1544

Bell Laboratories Cover Sheet for Technical Memorandum

The information contained herein is for the use of employees of Bell Laboratories and is not for publication (see GEI 13.9-3)

Title: **A Memory Device Driver for UNIX™**

Date: **May 9, 1980**

Other Keywords: **memory files**
memory filesystems
block device
character device

TM: **80-3168-5**

Author(s)	Location	Extension	Charging Case: 49408-120
Alan L. Glasser	**HO 1E-335**	**6569**	Filing Case: **40324-2**

ABSTRACT

A device driver that simulates a fixed-head disk with main memory is described. Both *block* and *character* interfaces are provided; hence, file systems can be "mounted" in memory. The maximum size of such a file is user specifiable; a set of preset size files are provided with sizes from 64K bytes to 512K bytes. The user interface, implementation, performance and possible future work are presented.

TM-80-3168-5

Pages Text: 3	Other: 1	Total: 4
No. Figures: 0	No. Tables: 0	No. Refs.: 3

LAUTENBACH,DEBORAH A CM
PY2H210 05/23/80
SUBJECT MATCH UNOS

Bell Laboratories

subject: **A Memory Device Driver for UNIX™**
Charge Case 49408-120
File Case 40324-2

date: **May 9, 1980**

from: **Alan L. Glasser**
HO 3168
1E-335 x6569

TM 80-3168-5

MEMORANDUM FOR FILE

1. INTRODUCTION

With the cost of memory dropping, and larger amounts of memory becoming commonplace we have begun investigating new ways to utilize that memory. One technique that appears to be attractive for DEC PDP-11/70 UNIX systems is to use memory to contain entire files or file systems. This technique is not new; it was used by Ken Thompson and Rick Brandt when they ran benchmarks on the first PDP-11/70 UNIX system [1]. The details of that early memory driver were never published. This memorandum documents a memory driver that runs on a PDP-11/70 under the PWB 2.0 (UNIX/TS 1.1) version of the UNIX kernel.

2. USER INTERFACE

The driver provides both *block* and *character* (*raw*) interfaces. The *minor device number* [2] is used to further divide the files provided into two categories: *auto-allocate* and non-*auto-allocate*. Auto-allocate files cause memory space to be allocated when the file is opened; non-auto-allocate files require *stty* [3] system calls to cause allocation.* Space is freed only via *stty*. This can lead to some unfortunate situations: if an auto-allocate special file is used for the temporary file of an *fsck* command, the space allocated is not automatically freed when the *fsck* terminates. The system, if allowed to continue in this way, will run with a reduced complement of useable primary memory.

The block device interface allows file systems to be mounted in memory, and, of course, utilizes the buffer cache in the kernel. This interface requires a minimum of one data copy operation, and, depending on cache performance, possibly two copy operations for each *read* and *write*. The *raw* interface, however, copies the data only once.

3. PERFORMANCE

The reason this driver was initially written was to provide a high performance temporary file for *fsck*. Additionally, this has been our most frequent use of this driver. The driver has been very convenient to use, and is faster than using a disk file. We have two PDP-11/70 systems with only a single RP06 drive per system. When the *fsck* temporary file resides on the same

---

* The attached manual page provides the necessary details to actually use the driver.

drive as the file system being checked, the resultant head movement causes rather poor performance. This driver provides significantly better performance than using a single disk drive, and somewhat better performance than using an additional disk drive.

We have not made extensive use of the block device interface, and thus do not have much performance data to quote. One series of experiments was performed, on an otherwise quiet system, with /tmp mounted on /dev/mos11. A program was compiled a few times. The real time per compile was approximately 16.5 seconds (as reported by the *time* command). With /tmp in its normal place on disk the time per compile was approximately 18.1 seconds—not a significant or outstanding improvement. However, there was little real disk traffic due to a 127 block buffer cache.

No tests have been made under load. While this driver may provide significant performance improvements (mounting /tmp, /bin or /lib, perhaps), much measurement must be done to find the area of maximum improvement.

