tmpfs: A Virtual Memory File System

Peter Snyder

Sun Microsystems Inc.
2550 Garcia Avenue
Mountain View, CA 94043

ABSTRACT

This paper describes *tmpfs,* a memory-based file system that uses resources and structures of the SunOS virtual memory subsystem. Rather than using dedicated physical memory such as a ‘‘RAM disk’’, tmpfs uses the operating system page cache for file data. It provides increased performance for file reads and writes, allows dynamic sizing of the file system while requiring no disk space, and has no adverse effects on overall system performance. The paper begins with a discussion of the motivations and goals behind the development of tmpfs, followed by a brief description of the virtual memory resources required in its implementation. It then discusses how some common file system operations are accomplished. Finally, system performance with and without tmpfs is compared and analyzed.

1. Introduction

This paper describes the design and implementation of *tmpfs,* a file system based on SunOS virtual memory resources. Tmpfs does not use traditional non-volatile media to store file data; instead, tmpfs files exist solely in virtual memory maintained by the UNIX kernel. Because tmpfs file systems do not use dedicated physical memory for file data but instead use VM system resources and facilities, they can take advantage of kernel resource management policies.

Tmpfs is designed primarily as a performance enhancement to allow short lived files to be written and accessed without generating disk or network I/O. Tmpfs maximises file manipulation speed while preserving UNIX file semantics. It does not require dedicated disk space for files and has no negative performance impact.

Tmpfs is intimately tied to many of the SunOS virtual memory system facilities. Tmpfs files are written and accessed directly from the memory maintained by the kernel; they are not differentiated from other uses of physical memory. This means tmpfs file data can be ‘‘swapped’’ or paged to disk, freeing VM resources for other needs. General VM system routines are used to perform many low level tmpfs file system operations. This reduces the amount of code needed to maintain the file system, and ensures that tmpfs resource demands may coexist with other VM resource users with no adverse effects.

This paper begins by describing why tmpfs was developed and compares its implementation against other projects with similar goals. Following that is a description of its use and its appearance outside the kernel. Section 4 briefly discusses some of the structures and interfaces used in the tmpfs design and implementation. Section 5 explains basic file system operations and how tmpfs performs them differently than other file system types. Section 6 discusses and analyzes performance measurements. The paper concludes with a summary of tmpfs goals and features.

2. Implementation Goals

Tmpfs was designed to provide the performance gains inherent to memory-based file systems, while making use of existing kernel interfaces and minimising impact on memory resources. Tmpfs should support UNIX file semantics while remaining fully compatible with other file system types. Tmpfs should also provide additional file system space without using additional disk space or affecting other VM resource users.

File systems are comprised of two types of information; the data for files residing on a file system, and control and attribute information used to maintain the state of a file system. Some file system operations require that control information be updated synchronously so that the integrity of the file system is preserved. This causes performance degradation because of delays in waiting for the update's I/O request to complete.

Memory-based file systems overcome this and provide greater performance because file access only causes a memory-to-memory copy of data, no I/O requests for file control updates are generated. Physical memory-based file systems, usually called RAM disks, have existed for some time. RAM disks reserve a fairly large chunk of physical memory for exclusive use as a file system. These file systems are maintained in various ways; for example, a kernel process may fulfill I/O requests from a driver-level strategy routine that reads or writes data from private memory [McKusick1990].

RAM disks use memory inefficiently; file data exists twice in both RAM disk memory and kernel memory, and RAM disk memory that is not being used by the file system is wasted. RAM disk memory is maintained separately from kernel memory, so that multiple memory-to-memory copies are needed to update file system data.

Tmpfs uses memory much more efficiently. It provides the speed of a RAM disk because file data is likely to be in main memory, causing a single memory-to-memory copy on access, and because all file system attributes are stored once in physical memory, no additional I/O requests are needed to maintain the file system. Instead of allocating a fixed amount of memory for exclusive use as a file system, tmpfs file system size is dynamic depending on use, allowing the system to decide the optimal use of memory.

3. tmpfs usage

Tmpfs file systems are created by invoking the mount command with ‘‘tmp’’ specified as the file system type. The resource argument to mount (e.g., raw device) is ignored because tmpfs always uses memory as the file system resource. There are currently no mount options for tmpfs. Most standard mount options are irrelevant to tmpfs; for example, a ‘‘read only’’ mount of tmpfs is useless because tmpfs file systems are always empty when first mounted. All file types are supported, including symbolic links and block and character special device files. UNIX file semantics are supported[1]. Multiple tmpfs file systems can be mounted on a single system, but they all share the same pool of resources.

