14

File System Framework

From its inception, UNIX has been built around two fundamental entities: *processes* and *files*. In this chapter, we look at the implementation of files in Solaris and discuss the framework for file systems.

14.1 File System Framework

Solaris OS includes a framework, the *virtual file system framework*, under which multiple file system types are implemented. Earlier implementations of UNIX used a single file system type for all of the mounted file systems, typically, the UFS file system from BSD UNIX. The virtual file system framework was developed to allow Sun's distributed computing file system (NFS) to coexist with the UFS file system in SunOS 2.0; it became a standard part of System V in SVR4 and Solaris OS. We can categorize Solaris file systems into the following types:

- **Storage-based.** Regular file systems that provide facilities for persistent storage and management of data. The Solaris UFS and PC/DOS file systems are examples.
- **Network file systems.** File systems that provide files that are accessible in a local directory structure but are stored on a remote network server; for example, NFS.
- **Pseudo file systems.** File systems that present various abstractions as files in a file system. The `/proc` pseudo file system represents the address space of a process as a series of files.

The framework provides a single set of well-defined interfaces that are file system independent; the implementation details of each file system are hidden behind these interfaces. Two key objects represent these interfaces: the virtual file, or *vnode,* and the virtual file system, or *vfs* objects. The `vnode` interfaces implement file-related functions, and the `vfs` interfaces implement file system management functions. The `vnode` and `vfs` interfaces direct functions to specific file systems, depending on the type of file system being operated on. Figure 14.1 shows the file system layers. File-related functions are initiated through a system call or from another kernel subsystem and are directed to the appropriate file system by the `vnode/vfs` layer.

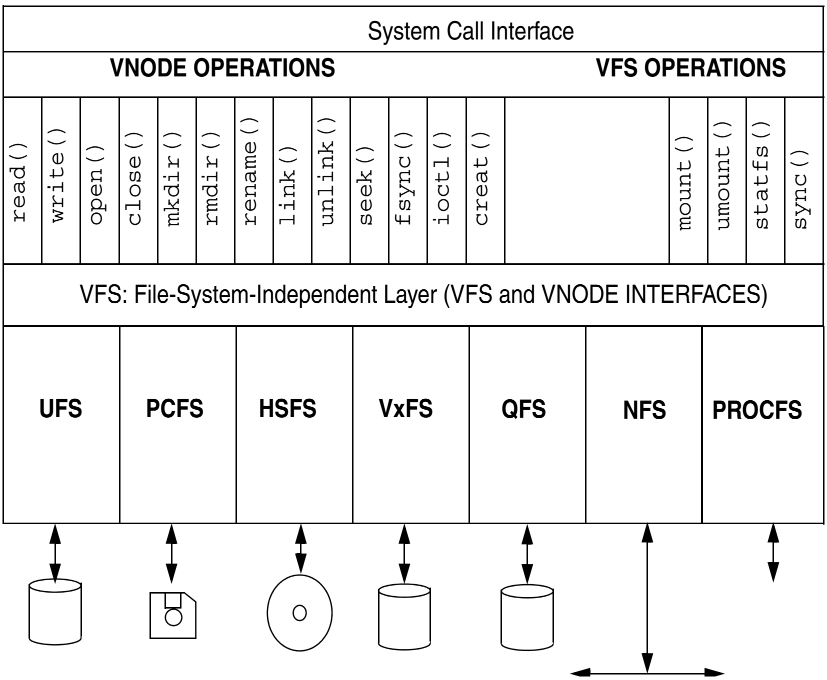

Figure 14.1 Solaris File System Framework

14.2 Process-Level File Abstractions

Within a process, a file is referenced through a *file descriptor.* An integer space of file descriptors per process is shared by multiple threads within each process. A file descriptor is a value in an integer space. It is assigned when a file is first opened and freed when a file is closed.

Each process has a list of active file descriptors, which are an index into a *per-process file table*. Each file table entry holds process-specific data including the current file's seek offset and has a reference to a systemwide virtual file node (`vnode`). The list of open file descriptors is kept inside a process's user area (struct user) in an `fi_list` array indexed by the file descriptor number. The `fi_list` is an array of `uf_entry_t` structures, each with its own lock and a pointer to the corresponding `file_t` file table entry.

Although multiple file table entries might reference the same file, there is a single vnode entry, as Figure 14.2 highlights. The `vnode` holds systemwide information about a file, including its type, size, and containing file system.

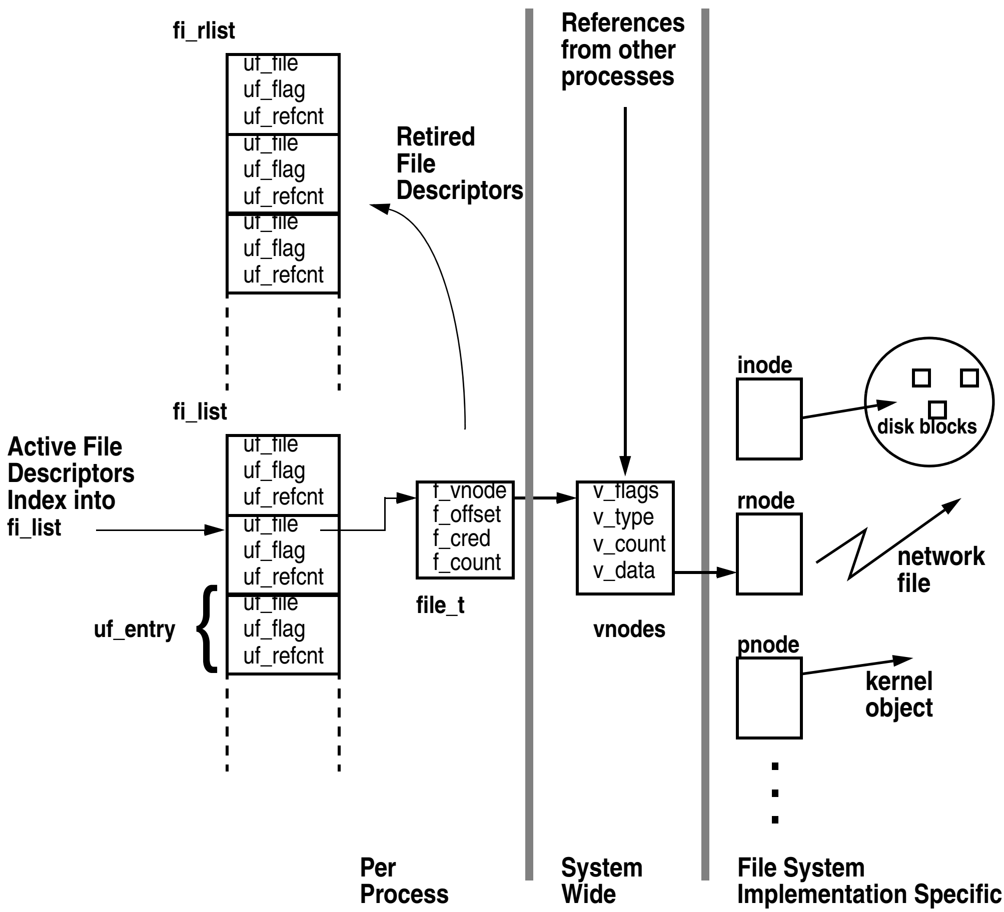

Figure 14.2 Structures Used for File Access

14.2.1 File Descriptors

A *file descriptor* is a non-negative integer that is returned from the system calls `open()`, `fcntl()`, `pipe()`, or `dup()`. A process uses the file descriptor on other system calls, such as `read()` and `write()`, that perform operations on open files. Each file descriptor is represented by a `uf_entry_t`, shown below, and the file descriptor is used as an index into an array of `uf_entry_t` entries.

```
/*
 * Entry in the per-process list of open files.
 * Note: only certain fields are copied in flist_grow() and flist_fork().
 * This is indicated in brackets in the structure member comments.
 */
typedef struct uf_entry {
        kmutex_t        uf_lock;        /* per-fd lock [never copied] */
        struct file     *uf_file;       /* file pointer [grow, fork] */
        struct fpollinfo *uf_fpollinfo; /* poll state [grow] */
        int             uf_refcnt;      /* LWPs accessing this file [grow] */
        int             uf_alloc;       /* right subtree allocs [grow, fork] */
        short           uf_flag;        /* fcntl F_GETFD flags [grow, fork] */
        short           uf_busy;        /* file is allocated [grow, fork] */
        kcondvar_t      uf_wanted_cv;   /* waiting for setf() [never copied] */
        kcondvar_t      uf_closing_cv;  /* waiting for close() [never copied] */
        struct portfd   *uf_portfd;     /* associated with port [grow] */
        /* Avoid false sharing - pad to coherency granularity (64 bytes) */
        char            uf_pad[64 - sizeof (kmutex_t) - 2 * sizeof (void*) -
                2 * sizeof (int) - 2 * sizeof (short) -
                2 * sizeof (kcondvar_t) - sizeof (struct portfd *)];
} uf_entry_t;
                                        See usr/src/uts/common/sys/user.h
```

The file descriptor list is anchored in the process's user area in the `uf_info_t` structure pointed to by `u_finfo`.

```
typedef struct  user {
...
        uf_info_t       u_finfo;        /* open file information */
} user_t;

/*
 * Per-process file information.
 */
typedef struct uf_info {
        kmutex_t        fi_lock;        /* see below */
        kmutex_t        fi_pad;         /* unused -- remove in next release */
        int             fi_nfiles;      /* number of entries in fi_list[] */
        uf_entry_t *volatile fi_list;   /* current file list */
        uf_rlist_t      *fi_rlist;      /* retired file lists */
} uf_info_t;
                                        See usr/src/uts/common/sys/user.h
```

There are two lists of file descriptor entries in each process: an *active set* (fi_list) and a *retired set* (fi_rlist). The active set contains all the current file descriptor entries (open and closed), each of which points to a corresponding file_t file table entry. The retired set is used when the fi_list array is resized; as part of a lock-less find algorithm, once file_t entries are allocated, they are never unallocated, so pointers to file_t entries are always valid. In this manner, the algorithm need only lock the fi_list during resize, making the common case (find) fast and scalable.

