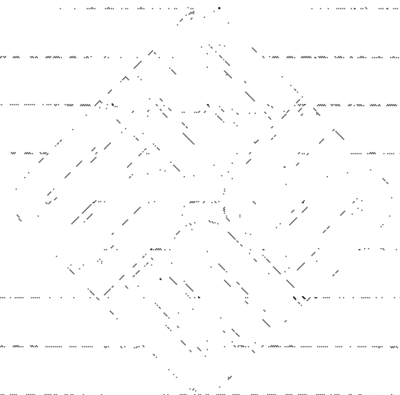
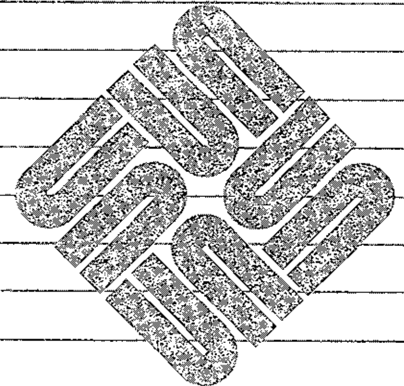
System Internals Manual
for the Sun Workstation



Sun Microsystems, Inc. • 2550 Garcia Avenue • Mountain View, CA 94043 • 415-960-1300

System Internals Manual *for the* Sun Workstation



Sun Microsystems, Inc. • 2550 Garcia Avenue • Mountain View, CA 94043 • 415-960-1300

Part No: 800-1181-01
Revision D of 15 May, 1985

Credits and Acknowledgements

The chapters of this manual were originally derived from the work of many people at the University of California at Berkeley and other noble institutions. Their names and the titles of the original works appear here.

A Fast File System for UNIX
Marshall Kirk McKusick, William N. Joy Samuel J. Leffler, and Robert S. Fabry of the University of California, Berkeley.

Using ADB to Debug the UNIX Kernel
is a revised version of an earlier paper by Sam Leffler and Bill Joy of the University of California at Berkeley.

Trademarks

Multibus is a trademark of Intel Corporation.

Sun Workstation is a trademark of Sun Microsystems Incorporated.

UNIX is a trademark of Bell Laboratories.

Copyright © 1983 by Sun Microsystems.

This publication is protected by Federal Copyright Law, with all rights reserved. No part of this publication may be reproduced, stored in a retrieval system, translated, transcribed, or transmitted, in any form, or by any means manual, electric, electronic, electro-magnetic, mechanical, chemical, optical, or otherwise, without prior explicit written permission from Sun Microsystems.

Revision History

Revision	Date	Comments
A	15th July 1983	First release of this Manual.
B	15th August 1983	Second Release of this manual entailed a complete reorganization and some rewriting of the individual articles.
C	1st November 1983	Third Release of this manual entailed minor corrections and updates.
D	15 May 1985	Added section on using ***adb*** to debug the UNIX kernel. Minor corrections and updates. Broke the ***Device Driver Reference Manual*** into a separate self-contained document. The ***Networking Implementation Notes*** Paper is now part of the manual entitled ***Networking on the Sun Workstation.***

System Internals Manual

Table of Contents

This manual provides several papers on the internals of the Sun UNIX System:

1. Using ADB to Debug the UNIX Kernel.

2. A Fast File-System for UNIX.

3. The CPU PROM Monitor.

Using ADB to Debug the UNIX Kernel

Contents

1. Introduction **1**
1.1. Getting Started 1
1.2. Establishing Context 2

2. ADB Command Scripts **3**
2.1. Extending the Formatting Facilities 3
2.2. Traversing Data Structures 6
2.3. Supplying Parameters 7
2.4. Standard Scripts 8

3. Generating ADB Scripts with Adbgen **9**

4. Summary **9**

Using ADB to Debug the UNIX Kernel

This document describes the use of extensions made to the UNIX† debugger *adb* for the purpose of debugging the UNIX kernel. It discusses the changes made to allow standard *adb* commands to function properly with the kernel and introduces the basics necessary for users to write *adb* command scripts which may be used to augment the standard *adb* command set. The examination techniques described here may be applied to running systems, as well as the post-mortem dumps automatically created by the *savecore*(8) program after a system crash. The reader is expected to have at least a passing familiarity with the debugger command language.

1. Introduction

Modifications have been made to the standard UNIX debugger *adb* to simplify examination of post-mortem dumps automatically generated following a system crash. These changes may also be used when examining UNIX in its normal operation. This document serves as an introduction to the **use** of these facilities, and should not be construed as a description of ***how to debug the kernel.***

1.1. Getting Started

Use the **−k** option on the *adb* command when you want to examine the UNIX kernel:

```
tutorial% adb -k /vmunix /dev/mem
```

The **−k** option makes *adb* partially simulate the Sun-2 virtual memory hardware when accessing the *core* file. In addition the internal state maintained by the debugger is initialized from data structures maintained by the UNIX kernel explicitly for debugging[1]. A post-mortem dump may be examined in a similar fashion,

```
tutorial% adb -k vmunix.? vmcore.?
```

where the appropriate version of the saved operating system image and core dump are supplied in place of "?".

† UNIX is a trademark of Bell Laboratories.

[1] If the −k flag is not used when invoking *adb* the user must explicitly calculate virtual addresses. With the −k option *adb* interprets page tables to automatically perform virtual to physical address translation.

1.2. Establishing Context

During initialization ***adb*** attempts to establish the context of the "currently active process" by examining the value of the kernel variable ***panic_regs***. This structure contains the register values at the time of the call to the ***panic*** routine. Once the stack pointer has been located, the command

```
$c
```

generates a stack trace. An alternate method may be used when a trace of a particular process is required: see section 2.3.

2. ADB Command Scripts

2.1. Extending the Formatting Facilities

Once the process context has been established, the complete ***adb*** command set is available for interpreting data structures. In addition, a number of ***adb*** scripts have been created to simplify the structured printing of commonly referenced kernel data structures. The scripts normally reside in the directory ***/usr/lib/adb***, and are invoked with the **$<** operator. A later table lists the "standard" scripts.

As an example, consider the following listing which contains a dump of a faulty process's state (what the user types is shown in **bold typewriter text like this**.

```
tutorial% adb -k vmunix.3 vmcore.3
sbr 50030 slr 51e
physmem 3c0
$c
_panic[10fec](5234d) + 3c
_ialloc[16ea8](d44a2,2,dff) + c8
_maknode[1d476](dff) + 44
_copen[1c480](602,-1) + 4e
_creat() + 16
_syscall[2ea0a]() + 15e
level5() + 6c
5234d/s
_nldisp+175:    ialloc: dup alloc
u$<u
_u:
_u:             pc
                4be0
_u+4:           d2              d3              d4              d5
                13b0            0               0               0
_u+14:          d6              d7
                0               2604
_u+1c:          a2              a3              a4              a5
                0               c7800           5a958           d7160
_u+2c:          a6              a7
                3e62            3e48
_u+34:          sr
                27000000
_u+38:          p0br            p0lr            p1br            p1lr
                105000          40000022        fd7f4           1ffe
_u+48:          szpt            sswap
                1               0
_u+50:          procp           ar0             comm
                d7160           3fb2            dtime^@^@^@^@^@
_u+158:         arg0            arg1            arg2
                1001c           -1              ffffa4
_u+178:         uap             qsave                           error
                2958            2eb46           1               0
_u+1b2:         rv1             rv2             eosys
                0               14cac           0
```

```
_u+1bc:     uid     gid
            49      10
_u+1c0:     groups
            10              -1              -1              -1
            -1              -1              -1              -1
_u+1e0:     ruid    rgid
            49      10
_u+1e4:     tsize           dsize           ssize
            7               1b              2
_u+344:     odsize          ossize          outime
            0               0               0
_u+350:     signal
            0               0               0               0
            0               0               0               0
            0               0               0               0
            0               0               0               0
            0               0               0               0
            0               0               0               0
            0               0               0               0
            0               0               0               0
            sigmask
            0               0               0               0
            0               0               0               0
            0               0               0               0
            0               0               0               0
            0               0               0               0
            0               0               0               0
            0               0               0               0
            0               0               0               0
_u+450:     onstack         oldmask         code
            0               80002           0
_u+45c:     sigstack        onsigstack
            0               0
_u+464:     ofile
            d66b4           d66b4           d66b4           0
            0               0               0               0
            0               0               0               0
            0               0               0               0
            0               0               0               0

            pofile
            0       0       0       0       0       0       0       0
            0       0       0       0       0       0       0       0
            0       0       0       0
_u+4c8:     cdir            rdir            ttyp            ttyd    cmask
            d44a2           0               5c6c0           0       12

            ru & cru
_u+4d8:     utime                           stime
            0               0               0               35b60
_u+4e8:     maxrss          ixrss           idrss           isrss
            9               35              43
_u+4f8:     minflt          majflt          nswap
            0               5               0
```

```
_u+504:      inblock           oublock           msgsnd            msgrcv
             3                 7                 0                 0
_u+514:      nsignals          nvcsw             nivcsw
             0                 12                4
_u+520:      utime                               stime
             0                 0                 0                 0
_u+530:      maxrss            ixrss             idrss             isrss
             0                 0                 0
_u+540:      minflt            majflt            nswap
             0                 0                 0
_u+54c:      inblock           oublock           msgsnd            msgrcv
             0                 0                 0                 0
_u+55c:      nsignals          nvcsw             nivcsw
             0                 0                 0
0d7160$<proc
d7160:       link              rlink             addr
             590e0             0                 1057f4
d716c:       upri     pri      cpu      stat     time     nice     slp
             066      024      020      03       01       024      0
d7173:       cursig            sig
             0                 0
d7178:       mask              ignore            catch
             0                 0                 0
d7184:       flag              uid      pgrp     pid      ppid
             8001              31       2f       2f       23
d7190:       xstat             ru                poip     szpt     tsize
             0                 0                 0        1        7
d719e:       dsize             ssize             rssize            maxrss
             1b                2                 5                 fffff
d71ae:       swrss             swaddr            wchan             textp
             0                 0                 0                 d8418
d71be:       p0br              xlink             ticks
             105000            0                 15
d71c8:       %cpu                                ndx      idhash   pptr
             0                                   6        2        d70d4
d71d4:       real itimer
             0                 0                 0                 0
d71e4:       quota             ctx
             0                 5f236
0d8418$<text
d8418:       daddr
             284               0                 0                 0
             0                 0                 0                 0
             0                 0                 0                 0

             ptdaddr           size              caddr             iptr
             184               7                 d7160             d47e0

             rssize   swrss    count    ccount   flag     slptim   poip
             4        0        01       01       042      0        0
```

