A Pageable Memory Based Filesystem

Marshall Kirk McKusick

Michael J. Karels

Keith Bostic

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

email: mckusick@cs.Berkeley.EDU
telephone: 415-642-4948

ABSTRACT

This paper describes the motivations for memory-based filesystems. It compares techniques used to implement them and describes the drawbacks of using dedicated memory to support such filesystems. To avoid the drawbacks of using dedicated memory, it discusses building a simple memory-based filesystem in pageable memory. It details the performance characteristics of this filesystem and concludes with areas for future work.

1. Introduction

This paper describes the motivation for and implementation of a memory-based filesystem. Memory-based filesystems have existed for a long time; they have generally been marketed as RAM disks or sometimes as software packages that use the machine's general purpose memory.[White1980a]

A RAM disk is designed to appear like any other disk peripheral connected to a machine. It is normally interfaced to the processor through the I/O bus and is accessed through a device driver similar or sometimes identical to the device driver used for a normal magnetic disk. The device driver sends requests for blocks of data to the device and the requested data is then DMA'ed to or from the requested block. Instead of storing its data on a rotating magnetic disk, the RAM disk stores its data in a large array of random access memory or bubble memory. Thus, the latency of accessing the RAM disk is nearly zero compared to the 15-50 milliseconds of latency incurred when access rotating magnetic media. RAM disks also have the benefit of being able to transfer data at the maximum DMA rate of the system, while disks are typically limited by the rate that the data passes under the disk head.

Software packages simulating RAM disks operate by allocating a fixed partition of the system memory. The software then provides a device driver interface similar to the one described for hardware RAM disks, except that it uses memory-to-memory copy instead of DMA to move the data between the RAM disk and the system buffers, or it maps the contents of the RAM disk into the system buffers. Because the memory used by the RAM disk is not available for other purposes, software RAM-disk solutions are used primarily for machines with limited addressing capabilities such as PC's that do not have an effective way of using the extra memory anyway.

Most software RAM disks lose their contents when the system is powered down or rebooted. The contents can be saved by using battery backed-up memory, by storing critical filesystem data structures in the filesystem, and by running a consistency check program after each reboot. These conditions increase the hardware cost and potentially slow down the speed of the disk. Thus, RAM-disk filesystems are not typically designed to survive power failures; because of their volatility, their usefulness is limited to transient or easily recreated information such as might be found in **/tmp**. Their primary benefit is that they have higher throughput than disk based filesystems.[Smith1981a] This improved throughput is particularly useful for utilities that make heavy use of temporary files, such as compilers. On fast processors, nearly half of the elapsed time for a compilation is spent waiting for synchronous operations required for file creation and deletion. The use of the memory-based filesystem nearly eliminates this waiting time.

Using dedicated memory to exclusively support a RAM disk is a poor use of resources. The overall throughput of the system can be improved by using the memory where it is getting the highest access rate. These needs may shift between supporting process virtual address spaces and caching frequently used disk blocks. If the memory is dedicated to the filesystem, it is better used in a buffer cache. The buffer cache permits faster access to the data because it requires only a single memory-to-memory copy from the kernel to the user process. The use of memory is used in a RAM-disk configuration may require two memory-to-memory copies, one from the RAM disk to the buffer cache, then another copy from the buffer cache to the user process.

The new work being presented in this paper is building a prototype RAM-disk filesystem in pageable memory instead of dedicated memory. The goal is to provide the speed benefits of a RAM disk without paying the performance penalty inherent in dedicating part of the physical memory on the machine to the RAM disk. By building the filesystem in pageable memory, it competes with other processes for the available memory. When memory runs short, the paging system pushes its least-recently-used pages to backing store. Being pageable also allows the filesystem to be much larger than would be practical if it were limited by the amount of physical memory that could be dedicated to that purpose. We typically operate our **/tmp** with 30 to 60 megabytes of space which is larger than the amount of memory on the machine. This configuration allows small files to be accessed quickly, while still allowing **/tmp** to be used for big files, although at a speed more typical of normal, disk-based filesystems.

An alternative to building a memory-based filesystem would be to have a filesystem that never did operations synchronously and never flushed its dirty buffers to disk. However, we believe that such a filesystem would either use a disproportionately large percentage of the buffer cache space, to the detriment of other filesystems, or would require the paging system to flush its dirty pages. Waiting for other filesystems to push dirty pages subjects them to delays while waiting for the pages to be written. We await the results of others trying this approach.[Ohta1990a]

2. Implementation

The current implementation took less time to write than did this paper. It consists of 560 lines of kernel code (1.7K text + data) and some minor modifications to the program that builds disk based filesystems, *newfs*. A condensed version of the kernel code for the memory-based filesystem are reproduced in Appendix 1.

