Getting started with...

Berkeley Software for UNIX† on the VAX‡

(The third Berkeley Software Distribution)

A package of software for UNIX developed at the Computer Science Division of the University of California at Berkeley is installed on our system. This package includes a new version of the operating system kernel which supports a virtual memory, demand-paged environment. While the kernel change should be transparent to most programs, there are some things you should know if you plan to run large programs to optimize their performance in a virtual memory environment. There are also a number of other new programs which are now installed on our machine; the more important of these are described below.

Documentation

The new software is described in two new volumes of documentation. The first is a new version of volume 1 of the UNIX programmers manual which has integrated manual pages for the distributed software and incorporates changes to the system made while introducing the virtual memory facility. The second volume of documentation is numbered volume 2c, of which this paper is a section. This volume contains papers about programs which are in the distribution package.

Where are the programs?

Most new programs from Berkeley reside in the directory **/usr/ucb.** A major exception is the C shell, *csh*, which lives in **/bin.** We will later describe how you can arrange for the programs in the distribution to be part of your normal working environment.

Making use of the Virtual Memory

With a virtual memory system, it is no longer necessary for running programs to be fully resident in memory. Programs need have only a subset of their address space resident in memory, and pages which are not resident in memory will be *faulted* into memory if they are needed. This allows programs larger than memory to run, but also places a penalty on programs which do not exhibit *locality.* It is important to structure large programs so that at any one time they are referring to as small a number of pages as possible.

If you are going to create very large programs, then you should know about a new demand load format. This format causes large programs to begin execution more rapidly, without loading in all the pages of the program before execution begins. It is most suitable for programs which are large (say > 50K bytes of program and initialized data), and especially when a program has a large number of facilities, not all of which are used in any one run. To create an executable file with this format, you use the **−z** loader directive; thus you can say ‘‘cc −z ...’’ or ‘‘f77 −z ...’’. The *file* command will show such files to be ‘‘demand paged pure executable’’ files. See the manual page for *ld* in section 1 and *a.out* in section 5 of volume 1 of the manual for more information.

If you have or are writing a large program which creates new processes as children, then you should know about **vfork** system call. The **fork** system call creates a new process by copying the data space of the parent process to create a child process. In a virtual environment this is very expensive. **Vfork** allows creation of a new process without copying the parent’s address space by letting the parent execute in the child’s system context. The parent can set up the input/output for the child and then return to its own context after a call to **exec** or **_exit.** If you use the standard I/O routine **system()** to execute commands from within your programs, then **vfork** will be used automatically. If you have been calling **fork** yourself, you should read the manual page for **vfork** and use it when you can.

---

†UNIX is a trademark of Bell Laboratories.

‡VAX is a trademark of Digital Equipment Corporation.

In order that efficient random access be permitted in a portable way to large data files, a pair of new system calls has been added: **vread** and **vwrite.** These calls resemble the normal UNIX **read** and **write** calls, but are potentially much more efficient for sparse and random access to large data files. **Vread** does not cause all the data which is virtually read to be immediately transferred to your address space. Rather, the data can be fetched as required by references, at the systems discretion. At the point of the **vread,** the system merely computes the disk block numbers of the corresponding pages and stores these in the page tables. Faulting in a page from the file system is thus no more expensive than faulting in a page from the paging device. In both cases all the mapping information is immediately available or can be easily computed from incore information. **Vwrite** works with **vread** to allow efficient updating of large data which is only partially accessed, by rewriting to the file only those pages which have been modified.

Downward compatibility to non-virtual systems is achieved by the fact that **read** and **write** calls have the same semantics as **vread** and **vwrite** calls; only the efficiency is different. If you have programs which access large files, and do so sparsely, read the manual pages for **vread** and **vwrite** in section 2 of volume 1 of the manual.

File System blocksize changes

The size of blocks in the file system has been changed to improve the throughput of the disks. The constant ''512'' is not the ''best'' size to use when reading/writing the disk; rather you should use ''BUFSIZ'' blocks as defined in the include file <stdio.h>. Because this constant has been changed **all old .o files must be removed and then recreated.** They will not load successfully, since the distributed library routines assume that BUFSIZ is 1024 (its current value at our installation). Old executable images may be preserved by running the command

1kfix file

on each such file. Note that this only works for ''a.out'' type files, not ''.o's.'' It is recommended that all old programs be recompiled to take advantage of the larger disk block size.