The cause of the crash was a "panic" (see the stack trace) due to the a duplicate inode allocation detected by the *ialloc* routine The majority of the dump was done to illustrate the use of the command scripts used to format kernel data structures. The u script, invoked by the command

u$<u, is a lengthy series of commands which pretty-prints the user vector. Likewise, `proc` and `text` are scripts used to format the obvious data structures. Let's quickly examine the `text` script (the script has been broken into a number of lines for readability here; in actuality it is a single line of text).

```
./"daddr"n12Xn\
"ptdaddr"16t"size"16t"caddr"16t"iptr"n4Xn\
"rssize"8t"swrss"8t"count"8t"ccount"8t"flag"8t"slptim"8t"poip"n2x4bx
```

The first line produces the list of disk block addresses associated with a swapped out text segment. The `n` format forces a new-line character, with 12 hexadecimal integers printed immediately after. Likewise, the remaining two lines of the command format the remainder of the text structure. The expression `16t` tabs to the next column which is a multiple of 16.

The majority of the scripts provided are of this nature. When possible, the formatting scripts print a data structure with a single format to allow subsequent reuse when interrogating arrays of structures. That is, the previous script could have been written

```
./"daddr"n12Xn
+/"ptdaddr"16t"size"16t"caddr"16t"iptr"n4Xn
+/"rssize"8t"swrss"8t"count"8t"ccount"8t"flag"8t"slptim"8t"poip"n2x4bx
```

but then reuse of the format would have invoked only the last line of the format.

2.2. Traversing Data Structures

The *adb* command language can be used to traverse complex data structures. One such data structure, a linked list, occurs quite often in the kernel. By using *adb* variables and the normal expression operators it is a simple matter to construct a script which chains down the list printing each element along the way.

For instance, the queue of processes awaiting timer events, the callout queue, is printed with the following two scripts:

```
callout:

        calltodo/"time"16t"arg"16t"func"
        *(.+0t12)$<callout.next

callout.next:

        ./D2p
        *+>l
        ,#<l$<
        <l$<callout.next
```

The first line of the script `callout` starts the traversal at the global symbol *calltodo* and prints a set of headings. It then skips the empty portion of the structure used as the head of the queue. The second line then invokes the script `callout.next` moving "." to the top of the queue (`*+` performs the indirection through the link entry of the structure at the head of the queue).

`callout.next` prints values for each column, then performs a conditional test on the link to the next entry. This test is performed as follows,

`*+>l` Place the value of the "link" in the *adb* variable "<l".

`,#<l$<` If the value stored in "<l" is non-zero, then the current input stream (i.e. the script `callout.next`) is terminated. Otherwise, the expression "#<l" will be zero, and the "$<" will be ignored. That is, the combination of the logical negation operator "#", *adb* variable "<l", and "$<" operator creates a statement of the form,

```
if (!link) exit;
```

The remaining line of `callout.next` simply reapplies the script on the next element in the linked list.

A sample *callout* dump is shown below.

```
tutorial% adb -k /vmunix /dev/mem
sbr 50030 slr 51e
physmem 3c0
$<callout
_calltodo:
_calltodo:        time          arg           func
d9fc4:            5             0             _roundrobin
d9f94:            1             0             _if_slowtimo
d9fd4:            1             0             _schedcpu
d9fa4:            3             0             _pffasttimo
d9fe4:            0             0             _schedpaging
d9fb4:            15            0             _pfslowtimo
d9ff4:            12            0             _arptimer
da044:            736           d7390   _realitexpire
da004:            206           d6fbc         _realitexpire
da024:            649           d741c   _realitexpire
da034:            176929  d7304   _realitexpire
```

2.3. Supplying Parameters

If one is clever, a command script may use the address and count portions of an *adb* command as parameters. An example of this is the **setproc** script used to switch to the context of a process with a known process-id;

```
0t99$<setproc
```

The body of **setproc** is

```
.>4
*nproc>l
*proc>f
$<setproc.nxt
```

while **setproc.nxt** is

```
(*(<f+0t42)&0xffff)="pid "D
,#(((*(<f+0t42)&0xfff))-<4)$<setproc.done
<1-1>1
<f+0t140>f
,#<1$<
$<setproc.nxt
```

The process-id, supplied as the parameter, is stored in the variable <4, the number of processes is placed in <1, and the base of the array of process structures in <f. `setproc.nxt` then performs a linear search through the array until it matches the process-id requested, or until it runs out of process structures to check. The script `setproc.done` simply establishes the context of the process, then exits.

2.4. Standard Scripts

The following table summarizes the command scripts currently available in the directory */usr/lib/adb*.

Standard Command Scripts

Name	*Use*	*Description*
buf	*addr*$<buf	format block I/O buffer
callout	$<callout	print timer queue
clist	*addr*$<clist	format character I/O linked list
dino	*addr*$<dino	format directory inode
dir	*addr*$<dir	format directory entry
file	*addr*$<file	format open file structure
filsys	*addr*$<filsys	format in-core super block structure
findproc	*pid*$<findproc	find process by process id
ifnet	*addr*$<ifnet	format network interface structure
inode	*addr*$<inode	format in-core inode structure
inpcb	*addr*$<inpcb	format internet protocol control block
iovec	*addr*$<iovec	format a list of *iov* structures
ipreass	*addr*$<ipreass	format an ip reassembly queue
mact	*addr*$<mact	show "active" list of mbuf's
mbstat	$<mbstat	show mbuf statistics
mbuf	*addr*$<mbuf	show "next" list of mbuf's
mbufs	*addr*$<mbufs	show a number of mbuf's
mount	*addr*$<mount	format mount structure
pcb	*addr*$<pcb	format process context block
proc	*addr*$<proc	format process table entry
protosw	*addr*$<protosw	format protocol table entry
rawcb	*addr*$<rawcb	format a raw protocol control block
rtentry	*addr*$<rtentry	format a routing table entry
rusage	*addr*$<rusage	format resource usage block
setproc	*pid*$<setproc	switch process context to *pid*

Standard Command Scripts		
Name	*Use*	*Description*
socket	*addr*$<socket	format socket structure
stat	*addr*$<stat	format stat structure
tcpcb	*addr*$<tcpcb	format TCP control block
tcpip	*addr*$<tcpip	format a TCP/IP packet header
tcpreass	*addr*$<tcpreass	show a TCP reassembly queue
text	*addr*$<text	format text structure
traceall	$<traceall	show stack trace for all processes
tty	*addr*$<tty	format tty structure
u	*addr*$<u	format user vector, including pcb
uio	*addr*$<uio	format uio structure
vtimes	*addr*$<vtimes	format vtimes structure

3. Generating ADB Scripts with Adbgen

You can use the *adbgen*(8) program to write the scripts presented earlier in a way that does not depend on the structure member offsets of the items being referenced. For example, the **text** script given above depended on the fact that all the members to be printed were located contiguously in memory. Using *adbgen*, we could write the script as follows (again it is really on one line, but broken apart here for ease of display):

```
#include "sys/types.h"
#include "sys/text.h"

text
./"daddr"n{x_daddr,12X}n\
"ptdaddr"16t"size"16t"caddr"16t"iptr"n\
{x_ptdaddr,X}{x_size,X}{x_caddr,X}{x_iptr,X}n\
"rssize"8t"swrss"8t"count"8t"ccount"8t"flag"8t"slptim"8t"poip"n\
{x_rssize,x}{x_swrss,x}{x_count,b}{x_ccount,b}{x_flag,b}{x_slptime,b}{x_poip,x}{END}
```

The script starts with the names of the relevant header files, while the braces delimit structure member names and their formats. This script is then processed through *adbgen*(8) to get the *adb* script presented in the previous section. See *adbgen*(8) for a complete description of how to write *adbgen* scripts. The real value of writing scripts this way becomes apparent only with longer and more complicated scripts (the u script for example). Once the scripts are written this way they can be rerun if a structure definition changes without any human effort put into offset calculations.