14.2.2 The open Code Path

As an example, a common path through file descriptor and file allocation is through the open() system call. The open() system call returns a file descriptor to the process for a given path name. The open() system call is implemented by copen (common open), which first allocates a new file_t structure from the file_cache kernel allocator cache. The algorithm then looks for the next available file descriptor integer within the process's allocate fd integer space by using fd_find(), and reserves it. With an fd in hand, the lookup routine parses the "/"-separated components, calling the file-system-specific lookup function for each. After all path-name components are resolved, the vnode for the path is returned and linked into the file_t file table entry. The file-system-specific open function is called to increment the vnode reference count and do any other per-vnode open handling (typically very little else, since the majority of the open is done in lookup rather than the file systems' open() function. Once the file table handle is set up, it is linked into the process's file descriptor fi_list array and locked.

```
-> copen                                Common entry point from open(2)
  -> falloc                             Allocate a per-process file_t file table entry
    -> ufalloc_file                     Allocate a file descriptor uf_entry
      -> fd_find                        Find the next available fd integer
      <- fd_find
      -> fd_reserve                     Reserve the fd integer
      <- fd_reserve
    <- ufalloc_file
  <- falloc
  -> fop_lookup                         Look up the file name supplied in open()
    -> ufs_lookup                       In this case, get UFS to do the hard work
      -> dnlc_lookup                    Check in the DNLC
      <- dnlc_lookup
    <- ufs_lookup                       Return a vnode to copen()
  <- fop_lookup
  -> fop_open                           Call the file system specific open function
    -> ufs_open                         Bump ref count in vnode, etc...
    <- ufs_open
  <- fop_open
  -> setf                               Lock the processes uf_entry for this file
  <- setf
<- copen                                All done
```

14.2.3 Allocating and Deallocating File Descriptors

One of the central functions is that of managing the file descriptor integer space. The `fd_find(file_t *, int minfd)` and `fd_reserve()` functions are the primary interface into the file descriptor integer space management code. The `fd_find()` function locates the next lowest available file descriptor number, starting with `minfd`, to support `fcntl(fd, F_DUPFD, minfd)`. The `fd_reserve()` function either reserves or unreserves an entry by passing a 1 or –1 as an argument.

Beginning with Solaris 8, a significantly revised algorithm manages the integer space. The file descriptor integer space is a binary tree of per-process file entries (`uf_entry`) structures.

The algorithm is as follows. Keep all file descriptors in an infix binary tree in which each node records the number of descriptors allocated in its right subtree, including itself. Starting at `minfd`, ascend the tree until a non-fully allocated right subtree is found. Then descend that subtree in a binary search for the smallest `fd`. Finally, ascend the tree again to increment the allocation count of every subtree containing the newly allocated `fd`. Freeing an `fd` requires only the last step: Ascend the tree to decrement allocation counts. Each of these three steps (ascent to find non-full subtree, descent to find lowest `fd`, ascent to update allocation counts) is O(log n); thus the algorithm as a whole is O(log n).

We don't implement the `fd` tree by using the customary left/right/parent pointers, but instead take advantage of the glorious mathematics of full infix binary trees. For reference, here's an illustration of the logical structure of such a tree, rooted at 4 (binary 100), covering the range 1–7 (binary 001–111). Our canonical trees do not include `fd` 0; we deal with that later.

We make the following observations, all of which are easily proven by induction on the depth of the tree:

- **(T1).** The lowest-set bit (LSB) of any node is equal to its level in the tree. In our example, nodes 001, 011, 101, and 111 are at level 0; nodes 010 and 110 are at level 1; and node 100 is at level 2 (see Figure 14.3).

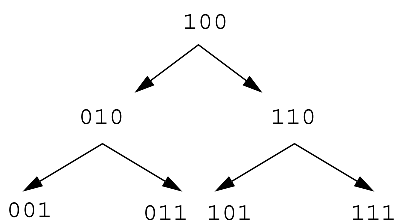

Figure 14.3 File Descriptor Integer Space as an *Infix* Binary Tree

- **(T2).** The child size (CSIZE) of node N—that is, the total number of right-branch descendants in a child of node N, including itself—is given by clearing all but the lowest-set bit of N. This follows immediately from (T1). Applying this rule to our example, we see that CSIZE(100) = 100, CSIZE(x10) = 10, and CSIZE(xx1) = 1.
- **(T3).** The nearest left ancestor (LPARENT) of node N—that is, the nearest ancestor containing node N in its right child—is given by clearing the LSB of N. For example, LPARENT(111) = 110 and LPARENT(110) = 100. Clearing the LSB of nodes 001, 010, or 100 yields zero, reflecting the fact that these are leftmost nodes. Note that this algorithm automatically skips generations as necessary. For example, the parent of node 101 is 110, which is a *right* ancestor (not what we want); but its grandparent is 100, which is a left ancestor. Clearing the LSB of 101 gets us to 100 directly, skipping right past the uninteresting generation (110).

  Note that since LPARENT clears the LSB, whereas CSIZE clears all *but* the LSB, we can express LPARENT() nicely in terms of CSIZE():

  $$LPARENT(N) = N - CSIZE(N)$$

- **(T4).** The nearest right ancestor (RPARENT) of node N is given by

  $$RPARENT(N) = N + CSIZE(N)$$

- **(T5).** For every interior node, the children differ from their parent by CSIZE(parent) / 2. In our example, CSIZE(100) / 2 = 2 = 10 binary, and indeed, the children of 100 are 100 ± 10 = 010 and 110.

Next, we need a few two's-complement math tricks. Suppose a number, N, has the following form:

$$N = xxxx10\ldots0$$

That is, the binary representation of N consists of some string of bits, then a 1, then all 0's. This amounts to nothing more than saying that N has a lowest-set bit, which is true for any $N \neq 0$. If we look at N and $N - 1$ together, we see that we can combine them in useful ways:

$$N - 1 = xxxx01\ldots1:$$

$$\begin{aligned} N \mathbin{\&} (N-1) &= xxxx000000 \\ N \mid (N-1) &= xxxx111111 \\ N \wedge (N-1) &= \quad\;\; 111111 \end{aligned}$$

In particular, this suggests several easy ways to clear all but the LSB, which by (T2) is exactly what we need to determine CSIZE(N) = 10...0. We opt for this formulation:

$$(C1)\ CSIZE(N) = (N - 1)\ \hat{}\ (N \mid (N - 1))$$

Similarly, we have an easy way to determine LPARENT(N), which requires that we clear the LSB of N:

$$(L1)\ LPARENT(N) = N\ \&\ (N - 1)$$

We note in the above relations that (N | (N – 1)) – N = CSIZE(N) – 1. When combined with (T4), this yields an easy way to compute RPARENT(N):

$$(R1)\ RPARENT(N) = (N \mid (N - 1)) + 1$$

Finally, to accommodate `fd` 0, we must adjust all of our results by ± 1 to move the `fd` range from [1, 2^n) to [0, 2^n – 1). This is straightforward, so there's no need to belabor the algebra; the revised relations become

$$(C1a)\ CSIZE(N) = N\ \hat{}\ (N \mid (N + 1))$$
$$(L1a)\ LPARENT(N) = (N\ \&\ (N + 1)) - 1$$
$$(R1a)\ RPARENT(N) = N \mid (N + 1)$$

This completes the mathematical framework. We now have all the tools we need to implement `fd_find()` and `fd_reserve()`.

The `fd_find(fip, minfd)` function finds the smallest available file descriptor ≥ `minfd`. It does not actually allocate the descriptor; that's done by `fd_reserve()`.

`fd_find()` proceeds in two steps:

1. Find the leftmost subtree that contains a descriptor ≥ `minfd`. We start at the right subtree rooted at `minfd`. If this subtree is not full—if `fip->fi_list[minfd].uf_alloc != CSIZE(minfd)`—then step 1 is done. Otherwise, we know that all `fd`s in this subtree are taken, so we ascend to RPARENT`(minfd)` using (R1a). We repeat this process until we either find a candidate subtree or exceed `fip->fi_nfiles`. We use (C1a) to compute CSIZE().
2. Find the smallest `fd` in the subtree discovered by step 1. Starting at the root of this subtree, we descend to find the smallest available `fd`. Since the left children have the smaller `fd`s, we descend rightward only when the left child is full.

