Virtual Swap Space in SunOS

Howard Chartock
Peter Snyder

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View, Ca. 94043
howard@sun.com
peter@sun.com

ABSTRACT

The concept of swap space in SunOS has been extended by the abstraction of a "virtual swap file system", which allows the system to treat main memory as if it were backing store. Further, this abstraction allows for more intelligent choices to be made about swap space allocation than in the current system. This paper contrasts the existing mechanisms and their limitations with the modifications made to implement virtual swap space.

1. Introduction

SunOS provides a unified set of abstractions and interfaces for interacting with virtual memory (VM) facilities both from user programs and from within the kernel. One of the major features of this architecture is that address spaces are constructed out of mappings to virtual memory objects. These mappings cause main memory pages to cache the contents of the virtual memory objects and each physical page is named by the VM object that backs it. [Gingell1987]

One common type of virtual memory object is an ordinary file. Establishing a mapping to a file makes its contents directly accessible at the address of the mapping. For example, the kernel creates a mapping to an executable file in order to execute it; mappings are also used by applications to access file contents without the copy overhead inherent in the read and write system calls.

A second commonly mapped VM object is known as *anonymous memory*. This term is used because, unlike file mappings, the names of the backing objects are unknown to the client. Anonymous memory mappings are backed by swap space; each physical page in the mapping is randomly assigned a name from the system's pool of available swap space at the time the page first comes into existence. The system uses anonymous memory for several purposes: for private copies of data created during copy-on-write faults, for process data and stack segments, and as a storage resource for the *tmpfs* file system [Snyder1990].

Although the existing anonymous memory scheme provides a service that is both powerful and flexible, it has some significant limitations. One is that physical backing store must always be reserved for anonymous memory mappings, even if the client's environment or application doesn't use it. The system requires backing store for anonymous memory so that page frames can be written out and reused when memory contention increases. Thus, for example, to run an application with a large data segment the system must be configured with lots of swap space, even if pages of the segment will seldom need to be written out to backing store.

A second limitation is that the algorithm for associating backing store with an anonymous memory page is, while very simple, limited and inflexible. The backing store for an anonymous page is chosen at random from the pool of available store when the page is first accessed, and can never be changed afterwards. If the backing store could be chosen later or dynamically moved to different locations, a more intelligent method could be used to make allocation decisions. Such decisions could, for example, employ information about page usage patterns to choose backing store locations that would increase I/O performance during paging and swapping.

To address these issues, the concept of *virtual swap space* was introduced into SunOS. Virtual swap space provides a layer of abstraction between anonymous memory pages and the physical store that may back those pages. The system's virtual swap space is equal to the sum of all its physical (i.e. disk-backed) swap space plus a portion of the currently available main memory. Because virtual swap space does not necessarily correspond to physical storage, the need to configure a system with large amounts of physical swap space can be significantly reduced. Also, because virtual swap space sits "in between" anonymous memory pages and physical swap space, the virtual swap space object can make more flexible and dynamic decisions than in the current system, about what physical backing store to allocate for a page.

To implement the concept of virtual swap space, a new pseudo-filesystem type called *swapfs* was added to the system. Swapfs is a pseudo-filesystem because it does not actually control any physical storage. Rather, its purpose is to provide names for anonymous memory pages. Whenever a part of the system, for example the pageout daemon, invokes a file system operation on a page named by swapfs, it gains control of the page. This gives swapfs great flexibility in deciding the page's fate; for example, at this time it may, if it chooses, change the name of the page so that it is backed by real physical store.

This paper describes the existing anonymous memory mechanisms in SunOS and the modifications that were made to them to implement the virtual swap space abstraction.

2. The Existing Implementation

The current anonymous memory mechanisms consist of an *anon layer,* which provides anonymous memory services to the rest of the kernel, and a *swap layer*, which provides backing store services to the anon layer. These layers interact heavily with file system objects, the page layer, and with mapping objects that employ their services.

2.1. Vnodes and Pages

In SunOS, the *vnode* [Kleiman1986] is the fundamental structure used to name file system objects. Vnodes provide a file system independent abstraction that allows access to the data comprising a file object. The vnode object provides a variety of methods for manipulating file objects, some of which are used to interact with the VM subsystem.

SunOS partitions main memory into a number of *page frames,* each of which has a corresponding *page structure* which describes the page frame. Each main memory page in the system is named by the backing store for that page. The name of the page, as stored in its associated page structure, is a (vnode, offset) pair; this name describes the location of the page's backing store.