Because the contents of a volatile memory-based file system are lost across a reboot or unmount, and because these files have relatively short lifetimes, they would be most appropriate under /tmp, (hence the name *tmpfs).* This means /usr/tmp is an inappropriate directory in which to mount a tmpfs file system because by convention its contents persist across reboots.

The amount of free space available to tmpfs depends on the amount of unallocated swap space in the system. The size of a tmpfs file system grows to accommodate the files written to it, but there are some inherent tradeoffs for heavy users of tmpfs. Tmpfs shares resources with the data and stack segments of executing programs. The execution of very large programs can be affected if tmpfs file systems are close to their maximum allowable size. Tmpfs is free to allocate all but 4MB of the system's swap space. This is enough to ensure most programs can execute, but it is possible that some programs could be prevented from executing if tmpfs file systems are close to full. Users who expect to run large programs and make extensive use of tmpfs should consider enlarging the swap space for the system.

4. Design

SunOS virtual memory consists of all its available physical memory resources (e.g., RAM and file systems). Physical memory is treated as a cache of ‘‘pages’’ containing data accessed as memory ‘‘objects’’.

Named memory objects are referenced through UNIX file systems, the most common of which is a regular file. SunOS also maintains a pool of unnamed (or *anonymous,* defined in section 4.4) memory to facilitate

[1] except file and record locking, which will be supported in a future SunOS release

cases where memory cannot be accessed through a file system. Anonymous memory is implemented using the processor's primary memory and *swap* space

Tmpfs uses anonymous memory in the page cache to store and maintain file data, and competes for pages along with other memory users. Because the system does not differentiate tmpfs file data from other page cache uses, tmpfs files can be written to swap space. Control information is maintained in physical memory allocated from kernel heap. Tmpfs file data is accessed through a level of indirection provided by the VM system, whereby the data's tmpfs file and offset are transformed to an offset into a page of anonymous memory.

Tmpfs itself never causes file pages to be written to disk. If a system is low on memory, anonymous pages can be written to a swap device if they are selected by the pageout daemon or are mapped into a process that is being swapped out. It is this mechanism that allows pages of a tmpfs file to be written to disk.

The following sections provide an overview of the design and implementation of tmpfs. For detailed information about the SunOS virtual memory design and implementation, please consult the references [Gingell1987] and [Moran1988] listed at the end of the paper.

To understand how tmpfs stores and retrieves file data, some of the structures used by the kernel to maintain virtual memory are described. The following is a brief description of some key VM and tmpfs structures and their use in this implementation.