We begin by comparing the number of allocated `fd`s in the root to the number of allocated `fd`s in its right child; if they differ by exactly CSIZE(child), we know the left subtree is full, so we descend right; that is, the right child becomes the search root. Otherwise, we leave the root alone and start following the right child's left children. As fortune would have it, this is simple computationally: by (T5), the right child of `fd` is just `fd` + size, where size = CSIZE`(fd)` / 2. Applying (T5) again, we find that the right child's left child is `fd` + size – (size / 2) = `fd` + (size / 2); *its* left child is `fd` + (size / 2) – (size / 4) = `fd` + (size / 4), and so on. In general, `fd`'s right child's leftmost nth descendant is `fd` + (size >> n). Thus, to follow the right child's left descendants, we just halve the size in each iteration of the search.

When we descend leftward, we must keep track of the number of `fd`s that were allocated in all the right subtrees we rejected so that we know how many of the root `fd`'s allocations are in the remaining (as yet unexplored) leftmost part of its right subtree. When we encounter a fully allocated left child—that is, when we find that `fip->fi_list[fd].uf_alloc == ralloc + size`—we descend right (as described earlier), resetting `ralloc` to zero.

The `fd_reserve(fip, fd, incr)` function either allocates or frees `fd`, depending on whether `incr` is 1 or –1. Starting at `fd`, `fd_reserve()` ascends the leftmost ancestors (see (T3)) and updates the allocation counts. At each step we use (L1a) to compute LPARENT(), the next left ancestor.

14.2.4 File Descriptor Limits

Each process has a hard and soft limit for the number of files it can have opened at any time; these limits are administered through the Resource Controls infrastructure by `process.max-file-descriptor` (see Section 7.5 for a description of Resource Controls). The limits are checked during `falloc()`. Limits can be viewed with the `prctl` command.

```
sol9$ prctl -n process.max-file-descriptor $$
process: 21471: -ksh
NAME    PRIVILEGE       VALUE    FLAG   ACTION                       RECIPIENT
process.max-file-descriptor
        basic             256       -   deny                             21471
        privileged      65.5K       -   deny                                 -
        system          2.15G     max   deny                                 -
```

If no resource controls are set for the process, then the defaults are taken from system tuneables; `rlim_fd_max` is the hard limit, and `rlim_fd_cur` is the current limit (or soft limit). You can set these parameters systemwide by placing entries in the `/etc/system` file.

```
set rlim_fd_max=8192
set rlim_fd_cur=1024
```

14.2.5 File Structures

A kernel object cache segment is allocated to hold file structures, and they are simply allocated and linked to the process and `vnode` as files are created and opened.

We can see in Figure 14.2 that each process uses file descriptors to reference a file. The file descriptors ultimately link to the kernel file structure, defined as a `file_t` data type, shown below.

```
/*
 * fio locking:
 *   f_rwlock   protects f_vnode and f_cred
 *   f_tlock    protects the rest
 *
 *   The purpose of locking in this layer is to keep the kernel
 *   from panicking if, for example, a thread calls close() while
 *   another thread is doing a read().  It is up to higher levels
 *   to make sure 2 threads doing I/O to the same file don't
 *   screw each other up.
 */
/*
 * One file structure is allocated for each open/creat/pipe call.
 * Main use is to hold the read/write pointer associated with
 * each open file.
 */
typedef struct file {
        kmutex_t        f_tlock;        /* short term lock */
        ushort_t        f_flag;
        ushort_t        f_pad;          /* Explicit pad to 4 byte boundary */
        struct vnode    *f_vnode;       /* pointer to vnode structure */
        offset_t        f_offset;       /* read/write character pointer */
        struct cred     *f_cred;        /* credentials of user who opened it */
        struct f_audit_data     *f_audit_data;  /* file audit data */
        int             f_count;        /* reference count */
} file_t;
                                                       See usr/src/uts/common/sys/file.h
```

The fields maintained in the file structure are, for the most part, self-explanatory. The `f_tlock` kernel mutex lock protects the various structure members. These include the `f_count` reference count, which lists how many file descriptors reference this structure, and the `f_flag` file flags.

Since files are allocated from a systemwide kernel allocator cache, you can use MDB's `::kmastat` dcmd to look at how many files are opened systemwide. The `sar` command also shows the same information in its `file-sz` column.

This example shows 1049 opened files. The format of the `sar` output is a holdover from the early days of static tables, which is why it is displayed as 1049/1049. Originally, the value on the left represented the current number of occupied table slots, and the value on the right represented the maximum number of slots. Since file structure allocation is completely dynamic in nature, both values will always be the same.

```
sol8# mdb -k
> ::kmastat !grep file
cache                        buf    buf    buf    memory     alloc alloc
name                        size in use  total    in use   succeed  fail
------------------------- ------ ------ ------ --------- --------- -----
file_cache                    56   1049   1368     77824   9794701     0

# sar -v 3 333

SunOS ozone 5.10 Generic i86pc    07/13/2005

17:46:49  proc-sz    ov  inod-sz    ov  file-sz    ov   lock-sz
17:46:52  131/16362    0 8884/70554    0 1049/1049    0    0/0
17:46:55  131/16362    0 8884/70554    0 1049/1049    0    0/0
```

We can use MDB's `::pfiles` dcmd to explore the linkage between a process and file table entries.

```
sol8# mdb -k
> 0t1119::pid2proc
ffffffff83135890
> ffffffff83135890::pfiles -fp
            FILE   FD     FLAG            VNODE     OFFSET             CRED  CNT
ffffffff85ced5e8    0        1 ffffffff857c8580          0 ffffffff83838a40    1
ffffffff85582120    1        2 ffffffff857c8580          0 ffffffff83838a40    2
ffffffff85582120    2        2 ffffffff857c8580          0 ffffffff83838a40    2
ffffffff8362be00    3     2001 ffffffff836d1680          0 ffffffff83838c08    1
ffffffff830d3b28    4        2 ffffffff837822c0          0 ffffffff83838a40    1
ffffffff834aacf0    5        2 ffffffff83875a80         33 ffffffff83838a40    1
> ffffffff8362be00::print file_t
{
    f_tlock = {
        _opaque = [ 0 ]
    }
    f_flag = 0x2001
    f_pad = 0xbadd
    f_vnode = 0xffffffff836d1680
    f_offset = 0
    f_cred = 0xffffffff83838c08
    f_audit_data = 0
    f_count = 0x1
}
> 0xffffffff836d1680::vnode2path
/zones/gallery/root/var/run/name_service_door
```

For a specific process, we use the `pfiles(1)` command to create a list of all the files opened.

```
sol8$ pfiles 1119
1119:   /usr/lib/sendmail -Ac -q15m
  Current rlimit: 1024 file descriptors
   0: S_IFCHR mode:0666 dev:281,2 ino:16484 uid:0 gid:3 rdev:13,2
      O_RDONLY
      /zones/gallery/root/dev/null
```

continues

```
    1: S_IFCHR mode:0666 dev:281,2 ino:16484 uid:0 gid:3 rdev:13,2
       O_WRONLY
       /zones/gallery/root/dev/null
    2: S_IFCHR mode:0666 dev:281,2 ino:16484 uid:0 gid:3 rdev:13,2
       O_WRONLY
       /zones/gallery/root/dev/null
    3: S_IFDOOR mode:0444 dev:279,0 ino:34 uid:0 gid:0 size:0
       O_RDONLY|O_LARGEFILE FD_CLOEXEC  door to nscd[762]
       /zones/gallery/root/var/run/name_service_door
    4: S_IFCHR mode:0666 dev:281,2 ino:16486 uid:0 gid:3 rdev:21,0
       O_WRONLY FD_CLOEXEC
       /zones/gallery/root/dev/conslog
    5: S_IFREG mode:0600 dev:102,198 ino:11239 uid:25 gid:25 size:33
       O_WRONLY
       /zones/gallery/root/var/spool/clientmqueue/sm-client.pid
```

In the preceding examples, the `pfiles` command is executed on PID 1119. The PID and process name are dumped, followed by a listing of the process's opened files. For each file, we see a listing of the file descriptor (the number to the left of the colon), the file type, file mode bits, the device from which the file originated, the inode number, file UID and GID, and the file size.

14.3 Solaris File System Framework

The `vnode/vfs` interfaces—the "top end" of the file system module—implement vnode and vfs objects. The "bottom end" of the file system uses other kernel interfaces to access, store, and cache the data they represent. Disk-based file systems interface to device drivers to provide persistent storage of their data. Network file systems access remote storage by using the networking subsystem to transmit and receive data. Pseudo file systems typically access local kernel functions and structures to gather the information they represent.

- **Loadable file system modules.** A dynamically loadable module type is provided for Solaris file systems. File system modules are dynamically loaded at the time each file system type is first mounted (except for the root file system, which is mounted explicitly at boot).
- **The `vnode` interface.** As discussed, this is a unified file-system-independent interface between the operating system and a file system implementation.
- **File system caching.** File systems that implement caching interface with the virtual memory system to map, unmap, and manage the memory used for caching. File systems use physical memory pages and the virtual memory system to cache files. The kernel's `seg_map` driver maps file system cache

into the kernel's address space when accessing the file system through the `read()` and `write()` system calls. (See Section 14.8.1.)

- **Path-name management.** Files are accessed by means of path names, which are assembled as a series of directory names and file names. The file system framework provides routines that resolve and manipulate path names by calling into the file system's `lookup()` function to convert paths into `vnode` pointers.
- **Directory name caching.** A central directory name lookup cache (DNLC) provides a mechanism to cache pathname-to-`vnode` mappings, so that the directory components need not be read from disk each time they are needed.