SunOS uses main memory as a cache of file system objects. The vnode object provides the principal mechanism for manipulating this cache. The VM subsystem performs file system-related operations on a page using the `putpage()` and `getpage()` methods of the vnode which names the page. The `putpage()` operation causes a specified page to be written to backing store. The `getpage()` operation returns a page with a specified name; this may entail allocating a page frame or reading the page data off disk.

2.2. Mappings

A process's address space is composed of a number of mappings to virtual memory objects. Each mapping is represented by an object referred to as a *segment.* The most important service a segment provides is the handling of faults on an address within the segment. The segment is responsible for resolving a fault, if necessary, by filling a physical page from the backing store that the segment maps.

Several different types of segment objects exist in the current system. Perhaps the most heavily used is the vnode segment type *(segvn),* which provides both shared and private mapping to files [Moran1988]. A shared mapping always writes the current data of its mapped object. The same is true for a private mapping except that when it is first written, the VM system's anonymous memory facilities are used to create a private copy of its backing object. Finally, the vnode segment type may be used solely to map anonymous memory; for example, the `exec()` system call creates a vnode segment in this manner to provide a stack segment for a process.

2.3. Anon Layer

The system provides anonymous memory services through the *anon layer*. The anon layer provides operations that allow the client to reserve and unreserve anonymous backing store, to allocate or release a page of anonymous memory, and to create and fill a page backed by anonymous storage.

The system enforces the policy that clients must reserve anonymous backing store up front and only thereafter can allocations for individual pages be made against this reservation. The reservation policy guarantees that, in the face of insufficient anonymous memory, a process will fail synchronously (on return from a system call such as `exec()` or `mmap()`), rather than asynchronously (on a failure to resolve a fault due to a lack of backing store). Reservations are made against the total pool of physical backing store that has been added to the system as swap space.

A segment typically allocates backing store the first time a fault occurs on a page within the segment. The segment calls the anon layer to request a single page-sized unit of backing store from the pool of physical swap space and obtains an opaque handle for the allocation, an *anon slot.*

The anon slot contains the name of the backing store which will become the name of the associated anonymous page. In the current system, the name is implied by the memory address of the anon slot data structure. This tight binding makes it quite difficult to change the backing store for a particular anonymous page while that page and its associated anon slot are in use.

An anonymous memory client often wishes to use several anonymous pages. Typically the client stores the anon slots for these pages in an *anonmap* structure which contains an array of slot pointers.

The client passes an anon slot to the anon layer to perform subsequent operations on its associated anonymous page. For example, to get the main memory page associated with an anon slot, the client invokes the anon layer's `getpage()` operator using the anon slot as an argument. This `getpage()` operator, similar to that provided by vnode objects, first translates the anon slot into the name for the backing store. With this name in hand, the anon layer calls the underlying vnode `getpage()` operator to acquire the page.

2.4. Swap Layer

The *swap layer* manages pools of physical swap space and allocates and de-allocates page-sized units of space on demand from the anon layer. Swap devices can be added to the pool of available swap space using a system call interface. When a device is added to the pool, the total amount of swap space available for reservation is increased by the size of the device. At this time the swap layer also creates an administrative data structure for the swap device, called a *swapinfo* structure, which includes an array of anon slots. When the anon layer requests a new unit of backing store, the swap layer consults the swapinfo structures for the configured swap devices and returns an free anon slot. The position of the anon slot in the swapinfo array matches its offset into the swap device.

2.5. Summary

To summarise the above discussion consider the example in figure 1 in which a process is executing and accesses a byte in the second page of its data segment.

When the process issues an *exec()* system call, the system creates a vnode segment backed by anonymous memory (the "anon client" in the figure) for its data segment and reserves an amount of anonymous memory equal to the size of the segment. A fault occurs when the process touches a byte in the second page for the first time. The page fault handling code directs the fault to the appropriate segment, in this case a vnode segment, which then calls the anon layer to allocate a page of backing store. The anon layer in turn calls the swap layer, which allocates the store (in this case from the swap device named by vnode *"swapvp")* and returns an anon slot for it, corresponding to offset *off2* on the device. The anon layer translates the anon slot to the name of its backing store, creates a page with this name, and zeroes the page before returning it to the segment. The segment stores the returned anon slot in its anonmap for future use. At some later time the system may call the swapvp `putpage()` operator to push the contents of the page out to its backing store on the physical swap device associated with *swapvp.*