4. Summary

The extensions made to *adb* provide basic support for debugging the UNIX kernel by eliminating the need for a user to carry out virtual to physical address translation. A collection of scripts have been written to nicely format the major kernel data structures and aid in switching between process contexts. This has been carried out with only minimal changes to the debugger.

More work is also required on the user interface to ***adb***. It appears the inscrutable ***adb*** command language has limited widespread use of much of the power of ***adb***. One possibility is to provide a more comprehensible "adb frontend", just as ***bc***(1) is used as a front end for ***dc***(1). Another possibility is to upgrade ***dbx***(1) to understand the kernel.

The Sun UNIX File System

Contents

1. Introduction **2**

2. Old File System **3**

3. New file system organization **5**
- 3.1. Optimizing storage utilization 5
- 3.2. File system parameterization 8
- 3.3. Layout policies 9

4. Performance **11**

5. File system functional enhancements **14**
- 5.1. Long file names 14
- 5.2. File locking 14
- 5.3. Symbolic links 15
- 5.4. Rename 16
- 5.5. Quotas 16

6. Software engineering **17**

A. Acknowledgements **19**

B. References **19**

A Fast File System for UNIX

This document describes a reimplementation of the UNIX file system. The reimplementation provides substantially higher throughput rates by using more flexible allocation policies, that allow better locality of reference and that can be adapted to a wide range of peripheral and processor characteristics. The new file system clusters data that is sequentially accessed and provides two block sizes to allow fast access for large files while not wasting large amounts of space for small files. File access rates of up to ten times faster than the traditional UNIX file system are experienced. Long needed enhancements to the user interface are discussed. These include a mechanism to lock files, extensions of the name space across file systems, the ability to use arbitrary length file names, and provisions for efficient administrative control of resource usage.

1. Introduction

This paper describes the changes between the original 512 byte UNIX file system to the file system implemented with the first Berkeley-compatible release of Sun's version of the UNIX system. It presents the motivations for the changes, the methods used to affect these changes, the rationale behind the design decisions, and a description of the new implementation. This discussion is followed by a summary of the results that have been obtained, directions for future work, and the additions and changes that have been made to the user visible facilities. The paper concludes with a history of the software engineering of the project.

The original UNIX system that runs on the PDP-11[1] has simple and elegant file system facilities. File system input/output is buffered by the kernel; there are no alignment constraints on data transfers and all operations are made to appear synchronous. All transfers to the disk are in 512 byte blocks, which can be placed arbitrarily within the data area of the file system. No constraints other than available disk space are placed on file growth [Ritchie74], [Thompson79].

When used together with other UNIX enhancements, the original 512 byte UNIX file system is incapable of providing the data throughput rates that many applications require. For example, applications that need to do a small amount of processing on a large quantities of data such as VLSI design and image processing, need to have a high throughput from the file system. High throughput rates are also needed by programs with large address spaces that are constructed by mapping files from the file system into virtual memory. Paging data in and out of the file system is likely to occur frequently. This requires a file system providing higher bandwidth than the original 512 byte UNIX one which provides only about two percent of the maximum disk bandwidth or about 20 kilobytes per second per arm [White80], [Smith81b].

Modifications have been made to the UNIX file system to improve its performance. Since the UNIX file system interface is well understood and not inherently slow, this development retained the abstraction and simply changed the underlying implementation to increase its throughput. Consequently users of the system have not been faced with massive software conversion.

Problems with file system performance have been dealt with extensively in the literature; see [Smith81a] for a survey. The UNIX operating system drew many of its ideas from Multics, a large, high performance operating system [Feiertag71]. Other work includes Hydra [Almes78], Spice [Thompson80], and a file system for a lisp environment [Symbolics81a].

A major goal of this project has been to build a file system that is extensible into a networked environment [Holler73]. Other work on network file systems describe centralized file servers [Accetta80], distributed file servers [Dion80], [Luniewski77], [Porcar82], and protocols to reduce the amount of information that must be transferred across a network [Symbolics81b], [Sturgis80].

[1] DEC, PDP, VAX, MASSBUS, and UNIBUS are trademarks of Digital Equipment Corporation.

2. Old File System

In the old file system developed at Bell Laboratories each disk drive contains one or more file systems[2]. A file system is described by its super-block, which contains the basic parameters of the file system. These include the number of data blocks in the file system, a count of the maximum number of files, and a pointer to a list of free blocks. All the free blocks in the system are chained together in a linked list. Within the file system are files. Certain files are distinguished as directories and contain pointers to files that may themselves be directories. Every file has a descriptor associated with it called an ***inode***. The inode contains information describing ownership of the file, time stamps marking last modification and access times for the file, and an array of indices that point to the data blocks for the file. For the purposes of this section, we assume that the first 8 blocks of the file are directly referenced by values stored in the inode structure itself[3]. The inode structure may also contain references to indirect blocks containing further data block indices. In a file system with a 512 byte block size, a singly indirect block contains 128 further block addresses, a doubly indirect block contains 128 addresses of further single indirect blocks, and a triply indirect block contains 128 addresses of further doubly indirect blocks.

A traditional 150 megabyte UNIX file system consists of 4 megabytes of inodes followed by 146 megabytes of data. This organization segregates the inode information from the data; thus accessing a file normally incurs a long seek from its inode to its data. Files in a single directory are not typically allocated slots in consecutive locations in the 4 megabytes of inodes, causing many non-consecutive blocks to be accessed when executing operations on all the files in a directory.

The allocation of data blocks to files is also suboptimum. The traditional file system never transfers more than 512 bytes per disk transaction and often finds that the next sequential data block is not on the same cylinder, forcing seeks between 512 byte transfers. The combination of the small block size, limited read-ahead in the system, and many seeks severely limits file system throughput.

The first work at Berkeley on the UNIX file system attempted to improve both reliability and throughput. The reliability was improved by changing the file system so that all modifications of critical information were staged so that they could either be completed or repaired cleanly by a program after a crash [Kowalski78]. The file system performance was improved by a factor of more than two by changing the basic block size from 512 to 1024 bytes. The increase was because of two factors; each disk transfer accessed twice as much data, and most files could be described without need to access through any indirect blocks since the direct blocks contained twice as much data. The file system with these changes will henceforth be referred to as the ***old file system.***

This performance improvement gave a strong indication that increasing the block size was a good method for improving throughput. Although the throughput had doubled, the old file system was still using only about four percent of the disk bandwidth. The main problem was that although the free list was initially ordered for optimal access, it quickly became scrambled as files were created and removed. Eventually the free list became entirely random causing files to have their blocks allocated randomly over the disk. This forced the disk to seek before every block access. Although old file systems provided transfer rates of up to 175 kilobytes per second

[2] A file system always resides on a single drive.

[3] The actual number may vary from system to system, but is usually in the range 5-13.

when they were first created, this rate deteriorated to 30 kilobytes per second after a few weeks of moderate use because of randomization of their free block list. There was no way of restoring the performance an old file system except to dump, rebuild, and restore the file system. Another possibility would be to have a process that periodically reorganized the data on the disk to restore locality as suggested by [Maruyama76].

3. New file system organization

As in the old file system organization each disk drive contains one or more file systems. A file system is described by its super-block, that is located at the beginning of its disk partition. Because the super-block contains critical data it is replicated to protect against catastrophic loss. This is done at the time that the file system is created; since the super-block data does not change, the copies need not be referenced unless a head crash or other hard disk error causes the default super-block to be unusable.

To ensure that it is possible to create files as large as 2↑32 bytes with only two levels of indirection, the minimum size of a file system block is 4096 bytes. The size of file system blocks can be any power of two greater than or equal to 4096. The block size of the file system is maintained in the super-block so it is possible for file systems with different block sizes to be accessible simultaneously on the same system. The block size must be decided at the time that the file system is created; it cannot be subsequently changed without rebuilding the file system.