4. IMPLEMENTATION

All that is required to implement this driver is to be able to copy memory. There are three classes of copying: (1) the *raw* case (physical address and user virtual address), (2) the system addressable buffer case (physical address and kernel virtual address), and (3) the non-addressable buffer case (two physical addresses). The first case uses the *copyio* routine, with a loop to copy no more than *BSIZE* (512) bytes at a time (*copyio* cannot copy arbitrarily large amounts of memory; *BSIZE* bytes seems like a reasonable amount to do at a time). The second case is a simple *copyio*. The last case takes advantage of the fact that both non-addressable buffers and the memory for these special files begin on *click* boundaries. The last case makes repeated calls on the *copyseg* routine.

5. FUTURE WORK

For the case of non-addressable system buffers one would like to devise a scheme that would eliminate the need for copying the data. After all, the data is in memory. Some way of changing the buffer address to point directly at the special file memory, with the ability to restore the buffer address is needed. No solutions are known by the author; this problem is mentioned in the hope of a reader discovering a solution.

Finally, this driver might lend itself to being the fast, small part of a *split-device* file system. A split-device file system is a file system where one portion (the *ilist*) resides on a fast, but small device, and the remainder of the file system resides on a slower, but large device. A split device file system is implemented with a pseudo-driver that forwards requests to two other drivers, where the boundary of the file system and the *names* of the other two devices are set via an *stty* system call. The split-device file system offers some possible performance improvement (about 45% of all file system traffic for typical *user* file systems is to the ilist) at the expense of rather obvious crash-recovery problems.

6. ACKNOWLEDGEMENT

The author gratefully acknowledges the debugging assistance provided by David M. Ungar when we both learned the sad truth that *copyio* can't copy an arbitrary amount of data.

REFERENCES

[1] R. B. Brandt and K. Thompson. Benchmark of UNIX on DEC 11/70. TM 75-8234-4, 75-1271-4.

[2] K. Thompson. UNIX Implementation. *The Bell System Technical Journal* **57**, 6 (July-August 1978, Part 2), pp. 1931-46.

[3] stty(2) in PWB/UNIX User's Manual, Release 2.0, June 1979.

Alan L. Glasser

HO-3168-ALG-alg

Alan L. Glasser

Att.
Manual page for *mos*(4).

NAME

mos — memory fixed-head disk simulator

DESCRIPTION

The files **mos0, mos1, ..., mos7, mos10, mos11, ..., mos17** are the interfaces to a driver that simulates a fixed-head disk with main memory. The files are not "exclusive use". It is expected that these files will be used more in a shared environment than in an exclusive use environment (e.g., tables that are created dynamically, relatively small, and referenced often).

The first 8 files (**mos0, ..., mos7**) are equivalent; 8 different files are provided for 8 possible different and simultaneous applications. The maximum size of one of these files is set via an *stty*(2) system call.

The *stty* function expects arguments of the form:

```
stty(mosfd, arg)
struct {
        int     code;
        int     *size;
        int     fill;
} *arg;
```

The codes are defined in */usr/include/sys/mos.h* and are:

```
# define GETSZ  0
# define SETSZ  1
# define FREE   2
```

GETSZ causes the current size to be stored in **size*. SETSZ causes the current size to be set to **size*, an error (ENOSPC) is returned if there isn't space. FREE frees the memory; *size* is unused in this subfunction. All sizes are in 64 byte basic memory units (*clicks*).

A typical sequence for using such a file is: open, stty to allocate space, use the file, stty to free the space, close the file. Changing the allocation after the initial allocation is unwise. The last 8 files (**mos10, ..., mos17**) allocate space automatically when the file is opened. The following table gives the size per file:

File	Size (K bytes)
/dev/mos10	64
/dev/mos11	128
/dev/mos12	192
/dev/mos13	256
/dev/mos14	320
/dev/mos15	384
/dev/mos16	448
/dev/mos17	512

All of these sizes are the maximum size the file can attain. The application must track the actual number of bytes written. In addition to these 16 block devices, there are also 16 corresponding character (*raw*) devices. As with other such devices, the names of the raw devices begin with an *r* (e.g., */dev/rmos10*).

CAVEAT

Undisciplined use of these files can result in severe memory fragmentation. These files should be opened and space allocated early in the system initialization process (*rc*(8)) to avoid serious fragmentation.