A filesystem is created by invoking the modified *newfs*, with an option telling it to create a memory-based filesystem. It allocates a section of virtual address space of the requested size and builds a filesystem in the memory instead of on a disk partition. When built, it does a *mount* system call specifying a filesystem type of MFS (Memory File System). The auxiliary data parameter to the mount call specifies a pointer to the base of the memory in which it has built the filesystem. (The auxiliary data parameter used by the local filesystem, *ufs*, specifies the block device containing the filesystem.)

The mount system call allocates and initializes a mount table entry and then calls the filesystem-specific mount routine. The filesystem-specific routine is responsible for doing the mount and initializing the filesystem-specific portion of the mount table entry. The memory-based filesystem-specific mount routine, *mfs_mount*(), is shown in Appendix 1. It allocates a block-device vnode to represent the memory disk device. In the private area of this vnode it stores the base address of the filesystem and the process identifier of the *newfs* process for later reference when doing I/O. It also initializes an I/O list that it uses to record outstanding I/O requests. It can then call the *ufs* filesystem mount routine, passing the special block-device vnode that it has created instead of the usual disk block-device vnode. The mount proceeds just as any other local mount, except that requests to read from the block device are vectored through *mfs_strategy*() (described below) instead of the usual *spec_strategy*() block device I/O function. When the mount is completed, *mfs_mount*() does not return as most other filesystem mount functions do; instead it sleeps in the kernel awaiting I/O requests. Each time an I/O request is posted for the filesystem, a wakeup is issued for the corresponding *newfs* process. When awakened, the process checks for requests on its buffer list. A read request is serviced by copying data from the section of the *newfs* address space corresponding to the requested disk block to the kernel buffer. Similarly a write request is serviced by copying data to the section of the *newfs* address space corresponding to the requested disk block from the kernel buffer. When all the requests have been serviced, the *newfs* process returns to sleep to await more requests.

Once mounted, all operations on files in the memory-based filesystem are handled by the *ufs* filesystem code until they get to the point where the filesystem needs to do I/O on the device. Here, the filesystem encounters the second piece of the memory-based filesystem. Instead of calling the special-device strategy routine, it calls the memory-based strategy routine, *mfs_strategy*(). Usually, the request is serviced by linking the buffer onto the I/O list for the memory-based filesystem vnode and sending a wakeup to the *newfs* process. This wakeup results in a context-switch to the *newfs* process, which does a copyin or copyout as described above. The strategy routine must be careful to check whether the I/O request is coming from the *newfs* process itself, however. Such requests happen during mount and unmount operations, when the kernel is reading and writing the superblock. Here, *mfs_strategy*() must do the I/O itself to avoid deadlock.

The final piece of kernel code to support the memory-based filesystem is the close routine. After the filesystem has been successfully unmounted, the device close routine is called. For a memory-based filesystem, the device close routine is *mfs_close*(). This routine flushes any pending I/O requests, then sets the I/O list head to a special value that is recognized by the I/O servicing loop in *mfs_mount*() as an indication that the filesystem is unmounted. The *mfs_mount*() routine exits, in turn causing the *newfs* process to exit, resulting in the filesystem vanishing in a cloud of dirty pages.

The paging of the filesystem does not require any additional code beyond that already in the kernel to support virtual memory. The *newfs* process competes with other processes on an equal basis for the machine's available memory. Data pages of the filesystem that have not yet been used are zero-fill-on-demand pages that do not occupy memory, although they currently allocate space in backing store. As long as memory is plentiful, the entire contents of the filesystem remain memory resident. When memory runs short, the oldest pages of *newfs* will be pushed to backing store as part of the normal paging activity. The pages that are pushed usually hold the contents of files that have been created in the memory-based filesystem but have not been recently accessed (or have been deleted).[Leffler1989a]

3. Performance

The performance of the current memory-based filesystem is determined by the memory-to-memory copy speed of the processor. Empirically we find that the throughput is about 45% of this memory-to-memory copy speed. The basic set of steps for each block written is:

1) memory-to-memory copy from the user process doing the write to a kernel buffer
2) context-switch to the *newfs* process
3) memory-to-memory copy from the kernel buffer to the *newfs* address space
4) context switch back to the writing process

Thus each write requires at least two memory-to-memory copies accounting for about 90% of the CPU time. The remaining 10% is consumed in the context switches and the filesystem allocation and block location code. The actual context switch count is really only about half of the worst case outlined above because read-ahead and write-behind allow multiple blocks to be handled with each context switch.