The new file system organization partitions the disk into one or more areas called ***cylinder groups***. A cylinder group is comprised of one or more consecutive cylinders on a disk. Associated with each cylinder group is some bookkeeping information that includes a redundant copy of the super-block, space for inodes, a bit map describing available blocks in the cylinder group, and summary information describing the usage of data blocks within the cylinder group. For each cylinder group a static number of inodes is allocated at file system creation time. The current policy is to allocate one inode for each 2048 bytes of disk space, expecting this to be far more than will ever be needed.

All the cylinder group bookkeeping information could be placed at the beginning of each cylinder group. However if this approach were used, all the redundant information would be on the top platter. Thus a single hardware failure that destroyed the top platter could cause the loss of all copies of the redundant super-blocks. Thus the cylinder group bookkeeping information begins at a floating offset from the beginning of the cylinder group. The offset for each successive cylinder group is calculated to be about one track further from the beginning of the cylinder group. In this way the redundant information spirals down into the pack so that any single track, cylinder, or platter can be lost without losing all copies of the super-blocks. Except for the first cylinder group, the space between the beginning of the cylinder group and the beginning of the cylinder group information is used for data blocks[4].

3.1. Optimizing storage utilization

Data is laid out so that larger blocks can be transferred in a single disk transfer, greatly increasing file system throughput. As an example, consider a file in the new file system composed of 4096 byte data blocks. In the old file system this file would be composed of 1024 byte blocks. By increasing the block size, disk accesses in the new file system may transfer up to four times as much information per disk transaction. In large files, several 4096 byte blocks may be allocated from the same cylinder so that even larger data transfers are possible before initiating a seek.

The main problem with bigger blocks is that most UNIX file systems are composed of many small files. A uniformly large block size wastes space. Table 1 shows the effect of file system

[4] While it appears that the first cylinder group could be laid out with its super-block at the "known" location, this would not work for file systems with blocks sizes of 16K or greater, because of the requirement that the cylinder group information must begin at a block boundary.

block size on the amount of wasted space in the file system. The machine measured to obtain these figures is one of our time sharing systems that has roughly 1.2 Gigabyte of on-line storage. The measurements are based on the active user file systems containing about 920 megabytes of formated space.

Table 1: Wasted Space as a function of Block Size

Space used	% waste	Organization
775.2 Mb	0.0	Data only, no separation between files
807.8 Mb	4.2	Data only, each file starts on 512 byte boundary
828.7 Mb	6.9	512 byte block UNIX file system
866.5 Mb	11.8	1024 byte block UNIX file system
948.5 Mb	22.4	2048 byte block UNIX file system
1128.3 Mb	45.6	4096 byte block UNIX file system

The space wasted is measured as the percentage of space on the disk not containing user data. As the block size on the disk increases, the waste rises quickly, to an intolerable 45.6% waste with 4096 byte file system blocks.

To be able to use large blocks without undue waste, small files must be stored in a more efficient way. The new file system accomplishes this goal by allowing the division of a single file system block into one or more ***fragments***. The file system fragment size is specified at the time that the file system is created; each file system block can be optionally broken into 2, 4, or 8 fragments, each of which is addressable. The lower bound on the size of these fragments is constrained by the disk sector size, typically 512 bytes. The block map associated with each cylinder group records the space availability at the fragment level; to determine block availability, aligned fragments are examined. Figure 1 shows a piece of a map from a 4096/1024 file system.

Bits in map	XXXX	XXOO	OOXX	OOOO
Fragment numbers	0-3	4-7	8-11	12-15
Block numbers	0	1	2	3

Figure 1: Example layout of blocks and fragments in a 4096/1024 file system

Each bit in the map records the status of a fragment; an "X" shows that the fragment is in use, while a "O" shows that the fragment is available for allocation. In this example, fragments 0–5, 10, and 11 are in use, while fragments 6–9, and 12–15 are free. Fragments of adjoining blocks cannot be used as a block, even if they are large enough. In this example, fragments 6–9 cannot be coalesced into a block; only fragments 12–15 are available for allocation as a block.

On a file system with a block size of 4096 bytes and a fragment size of 1024 bytes, a file is represented by zero or more 4096 byte blocks of data, and possibly a single fragmented block. If a file system block must be fragmented to obtain space for a small amount of data, the remainder of the block is made available for allocation to other files. As an example consider an 11000 byte file stored on a 4096/1024 byte file system. This file would uses two full size blocks and a 3072 byte fragment. If no 3072 byte fragments are available at the time the file is created, a full size block is split yielding the necessary 3072 byte fragment and an unused 1024 byte fragment. This remaining fragment can be allocated to another file as needed.

The granularity of allocation is the *write* system call. Each time data is written to a file, the system checks to see if the size of the file has increased[5]. If the file needs to hold the new data, one of three conditions exists:

1) There is enough space left in an already allocated block to hold the new data. The new data is written into the available space in the block.

2) Nothing has been allocated. If the new data contains more than 4096 bytes, a 4096 byte block is allocated and the first 4096 bytes of new data is written there. This process is repeated until less than 4096 bytes of new data remain. If the remaining new data to be written will fit in three or fewer 1024 byte pieces, an unallocated fragment is located, otherwise a 4096 byte block is located. The new data is written into the located piece.

3) A fragment has been allocated. If the number of bytes in the new data plus the number of bytes already in the fragment exceeds 4096 bytes, a 4096 byte block is allocated. The contents of the fragment is copied to the beginning of the block and the remainder of the block is filled with the new data. The process then continues as in (2) above. If the number of bytes in the new data plus the number of bytes already in the fragment will fit in three or fewer 1024 byte pieces, an unallocated fragment is located, otherwise a 4096 byte block is located. The contents of the previous fragment appended with the new data is written into the allocated piece.

The problem with allowing only a single fragment on a 4096/1024 byte file system is that data may be potentially copied up to three times as its requirements grow from a 1024 byte fragment to a 2048 byte fragment, then a 3072 byte fragment, and finally a 4096 byte block. The fragment reallocation can be avoided if the user program writes a full block at a time, except for a partial block at the end of the file. Because file systems with different block sizes may coexist on the same system, the file system interface been extended to provide the ability to determine the optimal size for a read or write. For files the optimal size is the block size of the file system on which the file is being accessed. For other objects, such as pipes and sockets, the optimal size is the underlying buffer size. This feature is used by the Standard Input/Output Library, a package used by most user programs. This feature is also used by certain system utilities such as archivers and loaders that do their own input and output management and need the highest possible file system bandwidth.

The space overhead in the 4096/1024 byte new file system organization is empirically observed to be about the same as in the 1024 byte old file system organization. A file system with 4096 byte blocks and 512 byte fragments has about the same amount of space overhead as the 512 byte block UNIX file system. The new file system is more space efficient than the 512 byte or 1024 byte file systems in that it uses the same amount of space for small files while requiring less indexing information for large files. This savings is offset by the need to use more space for keeping track of available free blocks. The net result is about the same disk utilization when the new file systems fragment size equals the old file systems block size.

In order for the layout policies to be effective, the disk cannot be kept completely full. Each file system maintains a parameter that gives the minimum acceptable percentage of file system blocks that can be free. If the the number of free blocks drops below this level only the system administrator can continue to allocate blocks. The value of this parameter can be changed at any time, even when the file system is mounted and active. The transfer rates to be given in section 4 were measured on file systems kept less than 90% full. If the reserve of free blocks is

[5] A program may be overwriting data in the middle of an existing file in which case space will already be allocated.

set to zero, the file system throughput rate tends to be cut in half, because of the inability of the file system to localize the blocks in a file. If the performance is impaired because of overfilling, it may be restored by removing enough files to obtain 10% free space. Access speed for files created during periods of little free space can be restored by recreating them once enough space is available. The amount of free space maintained must be added to the percentage of waste when comparing the organizations given in Table 1. Thus, a site running the old 1024 byte UNIX file system wastes 11.8% of the space and one could expect to fit the same amount of data into a 4096/512 byte new file system with 5% free space, since a 512 byte old file system wasted 6.9% of the space.

3.2. File system parameterization

Except for the initial creation of the free list, the old file system ignores the parameters of the underlying hardware. It has no information about either the physical characteristics of the mass storage device, or the hardware that interacts with it. A goal of the new file system is to parameterize the processor capabilities and mass storage characteristics so that blocks can be allocated in an optimum configuration dependent way. Parameters used include the speed of the processor, the hardware support for mass storage transfers, and the characteristics of the mass storage devices. Disk technology is constantly improving and a given installation can have several different disk technologies running on a single processor. Each file system is parameterized so that it can adapt to the characteristics of the disk on which it is placed.

For mass storage devices such as disks, the new file system tries to allocate new blocks on the same cylinder as the previous block in the same file. Optimally, these new blocks will also be well positioned rotationally. The distance between "rotationally optimal" blocks varies greatly; it can be a consecutive block or a rotationally delayed block depending on system characteristics. On a processor with a channel that does not require any processor intervention between mass storage transfer requests, two consecutive disk blocks often can be accessed without suffering lost time because of an intervening disk revolution. For processors without such channels, the main processor must field an interrupt and prepare for a new disk transfer. The expected time to service this interrupt and schedule a new disk transfer depends on the speed of the main processor.

