;
        return (i);
}

int A_message (n, c) register int n; register char c; {
        static char cc;
        static  char *A_mm [] = {
                "usage: %s\n",
                "Warning: use of option %c is non-standard\n",
                "Option %c not recognized\n",
                "No string for option %c\n",
                "Bad number for option %c\n",
                "Too few arguments\n",
                "Too many arguments\n",
                0
        };
again:  fprintf (stderr, "%s: ", A_rgv0);
        if (n==0) {
                if (A_usage)
                        fprintf (stderr, A_mm[0], A_usage);
                exit (1);
        }
        if (c != cc) {
                fprintf (stderr, A_mm[n], c);
                if (n>4) { n=0; goto again; }
                if (n>1) { cc = c; A_errcnt++; }
        }
        return (n);
}
```

```
#define A_MESS(n)  A_message(n, *p)

#define A_MESS1(n)  A_message(n, 1)

#define A_BEGSW(k) if(**q==k){ for(p=(*q)+1; ; p++){ switch(*p){

#define A_ENDSW default: if (*p) A_MESS(2); }/*end switch *p*/\
        if(*p=='\0'||A_done) break; }/*end for p*/ }/*end if **q*/

#define A_SUFF ( A_done++, (p+1) )

#define A_NEXT ( q[1] ? *(++q) : (A_MESS(3), NULL) )

#define A_STR ((*q!=(p-1))&&A_MESS(1),((*p&&(p[1]=='\0'))?A_NEXT:A_SUFF))

#define STANDIN case '\0': if (*q==(p-1)) goto A_SAVE

#define ARGBEGIN(k) {char **q, **qq, *p, *pp; A_rgv0=argv[0]; argc=1; \
        for(q=qq= &argv[1];*q;q++){ A_done = 0; A_BEGSW(k)

#define ARGEND STANDIN; A_ENDSW else A_SAVE: {*(qq++)= *q;  argc++; } \
        }/*end for q*/ if(A_errcnt)A_MESS1(0);*qq=0;ARGINIT;}/*end block*/

#define ARGINIT A_index = 0;

#define ARGCOUNT(ca,cb) {if(argc<ca)A_MESS1(5);if((cb>ca)&&(argc>cb))A_MESS1(6);}

#define ARG argv [A_index]

#define NEXTARG argv [ ARG ? ++A_index : A_index ]

#define ARGDISPLAY {printf("(%d) ",argc);  do printf("%s ",ARG); \
        while(NEXTARG);  putchar('\n'); ARGINIT;}

#define FLAG(a, aflag) case a: aflag++; break

#define NUMBER(c,cnum) case c: pp=A_STR; cnum=A_atoi(&pp); \
        if (pp[1] || cnum==-1) A_MESS(4); break

#define STRING(s, sptr) case s: sptr = A_STR; break

#define USAGE A_usage =

#define MAIN main (argc, argv) char *argv []; int argc;
```

Appendix B. nstdargs.h

```
#include "stdargs.h"

#undef ARGEND
#define ARGEND A_ENDSW else A_SAVE: {*(qq++)= *q;  argc++;} \
        }/*end for q*/ if(A_errcnt)A_MESS1(0);*qq=0;ARGINIT;}/*end block*/

#define ARGELSE(c) A_ENDSW else A_BEGSW(c)

#define SUFFIX(c, ptr) case c: ptr = A_SUFF; break

#define NAME(c, ptr) case c: ptr = A_NEXT; break

#define NUM(c, cnum) case c: cnum = A_atoi(&p); break

#define NUMLIST(c,a) case c: do { while(*p && !isdigit(*p)) p++; \
        if (*p=='\0') break; a[A_done++]=A_atoi(&p); \
        } while (*(++p)); a[A_done++] = -1; break

#define CHARNUM(c,d,y) case c: d = (isdigit(*(++p)) ? '\0' : *p++); \
        y = A_atoi(&p); break