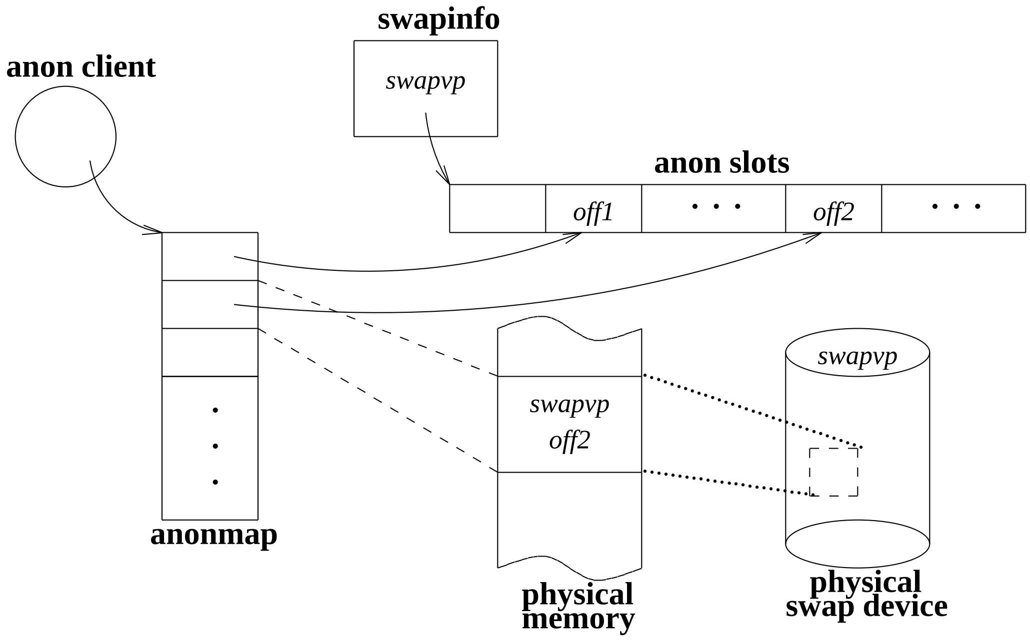

Figure 1. Existing swap/anon layers

3. The New Implementation: Virtual Swap Space

To address the limitations of the current anonymous memory scheme discussed in the introduction, the concept of *virtual swap space* was developed. Virtual swap space is an abstraction presented by the swap and anon layers to clients of anonymous memory. This abstraction causes anonymous memory pages to appear to the rest of the system as though they are backed by real swap space when in fact they are not.

As discussed above, vnode objects are used to name pages and associate them with backing store. Thus, to implement the notion of virtual backing store, a new vnode type, *swapfs,* was created to present virtual backing store to clients of the anon layer. Swapfs can hand out names for pages just as can any other file system type. However, unlike those of other file systems, these names do not correspond to real physical storage. Instead, they allow swapfs to gain control over the fate of the page when other parts of the system invoke file system operations on it. At such times swapfs can make decisions about renaming the page to be backed by real physical store.

The following discussion explains the changes in structure and mechanism used to implement the abstraction of virtual swap space.

3.1. Anonymous Memory Reservation

In the current system, a client of anonymous memory must reserve backing store up front before it attempts any allocations. The presence of swapfs in the system does not remove this requirement. However, swapfs does expand the pool of reservable swap space to include not just physical devices, but available main memory as well.

With swapfs, the total amount of reservable swap space becomes equal to the sum of all the space on the physical devices in the swap pool plus the amount of currently available main memory minus a safety factor. The ability to use main memory as allocatable swap space in conjunction with swapfs's ability to dynamically rename anonymous pages, allows users to run with reduced physical swap space and yet still have pages written out when needed.

The amount of available memory in the system is all that is not considered "wired down"[1]. Wired down

[1] Physical memory that is "wired down" is not pageable, and hence not available for use by the rest of the system.

memory includes space that the kernel has dynamically allocated for internal data structures as well as pages a user process has locked via the mlock() interface. Similarly, when swap space is reserved against main memory, this memory is also wired down, as the pages cannot be paged out.