On the six-MIPS CCI Power 6/32 machine, the raw reading and writing speed is only about twice that of a regular disk-based filesystem. However, for processes that create and delete many files, the speedup is considerably greater. The reason for the speedup is that the filesystem must do two synchronous operations to create a file, first writing the allocated inode to disk, then creating the directory entry. Deleting a file similarly requires at least two synchronous operations. Here, the low latency of the memory-based filesystem is noticeable compared to the disk-based filesystem, as a synchronous operation can be done with just two context switches instead of incurring the disk latency.

4. Future Work

The most obvious shortcoming of the current implementation is that filesystem blocks are copied twice, once between the *newfs* process' address space and the kernel buffer cache, and once between the kernel buffer and the requesting process. These copies are done in different process contexts, necessitating two context switches per group of I/O requests. These problems arise because of the current inability of the kernel to do page-in operations for an address space other than that of the currently-running process, and the current inconvenience of mapping process-owned pages into the kernel buffer cache. Both of these problems are expected to be solved in the next version of the virtual memory system, and thus we chose not to address them in the current implementation. With the new version of the virtual memory system, we expect to use any part of physical memory as part of the buffer cache, even though it will not be entirely addressable at once within the kernel. In that system, the implementation of a memory-based filesystem that avoids the double copy and context switches will be much easier.

Ideally part of the kernel's address space would reside in pageable memory. Once such a facility is available it would be most efficient to build a memory-based filesystem within the kernel. One potential problem with such a scheme is that many kernels are limited to a small address space (usually a few megabytes). This restriction limits the size of memory-based filesystem that such a machine can support. On such a machine, the kernel can describe a memory-based filesystem that is larger than its address space and use a "window" to map the larger filesystem address space into its limited address space. The window would maintain a cache of recently accessed pages. The problem with this scheme is that if the working set of active pages is greater than the size of the window, then much time is spent remapping pages and invalidating translation buffers. Alternatively, a separate address space could be constructed for each memory-based filesystem as in the current implementation, and the memory-resident pages of that address space could be mapped exactly as other cached pages are accessed.

The current system uses the existing local filesystem structures and code to implement the memory-based filesystem. The major advantages of this approach are the sharing of code and the simplicity of the approach. There are several disadvantages, however. One is that the size of the filesystem is fixed at mount time. This means that a fixed number of inodes (files) and data blocks can be supported. Currently, this approach requires enough swap space for the entire filesystem, and prevents expansion and contraction of the filesystem on demand. The current design also prevents the filesystem from taking advantage of the memory-resident character of the filesystem. It would be interesting to explore other filesystem implementations that would be less expensive to execute and that would make better use of the space. For example, the current filesystem structure is optimized for magnetic disks. It includes replicated control structures, ‘‘cylinder groups’’ with separate allocation maps and control structures, and data structures that optimize rotational layout of files. None of this is useful in a memory-based filesystem (at least when the backing store for the filesystem is dynamically allocated and not contiguous on a single disk type). On the other hand, directories could be implemented using dynamically-allocated memory organized as linked lists or trees rather than as files stored in ‘‘disk’’ blocks. Allocation and location of pages for file data might use virtual memory primitives and data structures rather than direct and indirect blocks. A reimplementation along these lines will be considered when the virtual memory system in the current system has been replaced.

5. References

Leffler1989a.

S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System,* Addison-Wesley, Reading, MA (1989).

Ohta1990a.

Masataka Ohta and Hiroshi Tezuka, ‘‘A Fast /tmp File System by Async Mount Option,’’ *USENIX Association Conference Proceedings*, p. ???−??? (June 1990).

Smith1981a.

A. J. Smith, ‘‘Bibliography on file and I/O system optimizations and related topics,’’ *Operating Systems Review* **14**(4), p. 39−54 (October 1981).

White1980a.