#define VALUE(y) case '0' : case '1' : case '2' : case '3' : \
        case '4' : case '5' : case '6' : case '7' : case '8' : case '9' : \
        y = A_atoi(&p); break

#define OCCURS(y) case '\0': if (*q==(p-1)) y++; break
```

Appendix C. Sample Program

```
extern    struct    _iobuf {
        char *_ptr;
        int   _cnt;
        char *_base;
        char _flag;
        char _file;
}
_iob[20];
struct _iobuf    *fopen();
struct _iobuf    *freopen();
struct _iobuf    *fdopen();
long ftell();
char *fgets();
extern    char _ctype_[];
int   A_errcnt = 0;
int   A_index = 0;
int   A_done = 0;
char *A_usage;
char *A_rgv0;
char *A_mm [] = {
        "usage: %s\n",
        "Warning: use of option %c is non-standard\n",
        "Option %c not recognized\n",
        "No string for option %c\n",
        "Bad number for option %c\n",
        "Too few arguments\n",
        "Too many arguments\n",
        0
};
int A_message (n, c) int n;
char c;
{
        static char cc;
        if (c != cc) {
                cc = c;
again:
                fprintf ((&_iob[2]), "%s: ", A_rgv0);
                if (n==0) {
                        fprintf ((&_iob[2]), A_mm[0], A_usage);
                        exit (1);
                }
                else {
                        fprintf ((&_iob[2]), A_mm[n], c);
                        if (n>1) A_errcnt++;
                        if (n>4) {
                                n=0;
                                goto again;
                        }
                }
        }
        return (0);
}
struct _iobuf *fin, *fout;
int tflag, fflag, wnum;
main (argc, argv) char *argv [];
```