The physical characteristics of each disk include the number of blocks per track and the rate at which the disk spins. The allocation policy routines use this information to calculate the number of milliseconds required to skip over a block. The characteristics of the processor include the expected time to schedule an interrupt. Given the previous block allocated to a file, the allocation routines calculate the number of blocks to skip over so that the next block in a file will be coming into position under the disk head in the expected amount of time that it takes to start a new disk transfer operation. For programs that sequentially access large amounts of data, this strategy minimizes the amount of time spent waiting for the disk to position itself.

To ease the calculation of finding rotationally optimal blocks, the cylinder group summary information includes a count of the availability of blocks at different rotational positions. Eight rotational positions are distinguished, so the resolution of the summary information is 2 milliseconds for a typical 3600 revolution per minute drive.

The parameter that defines the minimum number of milliseconds between the completion of a data transfer and the initiation of another data transfer on the same cylinder can be changed at any time, even when the file system is mounted and active. If a file system is parameterized to lay out blocks with rotational separation of 2 milliseconds, and the disk pack is then moved to a system that has a processor requiring 4 milliseconds to schedule a disk operation, the throughput will drop precipitously because of lost disk revolutions on nearly every block. If the eventual

target machine is known, the file system can be parameterized for it even though it is initially created on a different processor. Even if the move is not known in advance, the rotational layout delay can be reconfigured after the disk is moved so that all further allocation is done based on the characteristics of the new host.

3.3. Layout policies

The file system policies are divided into two distinct parts. At the top level are global policies that use file system wide summary information to make decisions regarding the placement of new inodes and data blocks. These routines are responsible for deciding the placement of new directories and files. They also calculate rotationally optimal block layouts, and decide when to force a long seek to a new cylinder group because there are insufficient blocks left in the current cylinder group to do reasonable layouts. Below the global policy routines are the local allocation routines that use a locally optimal scheme to lay out data blocks.

Two methods for improving file system performance are to increase the locality of reference to minimize seek latency as described by [Trivedi80], and to improve the layout of data to make larger transfers possible as described by [Nevalainen77]. The global layout policies try to improve performance by clustering related information. They cannot attempt to localize all data references, but must also try to spread unrelated data among different cylinder groups. If too much localization is attempted, the local cylinder group may run out of space forcing the data to be scattered to non-local cylinder groups. Taken to an extreme, total localization can result in a single huge cluster of data resembling the old file system. The global policies try to balance the two conflicting goals of localizing data that is concurrently accessed while spreading out unrelated data.

One allocatable resource is inodes. Inodes are used to describe both files and directories. Files in a directory are frequently accessed together. For example the "list directory" command often accesses the inode for each file in a directory. The layout policy tries to place all the files in a directory in the same cylinder group. To ensure that files are allocated throughout the disk, a different policy is used for directory allocation. A new directory is placed in the cylinder group that has a greater than average number of free inodes, and the fewest number of directories in it already. The intent of this policy is to allow the file clustering policy to succeed most of the time. The allocation of inodes within a cylinder group is done using a next free strategy. Although this allocates the inodes randomly within a cylinder group, all the inodes for each cylinder group can be read with 4 to 8 disk transfers. This puts a small and constant upper bound on the number of disk transfers required to access all the inodes for all the files in a directory as compared to the old file system where typically, one disk transfer is needed to get the inode for each file in a directory.

The other major resource is the data blocks. Since data blocks for a file are typically accessed together, the policy routines try to place all the data blocks for a file in the same cylinder group, preferably rotationally optimally on the same cylinder. The problem with allocating all the data blocks in the same cylinder group is that large files will quickly use up available space in the cylinder group, forcing a spill over to other areas. Using up all the space in a cylinder group has the added drawback that future allocations for any file in the cylinder group will also spill to other areas. Ideally none of the cylinder groups should ever become completely full. The solution devised is to redirect block allocation to a newly chosen cylinder group when a file exceeds 32 kilobytes, and at every megabyte thereafter. The newly chosen cylinder group is selected from those cylinder groups that have a greater than average number of free blocks left. Although big files tend to be spread out over the disk, a megabyte of data is typically accessible

before a long seek must be performed, and the cost of one long seek per megabyte is small.

The global policy routines call local allocation routines with requests for specific blocks. The local allocation routines will always allocate the requested block if it is free. If the requested block is not available, the allocator allocates a free block of the requested size that is rotationally closest to the requested block. If the global layout policies had complete information, they could always request unused blocks and the allocation routines would be reduced to simple bookkeeping. However, maintaining complete information is costly; thus the implementation of the global layout policy uses heuristic guesses based on partial information.

If a requested block is not available the local allocator uses a four level allocation strategy:

1) Use the available block rotationally closest to the requested block on the same cylinder.

2) If there are no blocks available on the same cylinder, use a block within the same cylinder group.

3) If the cylinder group is entirely full, quadratically rehash among the cylinder groups looking for a free block.

4) Finally if the rehash fails, apply an exhaustive search.

The use of quadratic rehash is prompted by studies of symbol table strategies used in programming languages. File systems that are parameterized to maintain at least 10% free space almost never use this strategy; file systems that are run without maintaining any free space typically have so few free blocks that almost any allocation is random. Consequently the most important characteristic of the strategy used when the file system is low on space is that it be fast.

4. Performance

Ultimately, the proof of the effectiveness of the algorithms described in the previous section is the long term performance of the new file system.

Our empiric studies have shown that the inode layout policy has been effective. When running the "list directory" command on a large directory that itself contains many directories, the number of disk accesses for inodes is cut by a factor of two. The improvements are even more dramatic for large directories containing only files, disk accesses for inodes being cut by a factor of eight. This is most encouraging for programs such as spooling daemons that access many small files, since these programs tend to flood the disk request queue on the old file system.

Table 2 summarizes the measured throughput of the new file system. Several comments need to be made about the conditions under which these tests were run. The test programs measure the rate that user programs can transfer data to or from a file without performing any processing on it. These programs must write enough data to ensure that buffering in the operating system does not affect the results. They should also be run at least three times in succession; the first to get the system into a known state and the second two to ensure that the experiment has stabilized and is repeatable. The methodology and test results are discussed in detail in [Kridle83][6]. The systems were running multi-user but were otherwise quiescent. There was no contention for either the cpu or the disk arm. The only difference between the UNIBUS and MASSBUS tests was the controller. All tests used an Ampex Capricorn 330 Megabyte Winchester disk. As Table 2 shows, all file system test runs were on a VAX 11/750. All file systems had been in production use for at least a month before being measured.

Table 2: Reading Rates of the Old and New UNIX File Systems

Type of File System	Processor and Bus Measured	Speed	Read Bandwidth	% CPU
old 1024	750/UNIBUS	29 Kbytes/sec	29/1100 3%	11%
new 4096/1024	750/UNIBUS	221 Kbytes/sec	221/1100 20%	43%
new 8192/1024	750/UNIBUS	233 Kbytes/sec	233/1100 21%	29%
new 4096/1024	750/MASSBUS	466 Kbytes/sec	466/1200 39%	73%
new 8192/1024	750/MASSBUS	466 Kbytes/sec	466/1200 39%	54%

[6] A UNIX command that is similar to the reading test that we used is, "cp file /dev/null", where "file" is eight Megabytes long.

Table 3: Writing rates of the old and new UNIX file systems

Type of File System	Processor and Bus Measured	Speed	Write Bandwidth	% CPU
old 1024	750/UNIBUS	48 Kbytes/sec	48/1100 4%	29%
new 4096/1024	750/UNIBUS	142 Kbytes/sec	142/1100 13%	43%
new 8192/1024	750/UNIBUS	215 Kbytes/sec	215/1100 19%	46%
new 4096/1024	750/MASSBUS	323 Kbytes/sec	323/1200 27%	94%
new 8192/1024	750/MASSBUS	466 Kbytes/sec	466/1200 39%	95%

Unlike the old file system, the transfer rates for the new file system do not appear to change over time. The throughput rate is tied much more strongly to the amount of free space that is maintained. The measurements in Table 2 were based on a file system run with 10% free space. Synthetic work loads suggest the performance deteriorates to about half the throughput rates given in Table 2 when no free space is maintained.

The percentage of bandwidth given in Table 2 is a measure of the effective utilization of the disk by the file system. An upper bound on the transfer rate from the disk is measured by doing 65536[7] byte reads from contiguous tracks on the disk. The bandwidth is calculated by comparing the data rates the file system is able to achieve as a percentage of this rate. Using this metric, the old file system is only able to use about 3-4% of the disk bandwidth, while the new file system uses up to 39% of the bandwidth.