14.3.1 Evolution of the File System Framework

Solaris 10 introduces a new file system interface that significantly improves the portability of file systems. In prior releases of Solaris OS, the `vnode` and `vfs` structures were entirely visible to their consumers. A file system client would reference, manipulate, or update raw `vfs` and `vnode` structure members directly, which meant that file systems had operating system revision-specific assumptions compiled into them. Whenever the `vfs` or `vnode` structures changed in the Solaris kernel, file systems would need to be recompiled to match the changes. The new interface allows the `vnode` structures to change in many ways without breaking file system compatibility.

The new model replaces the old file system VOP macros with a new set of functions. The goals of the new interface are as follows:

- It separates the `vnode` from FS-dependent node so that changes in the `vnode` structure that affect its size do not affect the size of other data structures.
- It provides interfaces to access nonpublic `vnode` structure members.
- It delivers a flexible operation registration mechanism that provides appropriate defaults for unspecified operations and allows the developer to specify a corresponding default or error routine.
- It delivers a flexible mechanism to invoke vnode/`vfs` operations without requiring the client module to have knowledge of how the operations are stored.
- It provides a facility for creation, initialization, and destruction of vnodes.
- It provides accessor functions for file systems that require information on the following characteristics of a vnode: existence of locks, existence of cached data, read-only attribute.

The following major changes have been made to the file system interface as part of this project:

- The following related vnode fields are now private: v_filocks, v_shrlocks, v_nbllock, v_pages and v_cv.
- Support routines allow a vnode/vfs client to set a vnode's or vfs's operations, retrieve the operations, compare the operations vector to a given value, compare a specific operation in the operations vector to a given value. The related vnode field v_op and the related vfs field vfs_op should not be directly accessed by file systems.
- An accessor routine returns a pointer to the vfs, if any, which may be mounted on a given vnode. Another routine determines whether a given vnode is mounted on. The related vnode field v_vfsmountedhere is now private.
- An operation registration mechanism can fill in default operation values (if appropriate) for operations that are not explicitly specified by the file system.
- The operation registration mechanism enables developers to add new operations to a new (updated) version of Solaris OS without requiring existing file systems to support those new operations, provided that the new operations have system-defined defaults.
- The file system module loading mechanism is updated to enable these changes.
- Vnodes are no longer embedded in file system data structures (for example, inodes).
- The following functions have been added to support the separation of the vnode from the FS-dependent node: vn_alloc(), vn_free(), and vn_reinit().
- Certain fields in the vnode have been made "private" to satisfy the requirements of other projects. Also, the fields in the vnode have been rearranged to put the "public" structure members at the top and the private members at the bottom.
- File systems now register their vnode and vfs operations by providing an operation definition table that specifies operations by using name/value pairs.
- The VOP and VFSOP macros no longer directly dereference the vnode and vfs structures and their operations tables. They each call corresponding functions that perform that task.
- File system module loading no longer takes a vfs switch entry. Instead, it takes a vfsdef structure that is similar. The difference is that the vfsdef structure includes a version number but does not include a vfsops table.

The following accessor functions have been added to provide information about the state and characteristics of a vnode.

- **vn_is_readonly().** Returns non-zero if the vnode is on a read-only file system.
- **vn_has_flocks().** Returns non-zero if the vnode has active file locks.
- **vn_has_mandatory_locks().** Returns non-zero if the vnode has mandatory locks.
- **vn_has_cached_data().** Returns non-zero if the vnode has pages in the page cache.
- **vn_mountedvfs().** Returns the `vfs` mounted on this `vnode`, if any.
- **vn_ismntpt().** Returns true (non-zero) if this `vnode` is mounted on, zero otherwise.

New interfaces have been developed to register `vnode` and `vfs` operations.

- **vn_make_ops().** Creates and builds the private `vnodeops` table.
- **vn_freevnodeops().** Frees a `vnodeops` structure created by `vn_make_ops()`.
- **vfs_setfsops().** Builds a `vfsops` table and associates it with a `vfs` switch table entry.
- **vfs_freevfsops_by_type().** Frees a `vfsops` structure created by `vfs_makefsops()`.
- **vfs_makefsops().** Creates and builds (dummy) `vfsops` structures.
- **vfs_freevfsops().** Frees a `vfsops` structure created by `vfs_makefsops()`.

The following support routines have been developed to set and provide information about the `vnode`'s operations vector.

- **vn_setops().** Sets the operations vector for this `vnode`.
- **vn_getops().** Retrieves the operations vector for this `vnode`.
- **vn_matchops().** Determines if the supplied operations vector matches the `vnode`'s operations vector. Note that this is a "shallow" match. The pointer to the operations vector is compared, not each individual operation.
- **vn_matchopval().** Determines if the supplied function exists for a particular operation in the `vnode`'s operations vector.

The following support routines have been developed to set and provide information about the `vfs`'s operations vector.

- **`vfs_setops()`.** Sets the operations vector for this `vfs`.
- **`vfs_getops()`.** Retrieves the operations vector for this `vfs`.
- **`vfs_matchops()`.** Determines if the supplied operations vector matches the `vfs`'s operations vector. Note that this is a "shallow" match. The pointer to the operations vector is compared, not each individual operation.
- **`vfs_can_sync()`.** Determines if a `vfs` has an FS-supplied (nondefault, non-error) sync routine.

14.3.2 The Solaris File System Interface

The file system interface can be categorized into three major parts:

- A single systemwide, file system module-specific declaration
- A per-file system mount instance declaration
- A set of per-file operations with each file system mount instance

14.4 File System Modules

A file system is implemented as a dynamically loadable kernel module. Each file system declares the standard module `_init`, `_info`, and `_fini` entry points, which are used to install and remove the file system within the running kernel instance.

The primary descriptive entry for each file system is provided by a static declaration of a `vfsdef_t`, which includes the following:

- The version of the `vfs` interface used at module compile time (by specification of `VFSDEF_VERSION`).
- The name of the file system (a string).
- The global initialization function to be called when the file system module is loaded. Although a file system module is typically loaded on the first mount, a module can be loaded `modload(1M)` without mounting a file system.
- A set of options that can be set at mount time.

```
static mntopt_t tmpfs_options[] = {
        /* Option name          Cancel Opt      Arg     Flags           Data */
        { MNTOPT_XATTR,         xattr_cancel,   NULL,   MO_DEFAULT,     NULL},
        { MNTOPT_NOXATTR,       noxattr_cancel, NULL,   NULL,           NULL},
        { "size",               NULL,           "0",    MO_HASVALUE,    NULL}
};

static mntopts_t tmpfs_proto_opttbl = {
        sizeof (tmpfs_options) / sizeof (mntopt_t),
        tmpfs_options
};

static vfsdef_t vfw = {
        VFSDEF_VERSION,
        "tmpfs",
        tmpfsinit,
        VSW_HASPROTO,
        &tmpfs_proto_opttbl
};
```
See usr/src/uts/common/fs/tmpfs/tmp_vfsops.c

14.4.1 Interfaces for Mount Options

The `options` template is used to accept and validate options at mount time. A standard set is defined in `sys/vfs.h`, but you can add your own by simply supplying a string (as `tmpfs` does for size).

The `mntopts_t` struct (usually called the mount options table) consists of a count of the number of options and an array of `options` structures of length count. Each file system should define a prototype mount options table that will be used by the `vfs_initopttbl()` function to initialize the working mount options table for each mount instance. The text below describes the initialization of the prototype mount options table. The `vfs_initopttbl()` function should be used to initialize working mount options tables from the prototype mount options table.

```
typedef struct mntopts {
        int             mo_count;       /* number of entries in table */
        mntopt_t        *mo_list;       /* list of mount options */
} mntopts_t;
```
See usr/src/uts/common/sys/vfs.h

Each mount option contains fields to drive the parser and fields to accept the results of the parser's execution. Here is the structure that defines an individual option in the mount options table.

```
typedef struct mntopt {
        char    *mo_name;       /* option name */
        char    **mo_cancel;    /* list of options cancelled by this one */
        char    *mo_arg;        /* argument string for this option */
        int     mo_flags;       /* flags for this mount option */
        void    *mo_data;       /* file system specific data */
} mntopt_t;
```
See usr/src/uts/common/sys/vfs.h

Each option must have a string that gives the name of the option. Additionally, if an option is one that invalidates other options, the `mo_cancel` field points to a `NULL`-terminated list of names of options to turn off if this option is recognized. If an option accepts an argument (that is, it is of the form `opt=arg`), then the `mo_arg` field should be initialized with the string that is the default for the argument (if it has a default value; otherwise `NULL`). During option parsing, the parser will then replace the string in the working mount options table with the string provided by the user if the option is recognized during option parsing. The following flags are recognized by or set by the parser during option parsing.

- **`MO_NODISPLAY`.** Option will not be listed in mounted file system table.
- **`MO_HASVALUE`.** Option is expected to have an argument (that is, of form `opt = arg`)
- **`MO_IGNORE`.** Option is ignored by the parser and will not be set even if seen in the options string. (Can be set manually with `vfs_setmntopt` function.)
- **`MO_DEFAULT`.** Option is set on by default and will show in `mnttab` even if not seen by parser in options string.

The `mo_data` field is for use by a file system to hold any option-specific data it may wish to make use of.