```
int argc;
{
    A_usage = "-ft -w[ ][n] [ infile [ outfile ]]";
    {
        char **q, **qq, *p, *pp;
        A_rgv0=argv[0];
        argc=1;
        for(q=qq= &argv[1];*q;q++){
            A_done = 0;
            if(**q=='-'){
                for(p=(*q)+1; ; p++){
                    switch(*p){
                    case 'f':
                        fflag++;
                        break;
                    case 't':
                        tflag++;
                        break;
                    case 'w':
                        wnum=atoi(pp=((*q!=(p-1))&&A_message(1,*p),
                            ((*p&&(p[1]==' '))?
                            (q[1]?*(++q):(A_message(3,*p),0)):
                            ( A_done++, (p+1) ))));
                        while(    ((_ctype_+1)[*pp]&04))pp++;
                        if (*pp)    A_message(4, *p);
                        break;
                    case ' ':
                        if (*q==(p-1)) goto A_SAVE;
                    default:
                        if (*p)    A_message(2, *p);
                    }
                    if(*p==' ' || A_done) break;
                }
            }
            else A_SAVE:
            {
                *(qq++)= *q;
                argc++;
            }
        }
        if (A_errcnt) {
            A_message(0, 1);
            exit(1);
        };
        *qq=0;
        A_index = 0;
        ;
    }
    ;
    {
        if(argc<1)  A_message(5, 1);
        if(( 3>1)&&(argc> 3))  A_message(6, 1);
    };
    fin  = fopen (argv [ argv [A_index] ? ++A_index : A_index ], "r");
    fout = fopen (argv [ argv [A_index] ? ++A_index : A_index ], "w");
}
```

THE UNIVERSITY OF NEW SOUTH WALES

P.O. BOX 1 • KENSINGTON • NEW SOUTH WALES • AUSTRALIA • 2033
TELEX AA26054 • TELEGRAPH: UNITECH, SYDNEY • TELEPHONE 663 0351
EXTN. 3781

PLEASE QUOTE

May 19, 1980

SCHOOL OF ELECTRICAL ENGINEERING

David R Woodrow,
St Peters Lutheran College,
Harts Rd,
Indooroopilly,
Queensland 4068.

Dear David,

It has come to my attention, via a long and devious route that you have written to Western Electric requesting a distribution of UNIX, on RK05s for a PDP11/34. Chris Maltby, from the Basser Dept. of Computer Science, also points out that you do not appear to have supplied all the information necessary for this to be accomplished quickly and with a minimum of paperwork.

You may not know this, but there is a large and growing UNIX User Group in Australia and it is as editor of the newsletter for this group that I am writing to you with some information that may be of help.

Normally requests for UNIX licenses are directed to:

Irma B. Biren,
Supervisor,
MH Computing Information Library,
600 Mountain Ave.,
Murray Hill,
N.J. 07974
U.S.A.

Ms. Biren requires the following information in order to start the ball rolling at her end.

- The class of license required:

  i. Commercial Use

  ii. Educational Institution Administrative Use

  iii. Educational and Academic Use

If you intended to purchase either of i) or ii) above you should write to:

Mr A. L. Arms
Patent Licensing Manager
Western Electric Co. Inc.
P.O. Box 20046
Greensboro, North Carolina 27420
U.S.A.

Should you require a license of the third type, as you apparently do, then you should supply:

- The address of your institutions administrative offices
- A description of the intended use of the software
- The specific address of the data center at which you plan to use the software (including street address and room number)
- The type and serial number of the central processing unit on which the software is to be implemented

Should Ms Biren agree that you qualify for an Educational and Academic Use license, you will, in due course, be forwarded a contract for your approval and execution. Payment of any fees requested, covering duplication and documentation costs, should be made by cheque, made out to Bell Laboratories, payable on a U.S. bank. Purchase orders and the like are not accepted.

Our contact at Bell also said that you had not made clear the particular version of UNIX that you require. There are many different flavours of UNIX, but the two you are probably interested in are V6 and V7.