The new reservation algorithm always reserves against physical swap space first. Only when this has been completely exhausted is the available main memory used. Regardless of how much main memory is available, the anon layer is never allowed to reserve a certain quantity (computed as a fixed fraction of the total main memory) for swap space. This policy insures that there will always be adequate memory for kernel data structure allocation. Once the anon layer begins to reserve against main memory, anonymous pages may be created in the system for which no physical backing store is available; these pages will be named by swapfs. If swapfs is unable to find physical store for such pages, they will effectively be unpageable; i.e., this memory will be unavailable to the rest of the system, until physical store is freed up.

When a swap space reservation is released, any main memory reserved by the anon layer is released first, and only when all main memory so held has been released, does the algorithm begin to release physical swap space. The algorithm works this way on the assumption that it is desirable to take main memory away from the system only as a last resort and to give it back at the earliest opportunity. Note that a reservation does not reserve a particular chunk of swap space, it simply guarantees that some is available somewhere in the total pool.

3.2. Allocating Backing Store

As discussed above, a client that has reserved swap space calls the anon layer to allocate a page-sized unit of backing store, and the name of that storage is returned to the client in the anon slot. In the current system, this allocation was made at random from the pool of available physical space, but in the new allocation scheme, it is always returned from swapfs, giving swapfs initial control over all anonymous pages in the system.

Thus, all clients of the anon layer receive backing store names from swapfs. Only swapfs itself requests real physical backing store from the swap layer, which it does when it wants to rename an anonymous page to physical store.

In the current system, the anon slot is returned by the swap layer with the name of the backing store implied in the memory address of the slot data structure. However, binding the name to the memory address of the data structure makes it quite difficult to change the name while the anon slot is in use. The anon slot is held by a client as long as the client makes use of the associated anonymous page; thus, if swapfs is to change the backing store while the page is in use, it must also change the name to which the anon slot refers.

To facilitate renaming, a level of indirection was added to the anon slot. The name of the backing store associated with the slot is now stored as *vnode* and *offset* fields in the slot data structure, allowing the values of these fields to be changed while the slot is in use. The swap layer no longer hands out anon slot data structures from per-swapinfo arrays; instead, the anon layer creates them on demand. The swap layer also no longer uses anon slots to track allocations; instead, it employs a bitmap of free slots for each swap device. The use of this bitmap solves a problem with current versions of SunOS, which keep unused slots on a freelist for later reuse. The algorithm for ordering this list can sometimes lead to poorly ordered swap allocations, which in turn cause poor paging performance.

3.3. Swapfs

The virtual swap space abstraction is implemented by the swapfs pseudo-filesystem type. Pseudo-filesystems have been used before to layer a new abstraction on top of what appears to the rest of the system as a file system [Rosenthal1990]. For example, *tmpfs* layers a filesystem on top of anonymous memory.

As is true for any other file system object, swapfs presents a set of methods for manipulating its objects. In constructing this new file system type, we were concerned only with its use within the kernel by the VM system. Thus, we did not build a full set of file operations which would, for example, allow a user to mount a swapfs file system, or give names to swapfs files. In fact, swapfs provides only three significant operations: `swapvp()`, which returns a vnode to the swap layer for use as a swap device; `getpage()`, which returns a page to the system backed by swapfs; and most importantly `putpage()`, which "pushes"[2] out an anonymous page backed by swapfs.

3.3.1. Swap Layer Modifications

Swapfs provides only one vnode to the system which is made available to the swap layer through the `swapvp()` operation when the system boots. For the most part, the swap layer treats swapfs as it does any other swap device. It is added to the list of swap devices managed by the swap layer and an administrative data structure, including an allocation bitmap, is created for the vnode.

Special-case modifications to some swap routines were needed to reflect some of the differences between swapfs and other swap file system types. For example, the interface routine called to allocate swap backing store, has been modified to take flags specifying where the backing store is to come from. As discussed above, all external clients of anonymous memory allocate backing store via the anon layer, which always requests this store from swapfs. Only swapfs itself makes allocation requests for physical swap space, when it wants to change the backing store for a given page.

3.3.2. getpage()

When a process faults on an anonymous page, the fault is directed to the segment that maps the address. On the first fault, the page may not exist, and there may be no backing store. In such a case, the segment calls the anon layer to allocate backing store. The backing store's name is extracted from the returned anon slot. The `getpage()` operator on this vnode is then called to create the page. Because client allocations of anonymous backing store are always satisfied by swapfs, this will result in a call to the swapfs `getpage()` operator. Swapfs satisfies this request by simplying creating a page with the requested name and handing it back to the segment. Unlike some other vnode objects, swapfs has no need to perform additional functions at this point, such as allocating physical disk space.