14.4.2 Module Initialization

A standard file system module will provide a module `_init` function and register an initialization function to be called back by the file system module-loader facility. The following example shows the initialization linkage between the module declaration and the file-system-specific initialization function.

```
static vfsdef_t vfw = {
        VFSDEF_VERSION,
        "tmpfs",
        tmpfsinit,
        VSW_HASPROTO,
        &tmpfs_proto_opttbl
};

/*
 * Module linkage information
 */
```

continues

```
static struct modlfs modlfs = {
        &mod_fsops, "filesystem for tmpfs", &vfw
};

static struct modlinkage modlinkage = {
        MODREV_1, &modlfs, NULL
};

int
_init()
{
        return (mod_install(&modlinkage));
}

/*
 * initialize global tmpfs locks and such
 * called when loading tmpfs module
 */
static int
tmpfsinit(int fstype, char *name)
{
...
}
                                    See usr/src/uts/common/fs/tmpfs/tmp_vfsops.c
```

The module is automatically loaded by the first invocation of `mount(2)` (typically from a `mount` command). Upon module load, the `_init()` function of the file system is called; this function completes its self-install with `mod_install()`, which subsequently calls the file system init function (`tmpfsinit()` in this example) defined in the `vfsdef_t`.

Note that file systems no longer need to create and install a `vfs` switch entry; this is done automatically by the module loading using the information supplied in the `vfsdef_t`.

14.5 The Virtual File System (vfs) Interface

The `vfs` layer provides an administrative interface into the file system to support commands like `mount` and `umount` in a file-system-independent manner. The interface achieves independence by means of a virtual file system (`vfs`) object. The `vfs` object represents an encapsulation of a file system's state and a set of methods for each of the file system administrative interfaces. Each file system type provides its own implementation of the object. Figure 14.4 illustrates the `vfs` object. A set of support functions provides access to the contents of the `vfs` structure; file systems should not directly modify the `vfs` object contents.

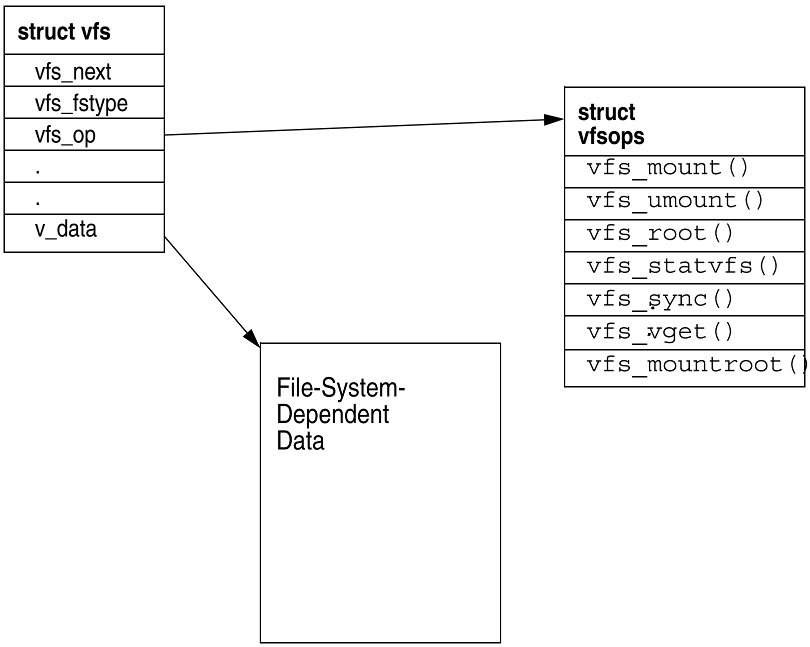

Figure 14.4 The vfs Object

14.5.1 `vfs` Methods

The methods within the file system implement operations on behalf of the common operating system code. For example, given a pointer to a `tmpfs`'s `vfs` object, the generic `VFS_MOUNT()` call will invoke the appropriate function in the underlying file system by calling the `tmpfs_mount()` method defined within that instance of the object.

```
#define VFS_MOUNT(vfsp, mvp, uap, cr) fsop_mount(vfsp, mvp, uap, cr)

int
fsop_mount(vfs_t *vfsp, vnode_t *mvp, struct mounta *uap, cred_t *cr)
{
        return (*(vfsp)->vfs_op->vfs_mount)(vfsp, mvp, uap, cr);
}
                                        See usr/src/uts/common/sys/vfs.h
```

A file system declares its vfs methods through a call to vfs_setfsops(). A template provides allows a selection of methods to be defined, according to Table 14.1.

Table 14.1 Solaris 10 vfs Interface Methods from sys/vfs.h

Method	Description
VFS_MOUNT	Mounts a file system on the supplied vnode. The file-system-dependent part of mount includes these actions. • Determine if mount device is appropriate. • Prepare mount device (e.g., flush pages/blocks). • Read file-system-dependent data from mount device. • Sanity-check file-system-dependent data. • Create/initialize file-system-dependent kernel data structures. • Reconcile any transaction devices.
VFS_UNMOUNT	Unmounts the file system. The file-system-dependent part of unmount includes these actions. • Lock out new transactions and complete current transactions. • Flush data to mount device. • Close down any helper threads. • Tear down file-system-dependent kernel data structures.
VFS_ROOT	Finds the root vnode for a file system.
VFS_STATVFS	Queries statistics on a file system.
VFS_SYNC	Flushes the file system cache.
VFS_VGET	Finds a vnode that matches a unique file ID.
VFS_MOUNTROOT	Mounts the file system on the root directory.
VFS_FREEVFS	Calls back to free resources after last unmount. NFS appears to be the only one that needs this. All others default to fs_freevfs(), which is a no-op.
VFS_VNSTATE	Interface for vnode life cycle reporting.

A regular file system will define mount, unmount, root, statvfs, and vget methods. The vfs methods are defined in an fs_operation_def_t template, terminated by a NULL entry. The template is constructed from an array of fs_operation_def_t structures. The following example from the tmpfs implementation shows how the template is initialized and then instantiated with vfs_setfsops(). The call to vfs_setfsops() is typically done once per module initialization, systemwide.

```
static int
tmpfsinit(int fstype, char *name)
{
        static const fs_operation_def_t tmp_vfsops_template[] = {
                VFSNAME_MOUNT, tmp_mount,
                VFSNAME_UNMOUNT, tmp_unmount,
                VFSNAME_ROOT, tmp_root,
                VFSNAME_STATVFS, tmp_statvfs,
                VFSNAME_VGET, tmp_vget,
                NULL, NULL
        };
        int error;

        error = vfs_setfsops(fstype, tmp_vfsops_template, NULL);
...
}
                                 See usr/src/uts/common/fs/tmpfs/tmp_vfsops.c
```

A corresponding free of the `vfs` methods is required at module unload time and is typically located in the `_fini()` function of the module.

```
int
_fini()
{
        int error;

        error = mod_remove(&modlinkage);
        if (error)
                return (error);
        /*
         * Tear down the operations vectors
         */
        (void) vfs_freevfsops_by_type(tmpfsfstype);
        vn_freevnodeops(tmp_vnodeops);
        return (0);
}
                                 See usr/src/uts/common/fs/tmpfs/tmp_vfsops.c
```

The following routines are available in the `vfs` layer to manipulate the `vfs` object. They provide support for creating and modifying the FS methods (fsops),

```
/*
 * File systems use arrays of fs_operation_def structures to form
 * name/value pairs of operations.  These arrays get passed to:
 *
 *      - vn_make_ops() to create vnodeops
 *      - vfs_makefsops()/vfs_setfsops() to create vfsops.
 */
typedef struct fs_operation_def {
        char *name;                     /* name of operation (NULL at end) */
        fs_generic_func_p func;         /* function implementing operation */
} fs_operation_def_t;

int vfs_makefsops(const fs_operation_def_t *template, vfsops_t **actual);

Creates and builds (dummy) vfsops structures
```

continues

```
void vfs_setops(vfs_t *vfsp, vfsops_t *vfsops);

Sets the operations vector for this vfs

vfsops_t * vfs_getops(vfs_t *vfsp);

Retrieves the operations vector for this vfs

void vfs_freevfsops(vfsops_t *vfsops);

Frees a vfsops structure created by vfs_makefsops()

int vfs_freevfsops_by_type(int fstype);

For a vfsops structure created by vfs_setfsops(), use vfs_freevfsops_by_type()

int vfs_matchops(vfs_t *vfsp, vfsops_t *vfsops);

Determines if the supplied operations vector matches the vfs's operations vector. Note
that this is a "shallow" match. The pointer to the operations vector is compared, not
each individual operation.
                                          See usr/src/uts/common/sys/vfs.h
```

14.5.2 vfs Support Functions

The following support functions are available for parsing option strings and filling in the necessary vfs structure fields. The file systems also need to parse the option strings to learn what options should be used in completing the mount request. The routines and data structures are all defined in the vfs.h header file.