UNIX/V6 will run as distributed on a PDP11/34, with one or two trivial alterations, but V7 has a number of extra features and utilities as described in the attachments to this letter. Unfortunately, V7 will not run unaltered on a PDP11/34, requiring a PDP with separated I and D spaces because of the increased size of the system. At first sight, it would seem that you want a V6 license and its associated distribution, but our contact at Bell says it is possible to obtain a combined V6/V7 license and so get the best of both worlds.

Further, we recently learned that it is possible to obtain 'Academic Use' licenses free, if you do not require Bell to send any magnetic tapes or documentation. Obviously you must arrange to obtain these things locally from other UNIX users, and to this end here is a list of contacts in Queensland.

Clary Harridge,
Department of Electrical Engineering,
University of Queensland,
St. Lucia,
Brisbane 4067,
Australia

Ross Gayler,
Psychology Department,
University of Queensland,
St Lucia,
Queensland 4067

C Stubbs,
Dept of Human Movement Studies,
University of Queensland,
St. Lucia,
Brisbane 4067,
Queensland

If your machine does not have a magnetic tape unit, then it is probable that you will have to contact one of these gentlemen, for it is no longer Bell Lab's policy to make software distributions on RK05 discs. For an experienced user it would be a fairly simple matter to copy the tape distribution onto a few RK05s.

As I mentioned earlier, I am the editor of the 'Australian UNIX Users Group Newsletter' (AUUGN for short). Sixty seven UNIX sites, both in Australia and overseas, feel that this publication is sufficiently interesting to pay a modest yearly subscription. I have, or am in the process of obtaining, exchange agreements with UNIX groups in the United Kingdom, U.S.A. and Canada. This exchange of information and software distributions combined with local news makes AUUGN particularly interesting to a new site such as yours. I have enclosed invoices for Vol 1 (backissues) and Vol 2 (current subscription) should you wish to subscribe. Only UNIX license holders may subscribe, so a photocopy of the relevant pages of your agreement should accompany your subscription.

I hope the above information has been of help to you. I tried to call you last week, but was told you would not be available until the 20th of May so I decided to write this letter instead. Should you have any questions please contact me.

Yours sincerely,

Peter Ivanov

Newsletter Editor,
Australian UNIX Users Group

CATEGORIES OF LICENSE

Bell System Software Packages

Commercial Use

Educational institutions, as well as governmental agencies and commercial entities, may be granted the right to use selected Bell System software packages under appropriate licensing terms such as the payment of an appropriate fee for each central processing unit on which the software is used, the holding of the software in confidence, and the use of the software solely for the licensee's internal business purposes.

Educational Institution Administrative Use

Qualified, nonprofit educational institutions may be granted rights to use selected Bell System software packages at reduced fees for "administrative purposes" subject to an obligation to maintain the software in confidence. Use for "administrative purposes" means use directly related to the institution's internal administration and operation. Such use excludes, without limitation, commercial use such as the development of software for sale or license, or use in research funded in whole or part by a third party in consideration for preferential access or rights to the fruits of such research, even though such excluded use may provide financial support for, or otherwise further, the "administrative purposes" of the institution.

Educational and Academic Use