4.1. tmpfs data structures

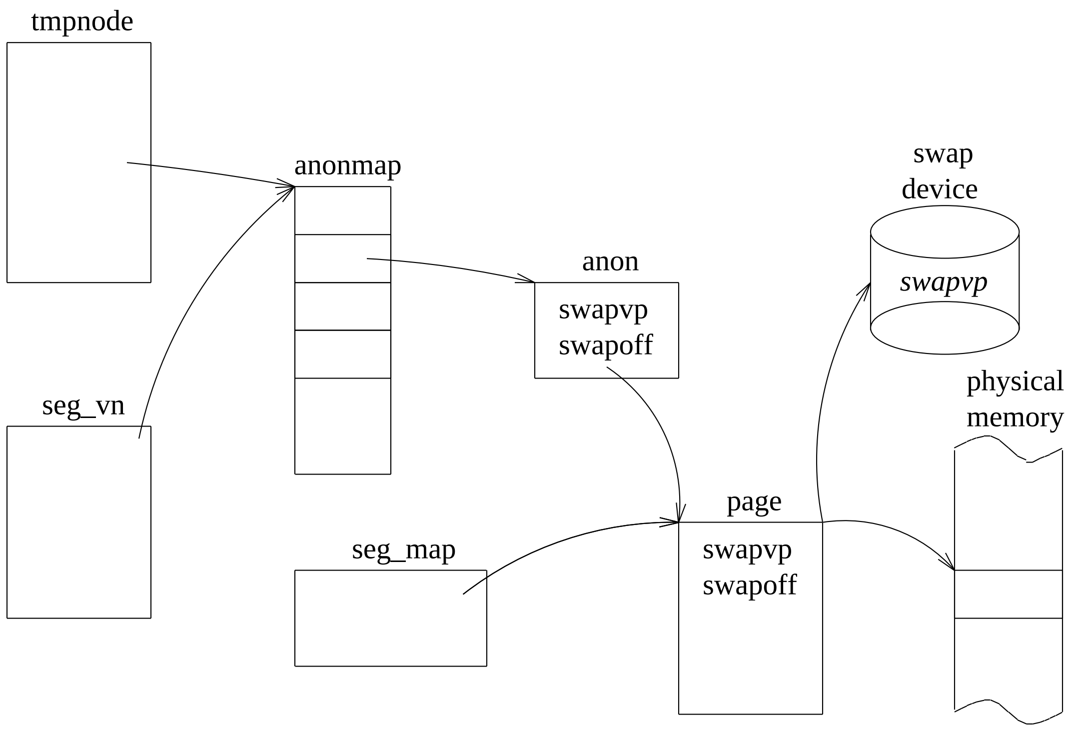

Figure 1. tmpfs block diagram

Figure 1 shows the basic structure and resources that tmpfs uses for files and their access. Their use in tmpfs is defined below.

4.2. vnodes

A *vnode* is the fundamental structure used within the kernel to name file system memory objects. Vnodes provide a file system independent structure that gives access to the data comprising a file object. Each open file has an associated vnode. This vnode contains a pointer to a file system dependent structure for the private use of the underlying file system object. It is by this mechanism that operations on a file pass through to the underlying file system object [Kleiman1986].

4.3. page cache

The VM system treats physical memory as a cache for file system objects. Physical memory is broken up into a number of *page frames,* each of which has a corresponding *page structure*. The kernel uses page structures to maintain the identity and status of each physical page frame in the system. Each page is identified by a vnode and offset pair that provides a handle for the physical page resident in memory and that also designates the part of the file system object the page frame caches.

4.4. anonymous memory

anonymous memory is term describing page structures whose name (i.e., vnode and offset pair) is not part of a file system object. Page structures associated with anonymous memory are identified by the vnode of a swap device and the offset into that device. It is not possible to allocate more anonymous memory than there is swap space in a system. The location of swap space for an anonymous page is determined when the anonymous memory is allocated. Anonymous memory is used for many purposes within the kernel, such as: the uninitialised data and stack segments of executing programs, System V shared memory, and pages created through copy on write faults [Moran1988].

An *anon structure* is used to name every anonymous page in the system. This structure introduces a level of indirection between anonymous pages and their position on a swap device.

An *anon_map* structure contains an array of pointers to anon structures and is used to treat a collection of anonymous pages as a unit. Access to an anonymous page structure is achieved by obtaining the correct anon structure from an anon_map. The anon structure contains the swap vnode and offset of the anonymous page.

4.5. tmpnode

A *tmpnode* contains information intrinsic to tmpfs file access. It is similar in content and function to an inode in other UNIX file systems (e.g., BSD Fast file system). All file-specific information is included in this structure, including file type, attributes, and size. Contained within a tmpnode is a vnode. Every tmpnode references an anon_map which allows the kernel to locate and access the file data. References to an offset within a tmpfs file are translated to the corresponding offset in an anon_map and passed to routines dealing with anonymous memory. In this way, an anon_map is used in similarly to the direct disk block array of an inode. tmpnodes are allocated out of kernel heap, as are all tmpfs control structures.

Directories in tmpfs are special instances of tmpnodes. A tmpfs directory structure consists of an array of filenames stored as character strings and their corresponding tmpnode pointers.

4.6. vnode segment (seg_vn)

SunOS treats individual mappings within a user address space in an object oriented manner, with public and private data and an operations vector to maintain the mapping. These objects are called ‘‘segments’’ and the routines in the ‘‘ops’’ vector are collectively termed the ‘‘segment driver’’.

The *seg_vn* structure and segment driver define a region of user address space that is mapped to a regular file. The seg_vn structure describes the range of a file being mapped, the type of mapping (e.g., shared or private), and its protections and access information. User mappings to regular files are established through the *mmap* system call.

The seg_vn data structure references an *anon_map* similarly in structure and use to tmpfs. When a mapping to a tmpfs file is first established, the seg_vn structure is initialised to point to the anon_map associated with the tmpfs file. This ensures that any change to the tmpfs file (e.g., truncation) is reflected in all seg_vn mappings to that file.

4.7. kernel mapping (seg_map)

The basic algorithm for SunOS read and write routines is for the system to first establish a mapping in kernel virtual address space to a vnode and offset, then copy the data from or to the kernel address as appropriate (i.e., for read or write, respectively). Kernel access to non-memory resident file data causes a page fault, which the seg_map driver handles by calling the appropriate file system routines to read in the data.

The kernel accesses file data much the same way a user process does when using file mappings. Users are presented with a consistent view of a file whether they are mapping the file directly or accessing it through read or write system calls.