It is expected that all the fields used by the file-system-specific mount code in the vfs structure are normally filled in and interrogated only during a mount system call. At mount time the vfs structure is private and not available to any other parts of the kernel. So during this time, locking of the fields used in mnttab/options is not necessary. If a file system wants to update or interrogate options at some later time, then it should be locked by the vfs_lock_wait()/vfs_unlock() functions. All memory allocated by the following routines is freed at umount time, so callers need not worry about memory leakage. Any arguments whose values are preserved in a structure after a call have been copied, so callers need not worry about retained references to any function arguments.

```
struct mntopts_t *vfs_opttblptr(struct vfs *vfsp);

Returns a pointer to the mount options table for the given vfs structure.

void vfs_initopttbl(const mntopts_t *proto, mntopts_t *tbl);

Initializes a mount options table from the prototype mount options table pointed to by
the first argument.  A file system should always initialize the mount options table in
the vfs structure for the current mount but may use this routine to initialize other
tables if desired.  See the documentation below on how to construct a prototype mount
options table. Note that the vfs_opttblptr() function described above should be used to
access the vfs structures mount options table.
```

continues

void vfs_parsemntopts(mntopts_t *tbl, char *optionstr);

Parses the option string pointed to by the second argument, using the mount options table pointed to by the first argument. Any recognized options will be marked by this function as set in the pointed-to options table and any arguments found are recorded there as well. Normally file systems would call this with a pointer to the mount options table in the vfs structure for the mount currently being processed. The mount options table may be examined after the parse is completed, to see which options have been recognized, by using the vfs_optionisset() function documented below. Note that the parser will alter the option string during parsing, but will restore it before returning. Any options in the option string being parsed that are not recognized are silently ignored. Also if an option requires an arg but it is not supplied, the argument pointer is silently set to NULL. Since options are parsed from left to right, the last specification for any particular option in the option string is the one used. Similarly, if options that toggle each other on or off (i.e. are mutually exclusive), are in the same options string, the last one seen in left to right parsing determines the state of the affected option(s).

void vfs_clearmntopt(mntopts_t *tbl, const char *opt);

Clears the option whose name is passed in the second argument from the option table pointed to by the first argument, i.e., marks the option as not set and frees any argument that may be associated with the option. Used by file systems to unset options if so desired in a mount options table. Note that the only way to return options to their default state is to reinitialize the options table with vfs_initopttbl().

void vfs_setmntopt(mntopts_t *tbl, const char *opt, const char *arg, int flags);

Marks the option whose name is given by the second argument as set in the mount options table pointed to by the first argument. If the option takes an argument, the third parameter points to the string for the argument. The flags arg is provided to affect the behavior of the vfs_setmntopt function. It can cause it to override the MO_IGNORE flag if the particular option being set has this flag enabled. It can also be used to request toggling the MO_NODISPLAY bit for the option on or off. (see the documentation for mount option tables). Used by file systems to manually mark options as set in a mount options table. Possible flags to vfs_setmntopt:

```
VFS_DISPLAY  0x02 /* Turn off MO_NODISPLAY bit for option */
VFS_NODISPLAY 0x04 /* Turn on MO_NODISPLAY bit for option */
```

int vfs_optionisset(mntopts_t *tbl, const char *opt, char **argp);

Inquires if the option named by the second argument is marked as set in the mount options table pointed to by the first argument. Returns non-zero if the option was set. If the option has an argument string, the arg pointed to by the argp pointer is filled in with a pointer to the argument string for the option. The pointer is to the saved argument string and not to a copy. Users should not directly alter the pointed to string. If any change is desired to the argument string the caller should use the set/clearmntopt() functions.

int vfs_buildoptionstr(mntopts_t *tbl, char *buf, int len);

Builds a comma-separated, null-terminated string of the mount options that are set in the table passed in the first argument. The buffer passed in the second argument is filled in with the generated options string. If the length passed in the third argument would be exceeded, the function returns EOVERFLOW; otherwise, it returns zero on success. If an error is returned, the contents of the result buffer are undefined.

int vfs_setoptprivate(mntopts_t *tbl, const char *opt, void *arg);

Sets the private data field of the given option in the specified option table to the provided value. Returns zero on success, non-zero if the named option does not exist in the table. Note that option private data is not managed for the user. If the private data field is a pointer to allocated memory, then it should be freed by the file system code prior to returning from a umount call.

continues

```
int vfs_getoptprivate(mntopts_t *tbl, const char *opt, void **argp);

Fills in the pointer pointed to by the argp pointer with the value of the private data
field of the given option in the specified table. Returns zero on success, non-zero if
the named option does not exist in the table.

void vfs_setmntpoint(struct vfs *vfsp, char *mp);

Sets the vfs_mntpt field of the vfs structure to the given mount point. File systems
call this if they want some value there other than what was passed by the mount system
call.

int vfs_can_sync(vfs_t *vfsp);

Determines if a vfs has an FS-supplied (non default, non error) sync routine.

void vfs_setresource(struct vfs *vfsp, char *resource);

Sets the vfs_resource field of the vfs structure to the given resource. File systems
call this if they want some value there other than what was passed by the mount system
call.
                                                 See usr/src/uts/common/sys/vfs.h
```

14.5.3 The `mount` Method

The `mount` method is responsible for initializing a per-mount instance of a file system. It is typically invoked as a result of a user-initiated `mount` command.

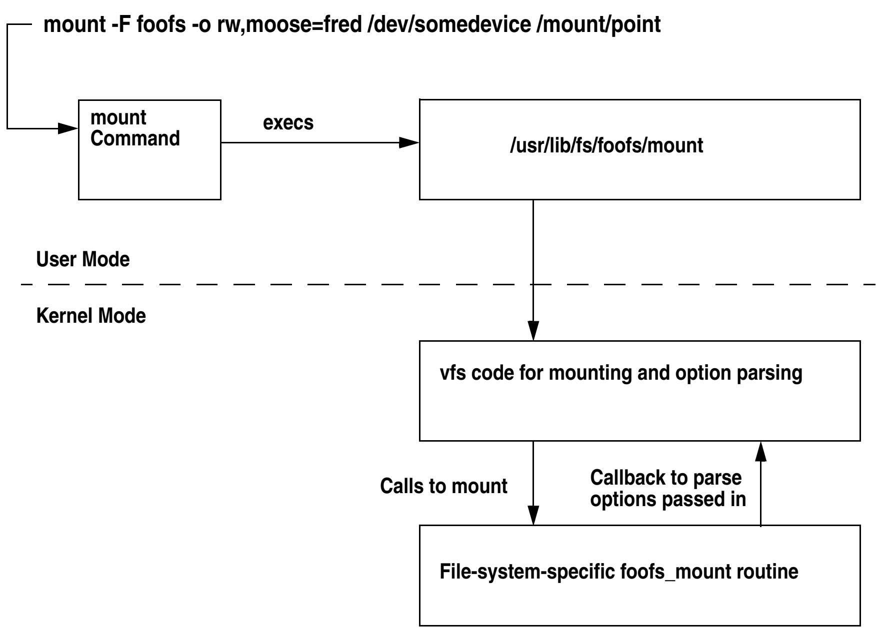

Figure 14.5 Mount Invocation

The tasks completed in the `mount` method will often include

- A security check, to ensure that the user has sufficient privileges to perform the requested mount. This is best done with a call to `secpolicy_fs_mount()`, with the Solaris Least Privilege framework.
- A check to see if the specified mount point is a directory.
- Initialization and allocation of per-file system mount structures and locks.
- Parsing of the options supplied into the mount call, with the assistance of the `vfs_option_*` support functions.
- Manufacture of a unique file system ID, with the help of `vfs_make_fsid()`. This is required to support NFS mount instances over the wire protocol using unique file system IDs.
- Creation or reading of the root inode for the file system.

An excerpt from the `tmpfs` implementation shows an example of the main functions within a file system mount method.

```
static int
tmp_mount(
        struct vfs *vfsp,
        struct vnode *mvp,
        struct mounta *uap,
        struct cred *cr)
{
        struct tmount *tm = NULL;
...
        if ((error = secpolicy_fs_mount(cr, mvp, vfsp)) != 0)
                return (error);

        if (mvp->v_type != VDIR)
                return (ENOTDIR);

        /* tmpfs doesn't support read-only mounts */
        if (vfs_optionisset(vfsp, MNTOPT_RO, NULL)) {
                error = EINVAL;
                goto out;
        }
...
        if (error = pn_get(uap->dir,
            (uap->flags & MS_SYSSPACE) ? UIO_SYSSPACE : UIO_USERSPACE, &dpn))
                goto out;

        if ((tm = tmp_memalloc(sizeof (struct tmount), 0)) == NULL) {
                pn_free(&dpn);
                error = ENOMEM;
                goto out;
        }
```

continues

```
...
        vfsp->vfs_data = (caddr_t)tm;
        vfsp->vfs_fstype = tmpfsfstype;
        vfsp->vfs_dev = tm->tm_dev;
        vfsp->vfs_bsize = PAGESIZE;
        vfsp->vfs_flag |= VFS_NOTRUNC;
        vfs_make_fsid(&vfsp->vfs_fsid, tm->tm_dev, tmpfsfstype);
...
        tm->tm_dev = makedevice(tmpfs_major, tmpfs_minor);
...
                                  See usr/src/uts/common/fs/tmpfs/tmp_vfsops.c
```

14.5.4 The `umount` Method

The `umount` method is almost the reverse of `mount`. The tasks completed in the `umount` method will often include

- A security check, to ensure that the user has sufficient privileges to perform the requested mount. This is best done with a call to `secpolicy_fs_mount()`, with the Solaris Least Privilege framework.
- A check to see if the mount is a forced mount (to take special action, or reject the request if the file system doesn't support forcible unmounts and the reference count on the root node is >1).
- Freeing of per-file system mount structures and locks.