It is our current practice, in appropriate circumstances, to honor requests by nonprofit educational institutions, having on-going teaching or degree-granting programs in compliance with applicable governmental regulations, for licenses without fee for selected software. A small service fee, however, will be charged for the reproduction of tapes and documentation. Such software is provided subject to an obligation to maintain it in confidence, and the sole authorized use of such software pursuant to a royalty-free license is limited to "academic and educational purposes," meaning purposes directly related to such teaching or degree-granting programs. Accordingly, all other uses of such software by or on behalf of your institution are unlicensed and would require a separate license agreement, whether or not such uses by your students or faculty are to provide financial support for, or to otherwise further, your educational and academic functions. Unlicensed uses include, without limitation, administrative uses, commercial uses such as the development of software for sale or license, and all uses for research funded in whole or part by a third party in consideration for preferential access or rights to the fruits of such research or for research not directly related to the teaching or degree-granting programs.

Monash University
CLAYTON VICTORIA AUSTRALIA 3168
TELEPHONE: 03 541 0811 TELEGRAMS: Monashuni Melbourne
TELEX: MONASH 32691
DEPT. OF COMPUTER SCIENCE

1-5-1980

Dear Peter:

Enclosed is our site survey. By the end of this year we shall have a configuration which is very similar to Melbourne Uni's — 1.5 Mbyte, 2 × RM63, 32+ terminal lines. We will also be running an almost identical UNIX version, namely Berkeley's VM UNIX plus U. of Melbourne mods (tty driver etc)

The International User's meeting is not a lost cause, as I had earlier foreshadowed, and we now have a venue for the weekend Oct 18, 19 — Robert Elz should be in touch shortly with more details.

Regards

Ken McDonell.

Kevin Hill
AGSM
21st April, 1980.

G.W. Gerrity,
Department of Computer Science,
RMC Duntroon.

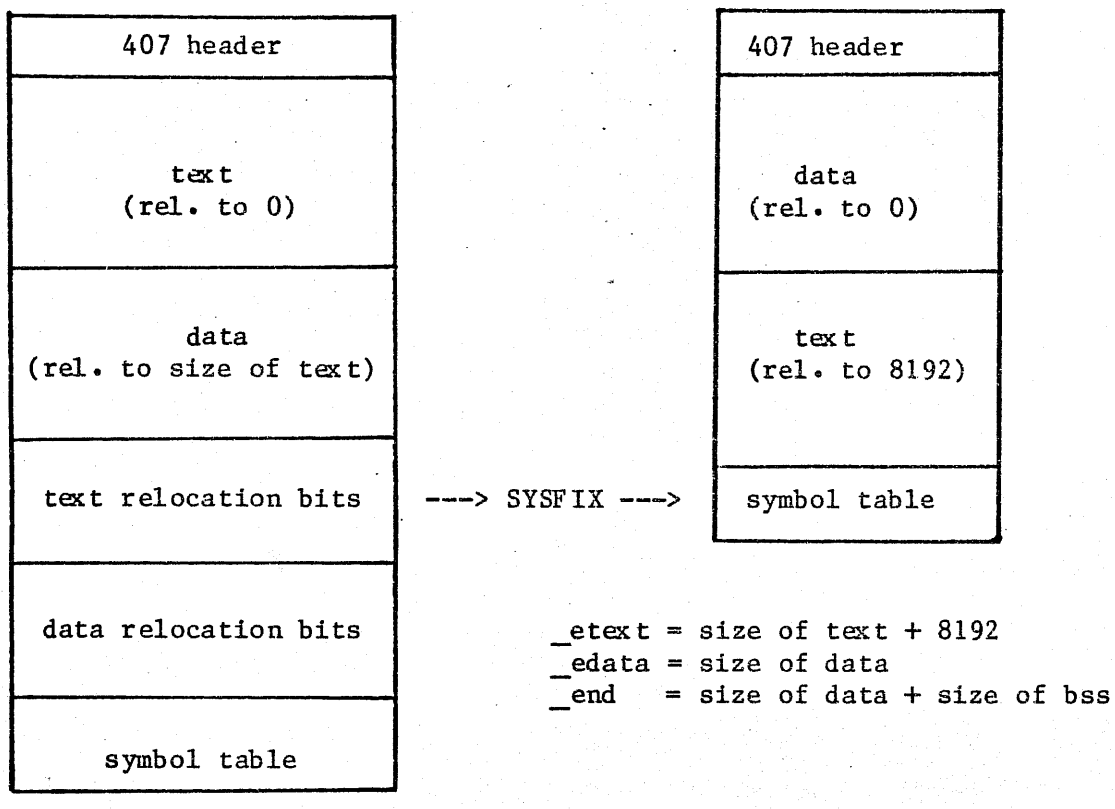
Dear George,

Some weeks ago you asked me several questions regarding the early stages when UNIX is first booted into core on 11/45s. At the time I was unable to give exact answers, but I have since studied the code for the AGSM 11/70, and can now provide much greater detail. As the UNIX on both machines runs in separated I and D spaces, I will discuss my case, but it should not be too different from yours. The following discussion assumes some familiarity with the UNIX a.out format (text, data, bss; relocation bits; header) (see A.OUT(5)).

In 170.s and m70.s (145.s and m45.s in your case), the interrupt vector page and startup routine, plus dumper, are all put in the data segment. The actual text does not start until the trap handlers (code00, code01, etc.). The standard 407-type a.out produced by assembling these things with the libraries is thus different from a normal a.out, in that the code that must be executed first is at the front of the data segment.

However, this 407-type a.out is not used directly. A program called "sysfix" is used to convert it into the unix that lives in your root directory. Sysfix performs several checks which have to do with maximum allowable sizes. It then produces the file unix which differs from the a.out fed into it in several important ways. First, the text and data segments are swapped around (the header information is not altered). Second, the data references and symbols are made relative to zero (the 407 a.out assumes text and data are contiguous, and so data references are offset by the size of the text segment). The data segment must be altered in this manner, because it will eventually be separated from the text segment, and will start at 0 in the data space. Third, the text references and symbols are made relative to a starting address of 8192 (instead of 0). This is because the first instruction space page will actually be pointing to the first data space page when the system is running.

I have illustrated the process over the page.

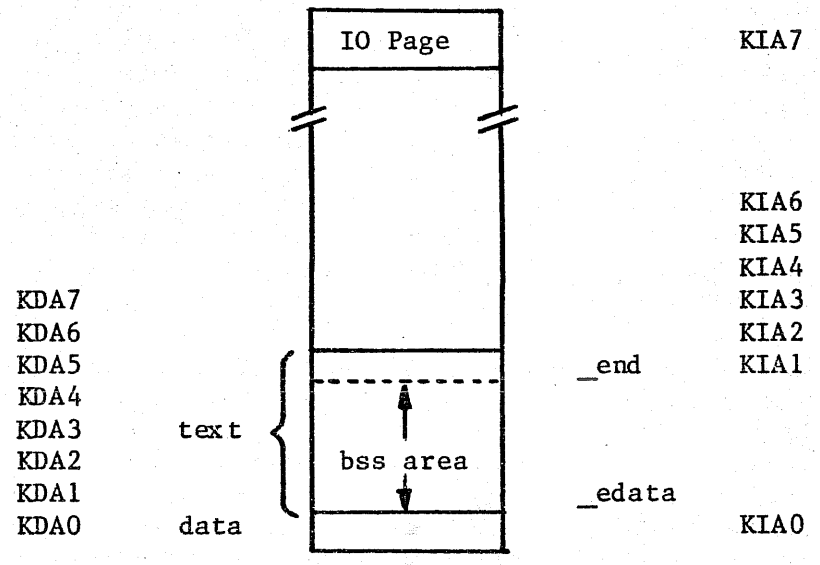


It is because the resulting unix is not a normal a.out (407, 410, 411, or 412) that "db" tends to play up with it. A standard 407 header is left on it, simply so that db can be used on the data space (as the data is at the start of the file, and relative to zero courtesy of sysfix, and as the 407 header implies text and data are contiguous, db is fooled into doing the correct thing). Thus it is quite safe to patch the data segment of unix using db. However, as db will look at the start of unix for the text segment, it cannot be used for patching the text space (by simply referring to text space symbol names). The text can only be patched by subtracting 8192 from the text symbol address (as report by "nm" for example), and then adding the data segment size. Thus, for example, nm tells me that the symbol _chmod lives at 041152T. I can get to the actual text by feeding db the address 041152 - 020000 + 05152 (octal numbers) = 026324 (where I used "size" on unix and converted the data size to octal to get 05152). However, as the addresses and text symbols do not correspond, it is rather difficult to do anything useful! Nonetheless, as the symbols have been adjusted for the final in-core resting places, the symbol table of unix can be used (by db) without fear on /dev/kmem (kernel data space) and /dev/kimem (kernel instruction space). nm will show the correct in-core values.