When a subsequent fault occurs on the same address, the fault will find its way to the same segment, which will again use the name in the anon slot to call the `getpage()` operator to retrieve the page. Most vnode `getpage()` operators have to be prepared to deal with the fact that the page may no longer be in memory. In this case they must create a new page and do I/O to fill it. Swapfs pages, however, once modified by the client, remain in memory unless renamed and paged out.

When a page has been modified, the system marks it "dirty". The system will not attempt to reuse the page until it has been marked "clean", and this is completely under the control of the vnode that owns the page. Typically a vnode marks a page clean in its `putpage()` operator when it pushes the page out to backing store. However, because swapfs represents virtual swap space, it does not page out pages itself and hence never marks them clean. Swapfs pages out pages by renaming them to be backed by real physical store, and the vnode to which the page is renamed then becomes responsible for cleaning the page.

Modified pages will never be reused by the system as long as they are named by swapfs. Because of this, when the swapfs `getpage()` operator is called, it knows that if a page has been modified it will be able to find it in the page cache. If the page has never been modified, then it may have been reused, but in such a case the client cannot possibly care about its contents, so swapfs simply creates a new one and returns it.

3.3.3. putpage()

At certain times the system will push pages to their backing store. This may happen, for example, when the pageout daemon pushes pages out to try to accommodate memory demands on the system, or it may happen when the scheduler decides to swap out a process. The pageout function entails getting the name of the page and calling the `putpage()` operator of the associated vnode. For swapfs, this operator does some very unusual things. Swapfs makes a call to the swap layer and asks it to allocate a page of physical backing store. If this request fails, nothing further is done with the page; it remains modified and simply stays in memory. However, if the swap allocation succeeds, swapfs renames the page to the new backing store and then calls the `putpage()` operator associated with the new store to actually push the page to

[2] For most file system types, "pushing" out a page typically entails writing it out to its backing store, marking it as "clean" (i.e. unmodified) and adding it to the free list of pages for re-use.

disk.

This operation must be performed with great care, because during the rename other parts of the system may be accessing or trying to access the page. It is perfectly legitimate for a process to be reading or writing the page during the rename. For example, the existing system already allows a user to access a page while the page is being written out. However, changing the name of the page or its corresponding anon slot is not allowed while I/O is in progress or while clients are using the old name to try to find the page. The page layer provides locks to protect against this happening to the page, however, the name of the page is stored in the anon slot as well. Thus both the anon slot and the page must be renamed atomically to insure that a client will not use the wrong name to get to the page while the rename is in progress. The anon slot contains a lock that protects its backing store name fields, among other things. To guarantee atomicity, both the anon slot and the page are locked, both are renamed, and then the locks are released. Any process trying to manipulate the page in the kernel will acquire one or both of these locks first and will thus be guaranteed a consistent view of the name of the page and its associated anon slot.