In the new file system, the reading rate is always at least as fast as the writing rate. This is to be expected since the kernel must do more work when allocating blocks than when simply reading them. Note that the write rates are about the same as the read rates in the 8192 byte block file system; the write rates are slower than the read rates in the 4096 byte block file system. The slower write rates occur because the kernel has to do twice as many disk allocations per second, and the processor is unable to keep up with the disk transfer rate.

In contrast the old file system is about 50% faster at writing files than reading them. This is because the *write* system call is asynchronous and the kernel can generate disk transfer requests much faster than they can be serviced, hence disk transfers build up in the disk buffer cache. Because the disk buffer cache is sorted by minimum seek order, the average seek between the scheduled disk writes is much less than they would be if the data blocks are written out in the order in which they are generated. However when the file is read, the *read* system call is processed synchronously so the disk blocks must be retrieved from the disk in the order in which they are allocated. This forces the disk scheduler to do long seeks resulting in a lower throughput rate.

The performance of the new file system is currently limited by a memory to memory copy operation because it transfers data from the disk into buffers in the kernel address space and then spends 40% of the processor cycles copying these buffers to user address space. If the buffers in both address spaces are properly aligned, this transfer can be affected without copying by using the VAX virtual memory management hardware. This is especially desirable when large amounts of data are to be transferred. We did not implement this because it would change the

[7] This number, 65536, is the maximal I/O size supported by the VAX hardware; it is a remnant of the system's PDP-11 ancestry.

semantics of the file system in two major ways; user programs would be required to allocate buffers on page boundaries, and data would disappear from buffers after being written.

Greater disk throughput could be achieved by rewriting the disk drivers to chain together kernel buffers. This would allow files to be allocated to contiguous disk blocks that could be read in a single disk transaction. Most disks contain either 32 or 48 512 byte sectors per track. The inability to use contiguous disk blocks effectively limits the performance on these disks to less than fifty percent of the available bandwidth. Since each track has a multiple of sixteen sectors it holds exactly two or three 8192 byte file system blocks, or four or six 4096 byte file system blocks. If the the next block for a file cannot be laid out contiguously, then the minimum spacing to the next allocatable block on any platter is between a sixth and a half a revolution. The implication of this is that the best possible layout without contiguous blocks uses only half of the bandwidth of any given track. If each track contains an odd number of sectors, then it is possible to resolve the rotational delay to any number of sectors by finding a block that begins at the desired rotational position on another track. The reason that block chaining has not been implemented is because it would require rewriting all the disk drivers in the system, and the current throughput rates are already limited by the speed of the available processors.

Currently only one block is allocated to a file at a time. A technique used by the DEMOS file system when it finds that a file is growing rapidly, is to preallocate several blocks at once, releasing them when the file is closed if they remain unused. By batching up the allocation the system can reduce the overhead of allocating at each write, and it can cut down on the number of disk writes needed to keep the block pointers on the disk synchronized with the block allocation [Powell79].

5. File system functional enhancements

The speed enhancements to the UNIX file system did not require any changes to the semantics or data structures viewed by the users. However several changes have been generally desired for some time but have not been introduced because they would require users to dump and restore all their file systems. Since the new file system already requires that all existing file systems be dumped and restored, these functional enhancements have been introduced at this time.

5.1. Long file names

File names can now be of nearly arbitrary length. The only user programs affected by this change are those that access directories. To maintain portability among UNIX systems that are not running the new file system, a set of directory access routines have been introduced that provide a uniform interface to directories on both old and new systems.

Directories are allocated in units of 512 bytes. This size is chosen so that each allocation can be transferred to disk in a single atomic operation. Each allocation unit contains variable-length directory entries. Each entry is wholly contained in a single allocation unit. The first three fields of a directory entry are fixed and contain an inode number, the length of the entry, and the length of the name contained in the entry. Following this fixed size information is the null terminated name, padded to a 4 byte boundary. The maximum length of a name in a directory is currently 255 characters.

Free space in a directory is held by entries that have a record length that exceeds the space required by the directory entry itself. All the bytes in a directory unit are claimed by the directory entries. This normally results in the last entry in a directory being large. When entries are deleted from a directory, the space is returned to the previous entry in the same directory unit by increasing its length. If the first entry of a directory unit is free, then its inode number is set to zero to show that it is unallocated.

5.2. File locking

The old file system had no provision for locking files. Processes that needed to synchronize the updates of a file had to create a separate "lock" file to synchronize their updates. A process would try to create a "lock" file. If the creation succeeded, then it could proceed with its update; if the creation failed, then it would wait, and try again. This mechanism had three drawbacks. Processes consumed CPU time, by looping over attempts to create locks. Locks were left lying around following system crashes and had to be cleaned up by hand. Finally, processes running as system administrator are always permitted to create files, so they had to use a different mechanism. While it is possible to get around all these problems, the solutions are not straight-forward, so a mechanism for locking files has been added.

The most general schemes allow processes to concurrently update a file. Several of these techniques are discussed in [Peterson83]. A simpler technique is to simply serialize access with locks. To attain reasonable efficiency, certain applications require the ability to lock pieces of a file. Locking down to the byte level has been implemented in the Onyx file system by [Bass81]. However, for the applications that currently run on the system, a mechanism that locks at the granularity of a file is sufficient.

Locking schemes fall into two classes, those using hard locks and those using advisory locks. The primary difference between advisory locks and hard locks is the decision of when to override them. A hard lock is always enforced whenever a program tries to access a file; an advisory lock is only applied when it is requested by a program. Thus advisory locks are only effective when all programs accessing a file use the locking scheme. With hard locks there must be some override policy implemented in the kernel, with advisory locks the policy is implemented by the user programs. In the UNIX system, programs with system administrator privilege can override any protection scheme. Because many of the programs that need to use locks run as system administrators, we chose to implement advisory locks rather than create a protection scheme that was contrary to the UNIX philosophy or could not be used by system administration programs.

The file locking facilities allow cooperating programs to apply advisory ***shared*** or ***exclusive*** locks on files. Only one process has an exclusive lock on a file while multiple shared locks may be present. Both shared and exclusive locks cannot be present on a file at the same time. If any lock is requested when another process holds an exclusive lock, or an exclusive lock is requested when another process holds any lock, the open will block until the lock can be gained. Because shared and exclusive locks are advisory only, even if a process has obtained a lock on a file, another process can override the lock by opening the same file without a lock.

Locks can be applied or removed on open files, so that locks can be manipulated without needing to close and reopen the file. This is useful, for example, when a process wishes to open a file with a shared lock to read some information, to determine whether an update is required. It can then get an exclusive lock so that it can do a read, modify, and write to update the file in a consistent manner.

A request for a lock will cause the process to block if the lock can not be immediately obtained. In certain instances this is unsatisfactory. For example, a process that wants only to check if a lock is present would require a separate mechanism to find out this information. Consequently, a process may specify that its locking request should return with an error if a lock can not be immediately obtained. Being able to poll for a lock is useful to "daemon" processes that wish to service a spooling area. If the first instance of the daemon locks the directory where spooling takes place, later daemon processes can easily check to see if an active daemon exists. Since the lock is removed when the process exits or the system crashes, there is no problem with unintentional locks files that must be cleared by hand.

Almost no deadlock detection is attempted. The only deadlock detection made by the system is that the file descriptor to which a lock is applied does not currently have a lock of the same type (i.e. the second of two successive calls to apply a lock of the same type will fail). Thus a process can deadlock itself by requesting locks on two separate file descriptors for the same object.

5.3. Symbolic links

The 512 byte UNIX file system allows multiple directory entries in the same file system to reference a single file. The link concept is fundamental; files do not live in directories, but exist separately and are referenced by links. When all the links are removed, the file is deallocated. This style of links does not allow references across physical file systems, nor does it support inter-machine linkage. To avoid these limitations ***symbolic links*** have been added similar to the scheme used by Multics [Feiertag71].

A symbolic link is implemented as a file that contains a pathname. When the system encounters a symbolic link while interpreting a component of a pathname, the contents of the symbolic link is prepended to the rest of the pathname, and this name is interpreted to yield the resulting

pathname. If the symbolic link contains an absolute pathname, the absolute pathname is used, otherwise the contents of the symbolic link is evaluated relative to the location of the link in the file hierarchy.

Normally programs do not want to be aware that there is a symbolic link in a pathname that they are using. However certain system utilities must be able to detect and manipulate symbolic links. Three new system calls provide the ability to detect, read, and write symbolic links, and seven system utilities were modified to use these calls.

In future Berkeley software distributions it will be possible to mount file systems from other machines within a local file system. When this occurs, it will be possible to create symbolic links that span machines.

5.4. Rename

Programs that create new versions of data files typically create the new version as a temporary file and then rename the temporary file with the original name of the data file. In the old UNIX file systems the renaming required three calls to the system. If the program were interrupted or the system crashed between these calls, the data file could be left with only its temporary name. To eliminate this possibility a single system call has been added that performs the rename in an atomic fashion to guarantee the existence of the original name.