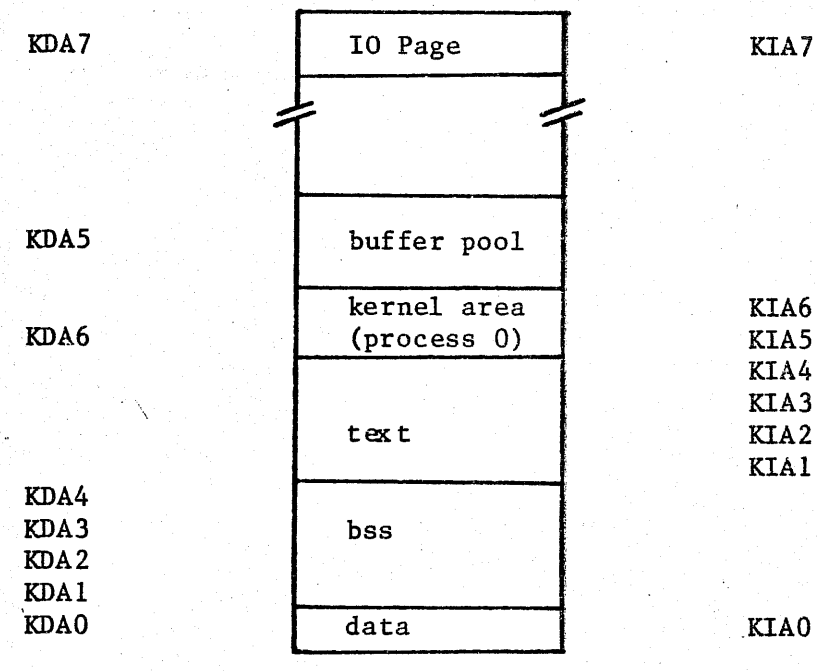
Getting back to unix, there is still a lot to be done! The bootstrap simply reads in the entire file (skipping the 16 byte header), and so lays unix in core exactly as it appears in the file /unix, i.e., data first, followed (contiguously) by text. On jumping to location zero, the first part of unix to be executed is thus the start of the data space. Hence the start-up code in 170.s/m70.s is marked as belonging to the data segment, and not the text segment (which is miles away in high core).

What has to happen now (apart from routine hack tasks like setting up segmentation registers and clearing the bss) is that room must be made for the bss segment, which must of course be contiguous with the data segment. Thus the text segment must be shifted in core by the size of the bss segment, plus the necessary wasted space to take it to a 64-byte boundary. It is in the code that performs this manoeuvre that various limits are set on the maximum size of unix, e.g., it is necessary to be able to access all of the text plus data segments with the 8 data space segmentation registers, and so the maximum allowable text plus data size is 64K bytes (rather than each having this limit, as is the case for a normal separated I and D space program). (My bootstrap places a further limit here: I can only load a maximum of 54K bytes.) Note that these are only limits because of the way it is being done - the method could be changed to allow much larger programs to be loaded.

In a little more detail, soon after unix is read into core and starts executing, the segmentation registers are set up as illustrated below. Memory management is still disabled, so there is no problem executing the start of the data area. However, as the bss segment may be quite large (and is), the text segment cannot be moved by simply using "mov" instructions, as the physical address of its new location may be unreachable using 16 bit addressing. Thus the instruction space registers are set to the places indicated on the illustration, memory management enabled, and the copy done by the interspace "mtpi" instruction, using a temporary one-word stack set up in the data segment. Note that instruction space register 0 must point to the same place as data space register 0, because the code doing this shuffling is at the start of the data segment. Instruction space register 7 points to the IO page, so that this page is not lost (think about it! If contact is lost with this page, it is then impossible to alter the segmentation registers (which live in the IO page) to re-access it). Instruction space register 7 is used because all the data space registers are being used in the copying to allow for the maximum possible text+data image.



On completion of the copy, the bss area is cleared, and data space register 7 is set to point to the IO page (using an "mtpi" instruction, and the fact that instruction space register 7 is pointing to it). Finally, the per-process data area for process 0 is set up and cleared, the stack set up correctly, and a "jsr pc,_main" executed, at which point the instruction space proper is entered, and the real text commences execution. The set up just prior to setting up process 1 is illustrated below. Note that I am using a "MAPPED_BUFFERS" system, in which the buffer pool is paged by data space register 5.

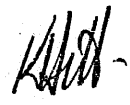