R. M. White, ‘‘Disk Storage Technology,’’ *Scientific American* **243**(2), p. 138−148 (August 1980).

6. Appendix A - Implementation Details

```
/*
 * This structure defines the control data for the memory
 * based file system.
 */
struct mfsnode {
        struct   vnode *mfs_vnode;        /* vnode associated with this mfsnode */
        caddr_t  mfs_baseoff;             /* base of file system in memory */
        long     mfs_size;                /* size of memory file system */
        pid_t    mfs_pid;                 /* supporting process pid */
        struct   buf *mfs_buflist;        /* list of I/O requests */
};

/*
 * Convert between mfsnode pointers and vnode pointers
 */
#define  VTOMFS(vp)      ((struct mfsnode *)(vp)->v_data)
#define  MFSTOV(mfsp)    ((mfsp)->mfs_vnode)
#define  MFS_EXIT        (struct buf *)-1
```

VTOMFS
MFSTOV

```
/*
 * Arguments to mount MFS
 */
struct mfs_args {
        char     *name;          /* name to export for statfs */
        caddr_t  base;           /* base address of file system in memory */
        u_long   size;           /* size of file system */
};
```

```
/*
 * Mount an MFS filesystem.
 */
mfs_mount(mp, path, data)                                                   mfs_mount
        struct mount *mp;
        char *path;
        caddr_t data;
{
        struct vnode *devvp;
        struct mfsnode *mfsp;
        struct buf *bp;
        struct mfs_args args;

        /*
         * Create a block device to represent the disk.
         */
        devvp = getnewvnode(VT_MFS, VBLK, &mfs_vnodeops);
        mfsp = VTOMFS(devvp);
        /*
         * Save base address of the filesystem from the supporting process.
         */
        copyin(data, &args, (sizeof mfs_args));
        mfsp->mfs_baseoff = args.base;
        mfsp->mfs_size = args.size;
        /*
         * Record the process identifier of the supporting process.
         */
        mfsp->mfs_pid = u.u_procp->p_pid;
        /*
         * Mount the filesystem.
         */
        mfsp->mfs_buflist = NULL;
        mountfs(devvp, mp);
        /*
         * Loop processing I/O requests.
         */
        while (mfsp->mfs_buflist != MFS_EXIT) {
                while (mfsp->mfs_buflist != NULL) {
                        bp = mfsp->mfs_buflist;
                        mfsp->mfs_buflist = bp->av_forw;
                        offset = mfsp->mfs_baseoff + (bp->b_blkno * DEV_BSIZE);
                        if (bp->b_flags & B_READ)
                                copyin(offset, bp->b_un.b_addr, bp->b_bcount);
                        else /* write_request */
                                copyout(bp->b_un.b_addr, offset, bp->b_bcount);
                        biodone(bp);
                }
                sleep((caddr_t)devvp, PWAIT);
        }
}
```

```
/*
 * If the MFS process requests the I/O then we must do it directly.
 * Otherwise put the request on the list and request the MFS process
 * to be run.
 */
mfs_strategy(bp)                                                    mfs_strategy
        struct buf *bp;
{
        struct vnode *devvp;
        struct mfsnode *mfsp;
        off_t offset;

        devvp = bp->b_vp;
        mfsp = VTOMFS(devvp);
        if (mfsp->mfs_pid == u.u_procp->p_pid) {
                offset = mfsp->mfs_baseoff + (bp->b_blkno * DEV_BSIZE);
                if (bp->b_flags & B_READ)
                        copyin(offset, bp->b_un.b_addr, bp->b_bcount);
                else /* write_request */
                        copyout(bp->b_un.b_addr, offset, bp->b_bcount);
                biodone(bp);
        } else {
                bp->av_forw = mfsp->mfs_buflist;
                mfsp->mfs_buflist = bp;
                wakeup((caddr_t)bp->b_vp);
        }
}

/*
 * The close routine is called by unmount after the filesystem
 * has been successfully unmounted.
 */
mfs_close(devvp)                                                    mfs_close
        struct vnode *devvp;
{
        struct mfsnode *mfsp = VTOMFS(vp);
        struct buf *bp;

        /*
         * Finish any pending I/O requests.
         */
        while (bp = mfsp->mfs_buflist) {
                mfsp->mfs_buflist = bp->av_forw;
                mfs_doio(bp, mfsp->mfs_baseoff);
                wakeup((caddr_t)bp);
        }
        /*
         * Send a request to the filesystem server to exit.
         */
        mfsp->mfs_buflist = MFS_EXIT;
        wakeup((caddr_t)vp);
}
```