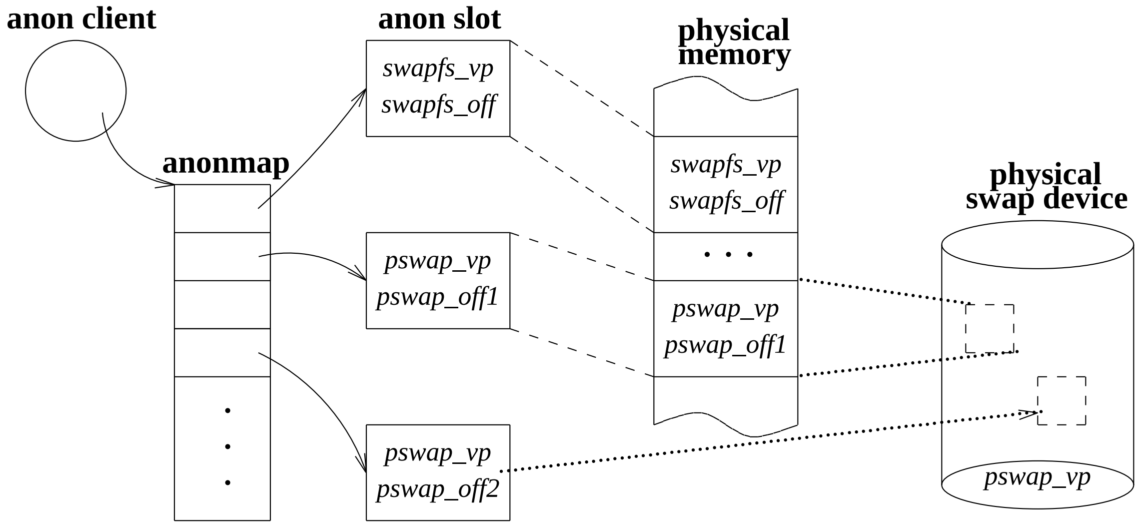

Figure 2. swap renaming

3.4. Summary

To summarise the interaction of swapfs with the rest of the VM system, consider figure 2 which shows some different states of anonymous memory. The first page in the figure (named *swapfs_vp, swapfs_off*) has been allocated and named by swapfs. Note that because it is named by swapfs it is memory resident and has no physical backing store.

The second page in the figure (named *pswap_vp, pswap_off1*) shows an anonymous page that has been renamed to a physical backing store location. This page was originally named by swapfs, until the `putpage()` operator was called to push out the page. At this time swapfs called the swap layer, which allocated a page with backing store on swap device *pswap_vp,* at offset *pswap_off1* This new name was returned to swapfs, which renamed the anonymous page and the slot to (*pswap_vp, pswap_off1*) and then called the `putpage()` operator of *pswap_vp* to write the page to disk.

The third page (named *pswap_vp, pswap_off2*) has also been renamed to physical backing store and the corresponding main memory page has been freed for another use in the system. The data for that anonymous page may be retrieved via the `getpage()` operator for the physical swap device.

Note that in contrast to figure 1, anon slots are no longer explicitly tied to a particular swap device and that the name of the backing store is now stored explicitly in the slot.

4. Performance Discussion

The initial performance goal for this project was that swapfs not degrade overall system performance. In particular we required a system running swapfs with as much physical swap space as the current SunOS default to perform as well as the existing SunOS implementation. Another concern was that configuring a system with small amounts of physical swap relative to the amount of main memory might degrade overall performance as memory became clogged with anonymous pages that could not be freed for reuse. However, it was a pleasant surprise to find that, even with a small percentage of physical swap relative to main memory (as little as 20%), there is very little degradation in system performance on standard benchmarks. This may be explained by the observation that, for a variety of workloads, the system tends to allocate only about 2/3 of the swap space it reserves; thus, even if all of this space is allocated from main memory, a substantial remainder is available to the rest of the system.

5. Future Work

Adding swapfs to the system opens the way to some interesting enhancements. For example, the current pageout daemon pushes out pages one at a time, and many of these pages are backed by swapfs. Slight modifications to the swapfs rename mechanism would provide the capability to rename a batch of pages to a continuous stretch of physical backing store; then all the pages in the batch could be pushed out in a single I/O.

Another interesting enhancement involves the notion of multiple swap vnodes. Although swapfs currently provides anonymous memory names from only one swap vnode, simple modifications to the existing interface could allow each client (e.g., a process) of the anon layer to have its own unique swapfs vnode; backing store allocation requests from a particular client could then always be satisfied from the swapfs vnode belonging to that client. This, in turn, could provide a basis for client-oriented clustering of anonymous pages on physical backing store.

6. Conclusions

Swapfs is a virtual swap file system that can, on demand, dynamically rename anonymous pages and push them out to physical backing store. This capability allows the system to treat main memory as swap space, but also allows it to reclaim this memory by pushing pages to available physical swap space when contention for memory increases.

7. Acknowledgements

We would especially like to acknowledge Bill Shannon who contributed many of the ideas that are central to this work. He and other members of the Systems Group at Sun including Glenn Skinner, Anil Shivalingiah, and Dock Williams, also provided many helpful suggestions.

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*, pp. 81-94, 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*, pp. 238-241, USENIX Association, Atlanta Georgia, USA, June 1986.

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

Rosenthal1990.
Davis S. H. Rosenthal, ‘‘Evolving the Vnode Interface,’’ *Proceedings of the Summer 1990 USENIX Conference*, pp. 107-117, USENIX Association, Anaheim, California USA, June 1990.

Snyder1990.

Peter Snyder, ‘‘tmpfs: A Virtual Memory File System,’’ *Proceedings of the Autumn 1990 EUUG Conference*, pp. 241-248, EUUG, Nice France, October 1990.