Two final points should be noted. First, the core dumper must be in the first data page, so that when the machine is started at location 44 (which resets the memory management unit back to 16 bit addressing), it can be directly executed (remember that the text segment is miles away in high core). Second, all of the start up code mentioned above, to the point where the infamous "jsr pc,_main; mov $170000,-(sp); clr -(sp); rtt" occurs, must be in the first data page (8192 bytes). This is because the first instruction page is set to point to the first data page (so that execution continues smoothly when memory management is enabled very early in the story). If the start up code crossed the page boundary, execution would continue in the second instruction page (which is much higher in core), instead of where the actual code would be (in the second data page). Effectively, the first instruction page is lost, limiting the text to 56K bytes (as the 8th page is also lost, the limit is actually 48K bytes).

After all this, you may well ask, "Is it all worth it?". Why go to all this trouble (sysfix, shuffling core, etc.)? The answer may be that the interrupt vector page is mapped through kernel data space page 0 - thus the standard a.out format of text followed by data would have the effect of placing the interrupt vectors much higher in core (certainly not in the first physical 512 bytes as on 11/40s etc.). Does this matter? On this matter I am not sure - Peter Ivanov, who has done a DEC 11/70 Hardware Maintenance Course, assures me that no interrupts/aborts/traps cause memory management to be reset (on 11/70s), and so it should not matter if the interrupt vector page is in high memory. To amuse myself, I am in the process of recoding 170.s/m70.s to do exactly this, and will let you know if it works.

I hope this has been of help. In case anyone else is interested, I have asked the AUUGN editor (Peter Ivanov) to consider this letter for publication in the next AUUGN. Please contact me if I can be of further assistance.

Yours sincerely,



Kevin Hill

Adrian Freed
AGSM,
University of New South Wales
P.O. Box 1
Kensington 2033

Dear Pete,

I would like to prompt some discussion in the User's group and would appreciate it if you published this in the newsletter.

Firstly, the issue of terminal drivers. I would like to turn the groups attention away from the implementation details of various individuals terminal drivers to the question: What does the ideal UNIX terminal look like from the USER's point of view? It would be an interesting and almost useful task to design such a terminal. The implementation in firm-ware would be straightforward, using micro-processors.

The next question is related to the fact that with Level 7, came a stable definition of the C language. Could the group agree on a formatting standard for C? Judging by the number of different styles of formatting I see at AGSM and at Elec. Eng., this is unlikely. This questions the usefulness of programs like 'para' or 'cb'. I suggest that if we can agree on a style that is not too abhorant to most people, the group sponsor's someone to adapt 'cb' to produce C in this style. This program could then be distributed around the UNIX sites and it would be up to individuals to 'customise' from this standard.

I hope I am not making too much of this issue, but to start the ball rolling I have included a listing of the output of the local 'para', which satisfies many people on this campus.

Next to the question of Standard I/O. People are easily confused when talking about I/O libraries. There is only ONE Standard I/O library and it is documented in the Level 7 manuals as the ONLY I/O library. It is the one with 'fopen("filename", "r")'. In the interests of portablility can I suggest that everyone makes an attempt to use this library. Can I also suggest that any complaints about the library be voiced.

Finally, some people may be aware of the UNIX reference card, printed at Bell Labs. How many times do you forget the flags to find(I) or the argument order for comm(I)? This is just when the card would be handy as it summarises the manual entries. It has sections for UNIX commands, Command details, Troff and Nroff, C system calls, Site-dependent commands. Is there any interest in producing a local one? This is probably best done when the issue of upgrade to Level 7 is resolved.

I am sure you will agree, Pete, that the newsletter lacks contributions, in general. Let's hope my comments initiate some discussion.

Come on, everyone out there, get those letters rolling in!!!

Adrian Freed