New Languages for the VAX

There are now available interpreters for APL and Pascal for the VAX, and a LISP system supporting a dialect of LISP compatible with a large subset of MACLISP. The APL interpreter is the 11 version, moved to the VAX, and now has a large workspace capability (but has not been extensively used.) The Pascal system has been used extensively for instruction and research and is the same system which was available on the PDP-11. The only limitations of the Pascal system are a maximum of 32K bytes per stack frame (due to the implementation of the interpreter), and 64K bytes per variable allocated with *new.* Essentially arbitrary sized programs can be run with the system, which supports a very standard Pascal with no language extensions. The Pascal system features very good error diagnostics, and includes a source level execution profiling facility.†

The LISP system, ''Franz Lisp'', was developed at Berkeley as part of a project to move the MIT MACSYMA system from the PDP-10 to the VAX. A compiler *liszt* for Franz Lisp, written at Bell Laboratories, is also included with the system.

For more information about APL refer to its manual page in volume 1 of the manual. The Pascal system consists of the programs **pi, px, pix, pxp, pxref,** and **pic,** all of which are documented in section 1 of volume 1 of the manual. There is also a paper introducing the system in volume 2c. The LISP system is described in *The Franz Lisp Manual* in volume 2c of the manual.

A display editor − *vi*

The system includes the latest version of the display editor *vi* which runs on a large number of intelligent and unintelligent display terminals. This editor runs using a terminal description data base and a library of routines for writing terminal independent programs which is also supplied. The editor has a mnemonic command set which is easy to learn and remember, and deals with the hierarchical structure of documents in a natural way. Editor users are protected against loss of work if the system crashes, and

† A compiler for Pascal based on this system is currently being developed, but is not part of this distribution.

against casual mistakes by a general *undo* facility as well as visual feedback. The editor is quite usable even on low speed lines and dumb terminals.

For users who prefer line oriented editing, the *ex* command enters the same editor, but in a line oriented editing mode. For beginners who have never used a line editor before, there is a version of the editor known as *edit* which has a well-written tutorial introducing it.

For more information about *edit* see *Edit: a Tutorial* in volume 2c of the manual. The line editor features are described in the *Ex Reference Manual* which is in volume 2c of the manual. Also in volume 2c are *An Introduction to Display Editing with Vi* and a *vi* reference card.

Command and mail processing programs

There is also a new command processor *csh* which caters to interactive users by providing a history list of recent commands, which can be easily repeated. The shell also has a powerful macro-like aliasing facility which can be used to tailor a friendly command environment. *Csh* is implemented so that both it and the standard shell **/bin/sh** can be run on the same system.

The *Introduction to the C shell* introduces the shell. If you have used the standard shell, then you should especially read about the *history* and *alias* mechanisms of the shell.

In order that the manual distributed with the tape correspond to the commands which are available to you, the default execution search path is

```
PATH=:/usr/ucb:/bin:/usr/bin
```

in the language of **/bin/sh** or

```
setenv PATH :/usr/ucb:/bin:/usr/bin:
set path=(. /usr/ucb /bin /usr/bin)
```

in the language of **/bin/csh.**

For sending and receiving mail, a new interactive mail processing command provides a hospitable environment, supporting items such as subject and carbon copy fields, and allowing creation of distribution lists. This command also has a mail reading mode which makes it convenient to deal with large volumes of mail. See the manual page for *mail* in section 1, volume 1 of the manual, and the *Mail reference Manual* in volume 2c of the manual for more details.

Better debugger support

A version of the symbolic debugger *sdb* is provided which now can debug FORTRAN 77 programs. The assembler has been rewritten and the C compiler modified to reduce greatly the overhead of using the symbolic debugger, making it much more feasible for heavy use. If you are interested, then you should read the new document for *sdb*, provided in volume 2c.

Other software

Other new programs include programs to simulate the phototypesetter on 200 bpi plotters, a common system messages facility, routines for data compression, a modified version of the standard I/O library permitting simultaneous reads and writes, a network for connecting heterogeneous UNIX systems at low cost (1 tty port per connection per machine and no system changes), and a new, flexible macro package for n/troff −*me*. New command **whatis** and **apropos** can be used to identify programs and to locate commands based on keywords. Try

```
cd /usr/ucb
whatis *
```

and to find out about Pascal:

```
apropos pascal
```

Monitoring the new system

If you want to see what is happening in the new system, you can use the new **vmstat** command, described in section 1 of the manual, which shows the current virtual load on the system. The system recomputes the information printed by **vmstat** every five seconds, so a ‘‘vmstat 5’’ is a good command to try.

To see what processes are active virtual processes, you can do

```
ps av
```

The command

```
ps v
```

will print only the active processes which you are running.