In addition, the rename facility allows directories to be moved around in the directory tree hierarchy. The rename system call performs special validation checks to ensure that the directory tree structure is not corrupted by the creation of loops or inaccessible directories. Such corruption would occur if a parent directory were moved into one of its descendants. The validation check requires tracing the ancestry of the target directory to ensure that it does not include the directory being moved.

5.5. Quotas

The UNIX system has traditionally attempted to share all available resources to the greatest extent possible. Thus any single user can allocate all the available space in the file system. In certain environments this is unacceptable. Consequently, a quota mechanism has been added for restricting the amount of file system resources that a user can obtain. The quota mechanism sets limits on both the number of files and the number of disk blocks that a user may allocate. A separate quota can be set for each user on each file system. Each resource is given both a hard and a soft limit. When a program exceeds a soft limit, a warning is printed on the users terminal; the offending program is not terminated unless it exceeds its hard limit. The idea is that users should stay below their soft limit between login sessions, but they may use more space while they are actively working. To encourage this behavior, users are warned when logging in if they are over any of their soft limits. If they fail to correct the problem for too many login sessions, they are eventually reprimanded by having their soft limit enforced as their hard limit.

6. Software engineering

The preliminary design was done by Bill Joy in late 1980; he presented the design at The USENIX Conference held in San Francisco in January 1981. The implementation of his design was done by Kirk McKusick in the summer of 1981. Most of the new system calls were implemented by Sam Leffler. The code for enforcing quotas was implemented by Robert Elz at the University of Melbourne.

To understand how the project was done it is necessary to understand the interfaces that the UNIX system provides to the hardware mass storage systems. At the lowest level is a ***raw disk.*** This interface provides access to the disk as a linear array of sectors. Normally this interface is only used by programs that need to do disk to disk copies or that wish to dump file systems. However, user programs with proper access rights can also access this interface. A disk is usually formated with a file system that is interpreted by the UNIX system to provide a directory hierarchy and files. The UNIX system interprets and multiplexes requests from user programs to create, read, write, and delete files by allocating and freeing inodes and data blocks. The interpretation of the data on the disk could be done by the user programs themselves. The reason that it is done by the UNIX system is to synchronize the user requests, so that two processes do not attempt to allocate or modify the same resource simultaneously. It also allows access to be restricted at the file level rather than at the disk level and allows the common file system routines to be shared between processes.

The implementation of the new file system amounted to using a different scheme for formating and interpreting the disk. Since the synchronization and disk access routines themselves were not being changed, the changes to the file system could be developed by moving the file system interpretation routines out of the kernel and into a user program. Thus, the first step was to extract the file system code for the old file system from the UNIX kernel and change its requests to the disk driver to accesses to a raw disk. This produced a library of routines that mapped what would normally be system calls into read or write operations on the raw disk. This library was then debugged by linking it into the system utilities that copy, remove, archive, and restore files.

A new cross file system utility was written that copied files from the simulated file system to the one implemented by the kernel. This was accomplished by calling the simulation library to do a read, and then writing the resultant data by using the conventional write system call. A similar utility copied data from the kernel to the simulated file system by doing a conventional read system call and then writing the resultant data using the simulated file system library.

The second step was to rewrite the file system simulation library to interpret the new file system. By linking the new simulation library into the cross file system copying utility, it was possible to easily copy files from the old file system into the new one and from the new one to the old one. Having the file system interpretation implemented in user code had several major benefits. These included being able to use the standard system tools such as the debuggers to set breakpoints and single step through the code. When bugs were discovered, the offending problem could be fixed and tested without the need to reboot the machine. There was never a period where it was necessary to maintain two concurrent file systems in the kernel. Finally it was not necessary to dedicate a machine entirely to file system development, except for a brief period while the new file system was boot strapped.

The final step was to merge the new file system back into the UNIX kernel. This was done in less than two weeks, since the only bugs remaining were those that involved interfacing to the synchronization routines that could not be tested in the simulated system. Again the simulation system proved useful since it enabled files to be easily copied between old and new file systems

regardless of which file system was running in the kernel. This greatly reduced the number of times that the system had to be rebooted.

The total design and debug time took about one man year. Most of the work was done on the file system utilities, and changing all the user programs to use the new facilities. The code changes in the kernel were minor, involving the addition of only about 800 lines of code.

Appendix A. Acknowledgements

We thank Robert Elz for his ongoing interest in the new file system, and for adding disk quotas in a rational and efficient manner. We also acknowledge Dennis Ritchie for his suggestions on the appropriate modifications to the user interface. We appreciate Michael Powell's explanations on how the DEMOS file system worked; many of his ideas were used in this implementation. Special commendation goes to Peter Kessler and Robert Henry for acting like real users during the early debugging stage when files were less stable than they should have been. Finally we thank our sponsors, the National Science Foundation under grant MCS80-05144, and the Defense Advance Research Projects Agency (DoD) under Arpa Order No. 4031 monitored by Naval Electronic System Command under Contract No. N00039-82-C-0235.

Appendix B. References

[Accetta80] Accetta, M., Robertson, G., Satyanarayanan, M., and Thompson, M. "The Design of a Network-Based Central File System", Carnegie-Mellon University, Dept of Computer Science Tech Report, #CMU-CS-80-134

[Almes78] Almes, G., and Robertson, G. "An Extensible File System for Hydra" Proceedings of the Third International Conference on Software Engineering, IEEE, May 1978.

[Bass81] Bass, J. "Implementation Description for File Locking", Onyx Systems Inc, 73 E. Trimble Rd, San Jose, CA 95131 Jan 1981.

[Dion80] Dion, J. "The Cambridge File Server", Operating Systems Review, 14, 4. Oct 1980. pp 26-35

[Eswaran74] Eswaran, K. "Placement of records in a file and file allocation in a computer network", Proceedings IFIPS, 1974. pp 304-307

[Holler73] Holler, J. "Files in Computer Networks", First European Workshop on Computer Networks, April 1973. pp 381-396

[Feiertag71] Feiertag, R. J. and Organick, E. I., "The Multics Input-Output System", Proceedings of the Third Symposium on Operating Systems Principles, ACM, Oct 1971. pp 35-41

[Kridle83] Kridle, R., and McKusick, M., "Performance Effects of Disk Subsystem Choices for VAX Systems Running 4.2BSD UNIX", Computer Systems Research Group, Dept of EECS, Berkeley, CA 94720, Technical Report #8.

[Kowalski78] Kowalski, T. "FSCK - The UNIX System Check Program", Bell Laboratory, Murray Hill, NJ 07974. March 1978

[Luniewski77] Luniewski, A. "File Allocation in a Distributed System", MIT Laboratory for Computer Science, Dec 1977.

[Maruyama76] Maruyama, K., and Smith, S. "Optimal reorganization of Distributed Space Disk Files", Communications of the ACM, 19, 11. Nov 1976. pp 634-642

[Nevalainen77] Nevalainen, O., Vesterinen, M. "Determining Blocking Factors for Sequential Files by Heuristic Methods", The Computer Journal, 20, 3. Aug 1977. pp 245-247

[Peterson83] Peterson, G. "Concurrent Reading While Writing", ACM Transactions on Programming Languages and Systems, ACM, 5, 1. Jan 1983. pp 46-55

[Powell79] Powell, M. "The DEMOS File System", Proceedings of the Sixth Symposium on Operating Systems Principles, ACM, Nov 1977. pp 33-42

[Porcar82] Porcar, J. "File Migration in Distributed Computer Systems", Ph.D. Thesis, Lawrence Berkeley Laboratory Tech Report #LBL-14763.

[Ritchie74] Ritchie, D. M. and Thompson, K., "The UNIX Time-Sharing System", CACM 17, 7. July 1974. pp 365-375

[Smith81a] Smith, A. "Input/Output Optimization and Disk Architectures: A Survey", Performance and Evaluation 1. Jan 1981. pp 104-117

[Smith81b] Smith, A. "Bibliography on File and I/O System Optimization and Related Topics", Operating Systems Review, 15, 4. Oct 1981. pp 39-54

[Sturgis80] Sturgis, H., Mitchell, J., and Israel, J. "Issues in the Design and Use of a Distributed File System", Operating Systems Review, 14, 3. pp 55-79

[Symbolics81a] "Symbolics File System", Symbolics Inc, 9600 DeSoto Ave, Chatsworth, CA 91311 Aug 1981.

[Symbolics81b] "Chaosnet FILE Protocol". Symbolics Inc, 9600 DeSoto Ave, Chatsworth, CA 91311 Sept 1981.

[Thompson79] Thompson, K. "UNIX Implementation", Section 31, Volume 2B, UNIX Programmers Manual, Bell Laboratory, Murray Hill, NJ 07974. Jan 1979

[Thompson80] Thompson, M. "Spice File System", Carnegie-Mellon University, Dept of Computer Science Tech Report, #CMU-CS-80-???