```
int argi;
int ldivr;
main(argc, argv)
int argc;
char *argv[];
{
        register char *cp;
        register wd;


        argc--;
        for(argi = 1; argi <= argc; argi++)
        {
                for(cp = argv[argi]; *cp; ++cp)
                {
                        if(*cp == '')
                        {
                                if(*++cp == 'n')
                                {
                                        putchar('0);
                                        continue;
                                }
                                else if(*cp == 'c')
                                        exit(0);
                                else if(*cp == '')
                                {
                                        putchar('');
                                        continue;
                                }
                                else if(*cp == '0')
                                {
                                        wd = 0;
                                        while(*++cp >= '0' && *cp <= '7')
                                        {
                                                wd =<< 3;
                                                wd =| (*cp-'0');
                                        }
                                        putchar(wd);
                                        --cp;
                                        continue;
                                }
                                else
                                        --cp;
                        }
                        putchar(*cp);
                }
                putchar(argi == argc?'0:' ');
        }
        exit(0);
}

putchar(c)
char c;
{
        write(1, &c, 1);
}
```

IN REPLY PLEASE QUOTE:

TELEPHONE: 692 3491
692 1122
EXT 3491

UNIVERSITY COMPUTING CENTRE
THE UNIVERSITY OF SYDNEY
NSW 2006

28 May 1980

It's that time of year again - AUUG meeting time. This one is to be held at -

Seminar Room
University Computing Centre H08
Sydney University

on Wednesday, July 2nd, 1980 from 0930 to 1600. Its cost will be $10 which will include a midday nosh.

Fill in, tear off, and post the form below (+ cheque/ postal note/money order) to -

Unix Users Group
University Computing Centre H08
Sydney University

I/We will be sending _____ person/people to the AUUG meeting. Enclosed is a cheque/postal note/money order for $_____ to cover everything.

I/We would/would not like to give a talk entitled _____ _____ to the rest of the group.

Name(s): _____

Affiliation: _____

Signature: _____