14.5.5 Root `vnode` Identification

The root method of the file system is a simple function used by the file system lookup functions when traversing across a mount point into a new file system. It simply returns a pointer to the root `vnode` in the supplied `vnode` pointer argument.

```
static int
tmp_root(struct vfs *vfsp, struct vnode **vpp)
{
        struct tmount *tm = (struct tmount *)VFSTOTM(vfsp);
        struct tmpnode *tp = tm->tm_rootnode;
        struct vnode *vp;

        ASSERT(tp);

        vp = TNTOV(tp);
        VN_HOLD(vp);
        *vpp = vp;
        return (0);
}
                                  See usr/src/uts/common/fs/tmpfs/tmp_vfsops.c
```

14.5.6 `vfs` Information Available with `MDB`

The mounted list of `vfs` objects is linked as shown in Figure 14.6.

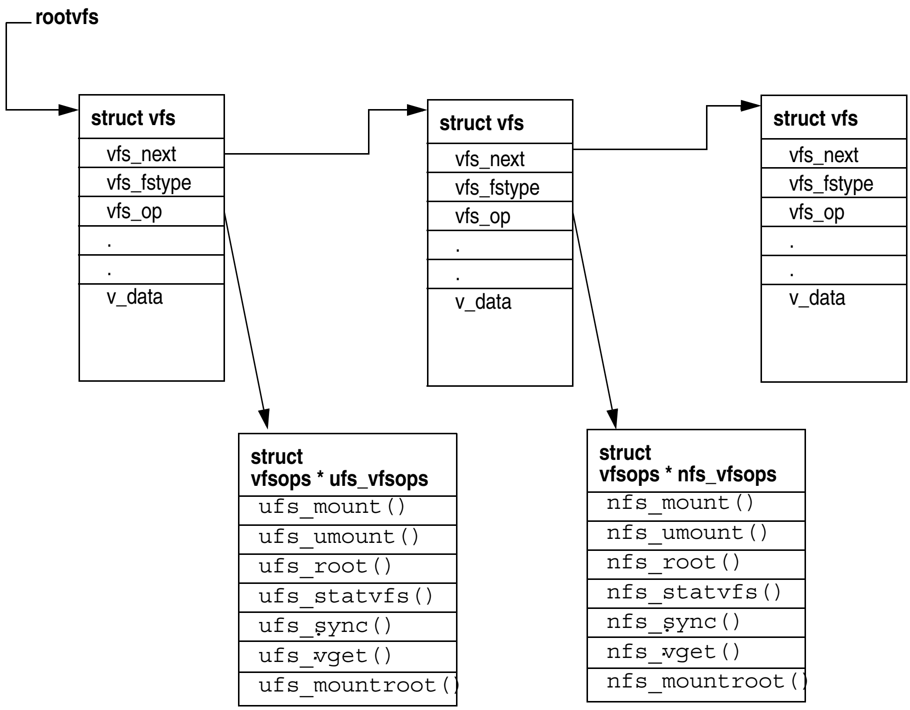

Figure 14.6 The Mounted `vfs` List

You can traverse the list with an `mdb` walker. Below is the output of such a traversal.

```
sol10# mdb -k
> ::walk vfs
fffffffffbc7a7a0
fffffffffbc7a860
> ::walk vfs |::fsinfo -v
            VFSP FS              MOUNT
fffffffffbc7a7a0 ufs             /
              R: /dev/dsk/c3d1s0
              O: remount,rw,intr,largefiles,logging,noquota,xattr,nodfratime
fffffffffbc7a860 devfs
/devices
              R: /devices
ffffffff80129300 ctfs            /system/contract
              R: ctfs
ffffffff80129240 proc            /proc
              R: proc
```

You can also inspect a vfs object with mdb. An example is shown below.

```
sol10# mdb -k
> ::walk vfs
fffffffffbc7a7a0
fffffffffbc7a860
> fffffffffbc7a7a0::print vfs_t
{
    vfs_next = devices
    vfs_prev = 0xffffffffba3ef0c0
    vfs_op = vfssw+0x138
    vfs_vnodecovered = 0
    vfs_flag = 0x420
    vfs_bsize = 0x2000
    vfs_fstype = 0x2
    vfs_fsid = {
        val = [ 0x19800c0, 0x2 ]
    }
    vfs_data = 0xffffffff8010ae00
    vfs_dev = 0x66000000c0
    vfs_bcount = 0
    vfs_list = 0
    vfs_hash = 0xffffffff816a8b40
    vfs_reflock = {
        _opaque = [ 0, 0 ]
    }
    vfs_count = 0x2
    vfs_mntopts = {
        mo_count = 0x20
        mo_list = 0xffffffff8133d580
    }
    vfs_resource = 0xffffffff8176dbb8
    vfs_mntpt = 0xffffffff81708590
    vfs_mtime = 2005 May 17 23:47:13
    vfs_femhead = 0
    vfs_zone = zone0
    vfs_zone_next = devices
    vfs_zone_prev = 0xffffffffba3ef0c0
}
```

14.6 The Vnode

A vnode is a file-system-independent representation of a file in the Solaris kernel. A vnode is said to be objectlike because it is an encapsulation of a file's state and the methods that can be used to perform operations on that file. A vnode represents a file within a file system; the vnode hides the implementation of the file system it resides in and exposes file-system-independent data and methods for that file to the rest of the kernel.

A vnode object contains three important items (see Figure 14.7).

- **File-system-independent data.** Information about the vnode, such as the type of vnode (file, directory, character device, etc.), flags that represent state, pointers to the file system that contains the vnode, and a reference count that keeps track of how many subsystems have references to the vnode.

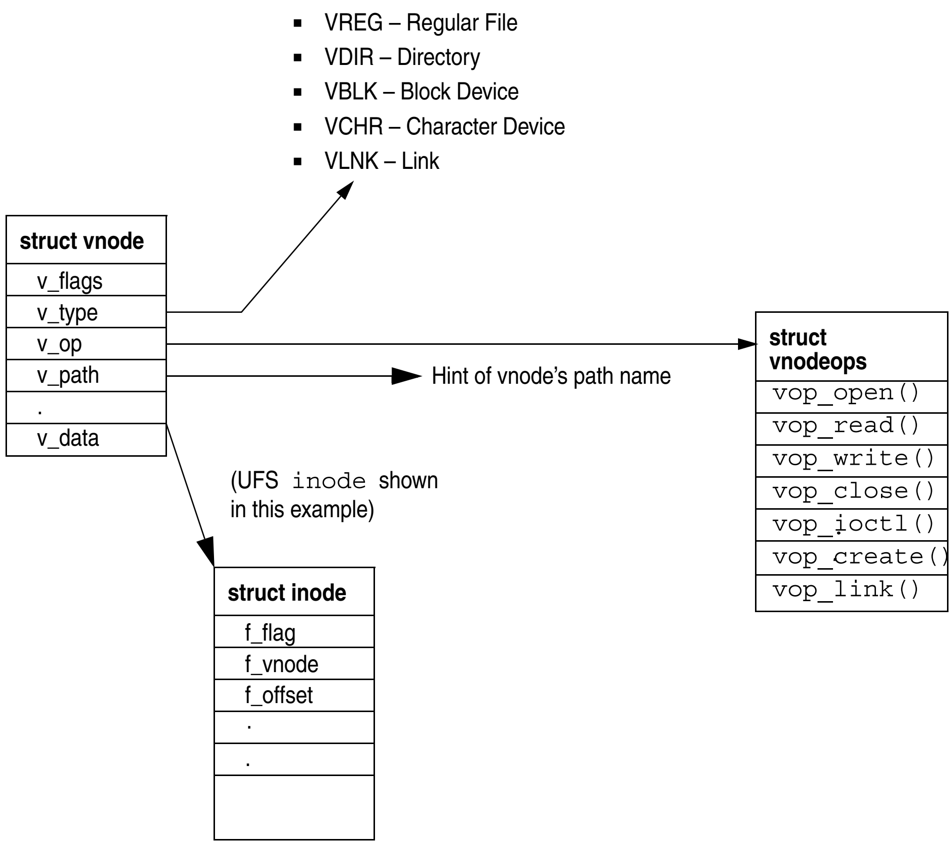

Figure 14.7 The `vnode` Object

- **Functions to implement file methods.** A structure of pointers to file-system-dependent functions to implement file functions such as `open()`, `close()`, `read()`, and `write()`.
- **File-system-specific data.** Data that is used internally by each file system implementation: typically, the in-memory `inode` that represents the `vnode` on the underlying file system. UFS uses an `inode`, NFS uses an `rnode`, and `tmpfs` uses a `tmpnode`.