The seg_map segment driver provides a structure for maintaining vnode and offset mappings of files and a way to resolve kernel page faults when this data is accessed. The seg_map driver maintains a cache of these mappings so that recently-accessed offsets within a file remain in memory, decreasing the amount of page fault activity.

5. Implementation

All file operations pass through the vnode layer, which in turn calls the appropriate tmpfs routine. In general, all operations that manipulate file control information (e.g., truncate, setattr, etc.) are handled directly by tmpfs. Tmpfs uses system virtual memory routines to read and write file data. Page faults are handled by anonymous memory code which reads in the data from a swap device if it is not memory resident.

Algorithms for some basic file system operations are outlined below.

5.1. mount

Tmpfs file systems are mounted in much the same manner as any other file system. As with other file systems, a vfs structure is allocated, initialised and added to the kernel's list of mounted file systems. A tmpfs-specific mount routine is then called, which allocates and initialises a tmpfs mount structure, and then allocates the root directory for the file system. A tmpfs mount point is the root of the entire tmpfs directory tree, which is always memory resident.

5.2. read/write

The offset and vnode for a particular tmpfs file are passed to tmpfs through the vnode layer from the read or write system call. The tmpfs read or write routine is called, which locates the anon structure for the specified offset. If a write operation is extending the file, or if a read tries to access an anon structure that does not yet exist (i.e., a hole exists in the file), an anon structure is allocated and initialised. If it is a write request, the anon_map is grown if it is not already large enough to cover the additional size. The true location of the tmpfs file page (i.e., the vnode and offset on the swap device) is found from the anon structure. A kernel mapping (using a seg_map structure) is made to the swap vnode and offset. The kernel then copies the data between the kernel and user address space. Because the kernel mapping has the vnode and offset of the swap device, if the file page needs to be faulted in from the swap device, the appropriate file system operations for that device will be executed.

5.3. Mapped files

Mappings to tmpfs files are handled in much the same way as in other file systems. A seg_vn structure is allocated to cover the specified vnode and offset range. However, tmpfs mappings are accessed differently than with other file systems.

With many file systems, a seg_vn structure contains the vnode and offset range corresponding to the file being mapped. With tmpfs, the segment is initialised with a null vnode. Some seg_vn segment driver routines assume that pages should be initialised to the vnode contained in the seg_vn structure, unless the vnode is null. If the vnode is set to be that of the tmpfs file, routines expecting to write the page out to a swap device (e.g., pageout daemon), would write the page to the tmpfs file with no effect. Instead, routines initialise pages with the swap vnode of the swap device.

Shared mappings to tmpfs files simply share the anon_map with the files tmpnode. Private mappings allocate a new anon_map and copy the pointers to the anon structures, so that copy-on-write operations can be performed.

6. Performance

All performance measurements were conducted a SPARCStation 1 configured with 16MB physical memory and 32MB of local (ufs) swap.

6.1. File system operations

Table 1 refers to results from a Sun internal test suite developed to verify basic operations of ‘‘NFS’’ file system implementations. Nine tests were used:

create	Create a directory tree 5 levels deep.
remove	Removes the directory tree from create.
lookup	stat a directory 250 times
setattr	chmod and stat 10 files 50 times each
read/write	write and close a 1MB file 10 times, then read the same file 10 times.
readdir	read 200 files in a directory 200 times.
link/rename	rename, link and unlink 10 files 200 times.
symlink	create and read 400 symlinks on 10 files.
statfs	stat the tmpfs mount point 1500 times.

Test type	nfs *(sec)*	ufs *(sec)*	tmpfs *(sec)*
create	24.15	16.44	0.14
remove	20.23	6.94	0.80
lookups	0.45	0.22	0.22
setattr	19.23	22.31	0.48
write	135.22	25.26	2.71
read	1.88	1.76	1.78
readdirs	10.20	5.45	1.85
link/rename	14.98	13.48	0.23
symlink	19.84	19.93	0.24
statfs	3.96	0.27	0.26

Table 1: nfs test suite

The create, remove, setattr, write, readdirs, link and symlink benchmarks all show an order of magnitude performance increase under tmpfs. This is because for tmpfs, these file system operations are performed completely within memory, but with ufs and nfs, some system I/O is required. The other operations (lookup, read, and statfs) do not show the same performance improvements largely because they take advantage of various caches maintained in the kernel.

6.2. File create and deletes

While the previous benchmark measured the component parts of file access, this benchmark measures overall access times. This benchmark was first used to compare file create and deletion times for various operating systems [Ousterhout1990].