[Trivedi80] Trivedi, K. "Optimal Selection of CPU Speed, Device Capabilities, and File Assignments", Journal of the ACM, 27, 3. July 1980. pp 457-473

[White80] White, R. M. "Disk Storage Technology", Scientific American, 243(2), August 1980.

Sun Workstation CPU PROM Monitor

Contents

1. Command Syntax **1**

2. Syntax for Memory and Register Access **1**

3. Command Descriptions **3**

The CPU PROM Monitor Commands

The central processor board (CPU) of the Sun Workstation has a set of PROM's containing a program generally known as the 'monitor'. The monitor controls the operation of the system before the UNIX kernel takes control. This document describes the PROM monitor commands. For information on the startup and boot functions of the monitor, including messages displayed, see the appendix on ***Automatic Startup*** in the ***System Manager's Manual for the Sun Workstation.***

1. Command Syntax

The monitor understands commands in quite a simple format. The format is:

< ***verb*** >< ***space*** >*[< ***argument*** >]< ***return*** >

< ***verb*** > is always one alphabetic character; case does not matter.

< ***space*** >* means that any number of spaces is skipped here.

< ***argument*** >
is normally a hexadecimal number or a single letter; again, case does not matter. Square brackets '[ ]' indicate that the argument portion is optional.

< ***return*** > means that you should press the carriage-return key.

When typing commands, < ***backspace*** > and < ***delete*** > (also called < ***rubout*** >, generated by the key labelled < ***backtab*** > on the non-VT100 Sun keyboard) erase one character; control-U erases the entire line.

2. Syntax for Memory and Register Access

Several of the commands ***open*** a memory location, map register, or processor register, so that you can examine and/or modify the contents of the specified location. These commands include a, d, e, l, m, o, p, and r.

Each of these commands takes the form of a command letter, possibly followed by a hexadecimal memory address or register number, followed by a sequence of zero or more 'action specifier' arguments. The various options are illustrated below, using the **e** command as an example. You type the parts as shown in **bold typewriter font**, with a < ***return*** > at the end of each command.

If no action specifier arguments are present, the address or register name is displayed along with its current contents. You may then type a new hexadecimal value, or simply < ***return*** > to go on the next address or register. Typing any non-hex character and < ***return*** > gets you back to

command level. For registers, 'next' means within the sequence of registers:

D0-D7 the data registers,
A0-A6 the address registers,
SS the system stack pointer,
US the user stack pointer,
SF the source function code register,
DF the destination function code register,
VB the vector base register,
SC the system context register,
UC the user context register,
SR the status register,
PC the program counter.

For example, the following command sets consecutive locations 0x1234 and 0x1236 to the values 0x5678 and 0x0000 respectively:

```
> e1234
001234: 007F? 5678
001236: 51A4? 0
001238: C022? q
>
```

A non-hex character (such as question mark) on the command line means read-only:

```
> e1000 ?
001000: 007F
>
```

Multiple nonhex characters read multiple locations:

```
> e1000 ???
001000: 007F
001002: 0064
001004: 1234
>
```

A hex number on the command line does store-only:

```
> e1000 4567
001000 -> 4567
>
```

Multiple hex writes multiple locations:

```
> e1000 1 2 3
001000 -> 0001
001002 -> 0002
001004 -> 0003
>
```

Nonhex followed by hex reads, then stores.

```
> e1000 ? 346
001000: 007F -> 0346
>
```

Finally, reads and writes can be interspersed:

```
> e1000 ? 1 ? ? 3 4
001000: 007F -> 0001
001002: 0064
001004: 1234 -> 0003
001006 -> 0004
>
```

Spaces are optional except between two consecutive numbers. When actions are specified on the command line after the address, no further input is taken from the keyboard for that command; after executing the specified actions, a new command is prompted for. Note that these commands provide the ability to write to a location (such as an I/O register) without reading from it; and provide the ability to query a location without having to interact.

3. Command Descriptions

In the descriptions listed below, the command letters in `typewriter text` are the commands, and things in *italic font* represent things that you substitute. Things in brackets are optional.

A [*n*][*actions*] Open A-register *n* ($0 \leq n \leq 7$, default zero). A7 is the System Stack Pointer; to see the User Stack Pointer, use the `r` command. For further explanation, see the section, 'Syntax for Memory and Register Access' above.

B [!][*args*] Boot. Resets appropriate parts of the system, then bootstraps the system. This allows bootstrap loading of programs from various devices such as disk, tape, or Ethernet. Typing 'b?' lists all possible boot devices. Simply typing 'b' gives you a default boot, which is configuration dependent. For an explanation of the booting options, see the sections on cw*Automatic Startup* in the appendix to the *System Manager's Manual for the Sun Workstation.*

If the first character of the argument is a '!', the system is not reset, and the bootstrapped program is not automatically executed. To execute it, use the 'C' command described below.

C [*addr*] Continue a program. The address *addr*, if given, is the address at which execution will begin; default is the current PC. The registers will be restored to the values shown by the A, D, and R commands.

D [*n*][*actions*] Open D-register *n* ($0 \leq n \leq 7$, default zero). For a detailed explanation, see the section, 'Syntax for Memory and Register Access' above.

E [*addr*][*actions*] Open the word at memory address *addr* (default zero) in the address space defined by the 'S' command. For a detailed explanation, see the section, 'Syntax for Memory and Register Access' above.

G [*addr*][*param*] Start the program by executing a subroutine call to the address *addr* if given, or else to the current PC. The values of the address and data registers are undefined; the status register will contain 0x2700. One parameter is passed to

the subroutine on the stack; it is the address of the remainder of the command line following the last digit of ***addr*** (and possible blanks).

K [*number*] If ***number*** is 0 (or not given), this does a 'Reset Instruction': it resets the system without affecting main memory or maps. If ***number*** is 1, this does a 'Medium Reset', which re-initializes most of the system without clearing memory. If ***number*** is 2, a hard reset is done and memory is cleared. This is equivalent to a power-on reset and runs the PROM-based diagnostics, which can take ten seconds or so.

L [*addr*][*actions*] Open the longword at memory address ***addr*** (default zero) in the address space defined by the 'S' command. For a detailed explanation, see the section, 'Syntax for Memory and Register Access' above.

M [*addr*] [*actions*] Opens the Segment Map entry which maps virtual address ***addr*** (default zero) in the current context. The choice of supervisor or user context is determined by the 'S' command setting (0–3 = user; 4–7 = supervisor). See the section, 'Syntax for Memory and Register Access' above.

O [*addr*][*actions*] Opens the byte location specified (default zero) in the address space defined by the 'S' command. See the section, 'Syntax for Memory and Register Access' above.

P [*addr*] [*actions*] Opens the Page Map entry which maps virtual address ***addr*** (default zero) in the current context. The choice of supervisor or user context is determined by the 'S' command setting (0–3 = user; 4–7 = supervisor). With each page map entry, the relevant segment map entry is displayed in brackets. See the section, 'Syntax for Memory and Register Access' above.

R [*actions*] Opens the miscellaneous registers (in order): SS (Supervisor Stack Pointer), US (User Stack Pointer), SF (Source Function Code), DF (Destination Function Code), VB (Vector Base), SC (System Context), UC (User Context), SR (Status Register), and PC (Program Counter). Alterations made to these registers (except SC and UC) do not take effect until the next 'C' command. For further explanation, see the section, 'Syntax for Memory and Register Access' above.

S [*number*] Sets or queries the address space to be used by subsequent memory access commands. ***number*** is the function code to be used, ranging from 1 to 7. Useful values are 1 (user data), 2 (user program), 3 (memory maps), 5 (supervisor data), 6 (supervisor program). If no ***number*** is supplied, the current setting is printed. Upon entry into the monitor, this is set to 5 if the program was in supervisor state, or to 1 if the program was in user state.

U [*arg*]

The U command manipulates the serial ports and switches the current input or output device. The argument may have the following values ('{AB}' means that either 'A' or 'B' is specified):

{AB}	Select serial port A (or B) as input and output device
{AB}io	Select serial port A (or B) as input and output device
{AB}i	Select serial port A (or B) for input only
{AB}o	Select serial port A (or B) for output only
k	Select keyboard for input
ki	Select keyboard for input
s	Select screen for output
so	Select screen for output
ks, sk	Select keyboard for input and screen for output
{AB}#	Set speed of serial port A (or B) to # (such as 1200, 9600, ...)
e	Echo input to output
ne	Don't echo input to output
u *addr*	Set virtual serial port address

If no argument is specified, the U command reports the current values of the settings. If no serial port is specified when changing speeds, the 'current' input device is changed.

At power-up, the following default settings are used: The default console input device is the Sun keyboard or, if the keyboard is unavailable, serial port A. The default console output device is the Sun screen or, if the graphics board is unavailable, serial port A. All serial ports are set to 9600 baud.