The benchmark opens a file, writes a specified amount of data to it, and closes the file. It then reopens the file, rereads the data, closes, and deletes the file. The numbers are the average of 100 runs.

File size *(kilobytes)*	nfs *(ms)*	ufs *(ms)*	tmpfs *(ms)*
0	82.63	72.34	1.61
10	236.29	130.50	7.25
100	992.45	405.45	46.30
1024 (1MB)	15600.86	2622.76	446.10

Table 2: File creates and deletes

File access under tmpfs show great performance gains over other file systems. As the file size grows, the difference in performance between file system types decreases. This is because as the file size increases, all of the file system read and write operations take greater advantage of the kernel page cache.

6.3. kernel compiles

Table 3 presents compilation measurements for various types of files with ‘‘/tmp’’ mounted from the listed file system types. The ‘‘large file’’ consisted of approximately 2400 lines of code. The ‘‘benchmark’’ compiled was the NFS test suite from the section above. The kernel was a complete kernel build from scratch.

Compile type	nfs	ufs	tmpfs
large file	50.46	40.22	32.82
benchmark	50.72	47.98	38.52
kernel	39min49.9	32min27.45	27min.8.11

Table 3: Typical compile times (in seconds)

Even though tmpfs is always faster than either ufs or nfs, the differences are not as great as with the previous benchmarks. This is because the compiler performance is affected more by the speed of the CPU and compiler rather than I/O rates to the file system. Also, there can be much system paging activity during compiles, causing tmpfs pages to be written to swap and decreasing the performance gains.

6.4. Performance discussion

Tmpfs performance varies depending on usage and machine configurations. A system with ample physical memory but slow disk or on a busy network, notices improvements from using tmpfs much more than a machine with minimal physical memory and a fast local disk. Applications that create and access files that fit within the available memory of a system have much faster performance than applications that create large files causing a demand for memory. When memory is in high demand, the system writes pages out and frees them for other uses. Pages associated with tmpfs files are just as likely to be written out to backing store as other file pages, minimising tmpfs performance gains.

Tmpfs essentially caches in memory those writes normally scheduled for a disk or network file system. Tmpfs provides the greatest performance gains for tasks that generate the greatest number of file control updates. Tmpfs never writes out control information for files, so directory and file manipulation is always a performance gain.

7. Summary

The tmpfs implementation meets its design goals. Tmpfs shows the performance gains associated with memory-based file systems but also provides significant advantages over RAM disk style file systems. Tmpfs uses memory efficiently because it is not a fixed size, and memory not used by tmpfs is available for other uses. It is tightly integrated with the virtual memory system and so takes advantage of system page cache and the kernel’s resource management policies. Tmpfs uses many of the kernel’s interfaces and facilities to perform file system operations. It provides additional file system space, and supports UNIX file semantics while remaining fully compatible with other file system types.

8. Acknowledgements

Thanks to Glenn Skinner, Howard Chartock, Bill Shannon, and Anil Shivalingiah, for their invaluable help and advice on this paper and the implementation of tmpfs. Also thanks to Larry McVoy for lively debate and suggestions, Marguerite Sprague for editing skills, and Herschel Shermis who wrote the initial implementation of tmpfs.

References

Gingell1987.
Robert A. Gingell, Joseph P. Moran, and William A. Shannon, ‘‘Virtual Memory Architecture in SunOS,’’ *Proceedings of the Summer 1987 Usenix Technical Conference*, Usenix Association, Phoenix Arizona, USA, June 1987.

Kleiman1986.
Steven R. Kleiman, ‘‘Vnodes: An Architecture for Multiple File Systems Types in Sun UNIX,’’ *Proceedings of the Summer 1986 Usenix Technical Conference*, Usenix Association, Phoenix Arizona, USA, June 1986.

McKusick1990.
Marshall Kirk McKusick, Michael J. Karels, and Keith Bostic, ‘‘A Pageable Memory Based Filesystem,’’ *Proceedings of the Summer 1990 Usenix Technical Conference*, Usenix Association, Anaheim California, USA, June 1990.

Moran1988.
Joseph P. Moran, ‘‘SunOS Virtual Memory Implementation,’’ *Proceedings for Spring 1988 EUUG Conference*, EUUG, London England, Spring 1988.

Ousterhout1990.
John K. Ousterhout, ‘‘Why Aren’t Operating Systems Getting Faster As Fast as Hardware,’’ *Proceedings of the Summer 1990 Usenix Technical Conference*, Usenix Association, Anaheim California, USA, June 1990.