14.6.1 Object Interface

The kernel uses wrapper functions to call `vnode` functions. In that way, it can perform `vnode` operations (for example, `read()`, `write()`, `open()`, `close()`) without knowing what the underlying file system containing the `vnode` is. For example, to read from a file without knowing that it resides on a UFS file system, the kernel would simply call the file-system-independent function for `read()`, `VOP_READ()`, which would call the `vop_read()` method of the `vnode`, which in turn calls the UFS function, `ufs_read()`. A sample of a `vnode` wrapper function from `sys/vnode.h` is shown below.

```
#define VOP_READ(vp, uiop, iof, cr, ct) \
        fop_read(vp, uiop, iof, cr, ct)
int
fop_read(
        vnode_t *vp,
        uio_t *uiop,
        int ioflag,
        cred_t *cr,
        struct caller_context *ct)
{
        return (*(vp)->v_op->vop_read)(vp, uiop, ioflag, cr, ct);
}
```
See usr/src/uts/common/sys/vnode.h

The `vnode` structure in Solaris OS can be found in `sys/vnode.h` and is shown below. It defines the basic interface elements and provides other information contained in the `vnode`.

```
typedef struct vnode {
        kmutex_t        v_lock;         /* protects vnode fields */
        uint_t          v_flag;         /* vnode flags (see below) */
        uint_t          v_count;        /* reference count */
        void            *v_data;        /* private data for fs */
        struct vfs      *v_vfsp;        /* ptr to containing VFS */
        struct stdata   *v_stream;      /* associated stream */
        enum vtype      v_type;         /* vnode type */
        dev_t           v_rdev;         /* device (VCHR, VBLK) */

        /* PRIVATE FIELDS BELOW - DO NOT USE */

        struct vfs      *v_vfsmountedhere; /* ptr to vfs mounted here */
        struct vnodeops *v_op;          /* vnode operations */
        struct page     *v_pages;       /* vnode pages list */
        pgcnt_t         v_npages;       /* # pages on this vnode */
        pgcnt_t         v_msnpages;     /* # pages charged to v_mset */
        struct page     *v_scanfront;   /* scanner front hand */
        struct page     *v_scanback;    /* scanner back hand */
        struct filock   *v_filocks;     /* ptr to filock list */
        struct shrlocklist *v_shrlocks; /* ptr to shrlock list */
        krwlock_t       v_nbllock;      /* sync for NBMAND locks */
        kcondvar_t      v_cv;           /* synchronize locking */
        void            *v_locality;    /* hook for locality info */
        struct fem_head *v_femhead;     /* fs monitoring */
        char            *v_path;        /* cached path */
        uint_t          v_rdcnt;        /* open for read count (VREG only) */
        uint_t          v_wrcnt;        /* open for write count (VREG only) */
        u_longlong_t    v_mmap_read;    /* mmap read count */
        u_longlong_t    v_mmap_write;   /* mmap write count */
```
continues

```
        void            *v_mpssdata;    /* info for large page mappings */
        hrtime_t        v_scantime;     /* last time this vnode was scanned */
        ushort_t        v_mset;         /* memory set ID */
        uint_t          v_msflags;      /* memory set flags */
        struct vnode    *v_msnext;      /* list of vnodes on an mset */
        struct vnode    *v_msprev;      /* list of vnodes on an mset */
        krwlock_t       v_mslock;       /* protects v_mset */
} vnode_t;
                                                See usr/src/uts/common/sys/vnode.h
```

14.6.2 vnode Types

Solaris OS has specific `vnode` types for files. The `v_type` field in the `vnode` structure indicates the type of `vnode`, as described in Table 14.2.

Table 14.2 Solaris 10 `vnode` Types from `sys/vnode.h`

Type	Description
VNON	No type
VREG	Regular file
VDIR	Directory
VBLK	Block device
VCHR	Character device
VLNK	Symbolic link
VFIFO	Named pipe
VDOOR	Doors interface
VPROC	`procfs` node
VSOCK	`sockfs` node (socket)
VPORT	Event port
VBAD	Bad `vnode`

14.6.3 vnode Method Registration

The `vnode` interface provides the set of file system object methods, some of which we saw in Figure 14.1. The file systems implement these methods to perform all file-system-specific file operations. Table 14.3 shows the `vnode` interface methods in Solaris OS.

File systems register their vnode and vfs operations by providing an operation definition table that specifies operations using name/value pairs. The definition is typically provided by a predefined template of type fs_operation_def_t, which is parsed by vn_make_ops(), as shown below. The definition is often set up in the file system initialization function.

```
/*
 * File systems use arrays of fs_operation_def structures to form
 * name/value pairs of operations.  These arrays get passed to:
 *
 *      - vn_make_ops() to create vnodeops
 *      - vfs_makefsops()/vfs_setfsops() to create vfsops.
 */
typedef struct fs_operation_def {
        char *name;                     /* name of operation (NULL at end) */
        fs_generic_func_p func;         /* function implementing operation */
} fs_operation_def_t;


int
vn_make_ops(
        const char *name,                       /* Name of file system */
        const fs_operation_def_t *templ,        /* Operation specification */
        vnodeops_t **actual);                   /* Return the vnodeops */

Creates and builds the private vnodeops table

void vn_freevnodeops(vnodeops_t *vnops);

Frees a vnodeops structure created by vn_make_ops()

void vn_setops(vnode_t *vp, vnodeops_t *vnodeops);

Sets the operations vector for this vnode

vnodeops_t * vn_getops(vnode_t *vp);

Retrieves the operations vector for this vnode

int vn_matchops(vnode_t *vp, vnodeops_t *vnodeops);

Determines if the supplied operations vector matches the vnode's operations vector.
Note that this is a "shallow" match. The pointer to the operations vector is compared,
not each individual operation. Returns non-zero (1) if the vnodeops matches that of the
vnode. Returns zero (0) if not.

int vn_matchopval(vnode_t *vp, char *vopname, fs_generic_func_p funcp)

Determines if the supplied function exists for a particular operation in the vnode's
operations vector
                                                See usr/src/uts/common/sys/vfs.h
```

The following example shows how the tmpfs file system sets up its vnode operations.

```
struct vnodeops *tmp_vnodeops;

const fs_operation_def_t tmp_vnodeops_template[] = {
        VOPNAME_OPEN, tmp_open,
        VOPNAME_CLOSE, tmp_close,
        VOPNAME_READ, tmp_read,
        VOPNAME_WRITE, tmp_write,
        VOPNAME_IOCTL, tmp_ioctl,
        VOPNAME_GETATTR, tmp_getattr,
        VOPNAME_SETATTR, tmp_setattr,
        VOPNAME_ACCESS, tmp_access,
                                        See usr/src/uts/common/fs/tmpfs/tmp_vnops.c


static int
tmpfsinit(int fstype, char *name)
{
...
        error = vn_make_ops(name, tmp_vnodeops_template, &tmp_vnodeops);
        if (error != 0) {
                (void) vfs_freevfsops_by_type(fstype);
                cmn_err(CE_WARN, "tmpfsinit: bad vnode ops template");
                return (error);
        }

...}
                                        See usr/src/uts/common/fs/tmpfs/tmp_vfsops.c
```

14.6.4 vnode Methods

The following section describes the method names that can be passed into vn_make_ops(), followed by the function prototypes for each method.

Table 14.3 Solaris 10 vnode Interface Methods from sys/vnode.h

Method	Description
VOP_ACCESS	Checks permissions
VOP_ADDMAP	Increments the map count
VOP_CLOSE	Closes the file
VOP_CMP	Compares two vnodes
VOP_CREATE	Creates the supplied path name
VOP_DELMAP	Decrements the map count
VOP_DISPOSE	Frees the given page from the vnode.
VOP_DUMP	Dumps data when the kernel is in a frozen state
VOP_DUMPCTL	Prepares the file system before and after a dump

continues

Table 14.3 Solaris 10 `vnode` Interface Methods from `sys/vnode.h` (*continued*)

Method	Description
VOP_FID	Gets unique file ID
VOP_FRLOCK	Locks files and records
VOP_FSYNC	Flushes out any dirty pages for the supplied vnode
VOP_GETATTR	Gets the attributes for the supplied vnode
VOP_GETPAGE	Gets pages for a vnode
VOP_GETSECATTR	Gets security access control list attributes
VOP_INACTIVE	Frees resources and releases the supplied vnode
VOP_IOCTL	Performs an I/O control on the supplied vnode
VOP_LINK	Creates a hard link to the supplied vnode
VOP_LOOKUP	Looks up the path name for the supplied vnode
VOP_MAP	Maps a range of pages into an address space
VOP_MKDIR	Makes a directory of the given name
VOP_VNEVENT	Support for File System Event Monitoring
VOP_OPEN	Opens a file referenced by the supplied vnode
VOP_PAGEIO	Supports page I/O for file system swap files
VOP_PATHCONF	Establishes file system parameters
VOP_POLL	Supports the poll() system call for file systems
VOP_PUTPAGE	Writes pages in a vnode
VOP_READ	Reads the range supplied for the given vnode
VOP_READDIR	Reads the contents of a directory
VOP_READLINK	Follows the symlink in the supplied vnode
VOP_REALVP	Gets the real vnode from the supplied vnode
VOP_REMOVE	Removes the file for the supplied vnode
VOP_RENAME	Renames the file to the new name
VOP_RMDIR	Removes a directory pointed to by the supplied vnode
VOP_RWLOCK	Holds the reader/writer lock for the supplied vnode
VOP_RWUNLOCK	Releases the reader/writer lock for the supplied vnode
VOP_SEEK	Checks seek bounds within the supplied vnode
VOP_SETATTR	Sets the attributes for the supplied vnode
VOP_SETFL	Sets file-system-dependent flags on the supplied vnode
VOP_SETSECATTR	Sets security access control list attributes

continues

Table 14.3 Solaris 10 `vnode` Interface Methods from `sys/vnode.h` (*continued*)

Method	Description
VOP_SHRLOCK	Supports NFS shared locks
VOP_SPACE	Frees space for the supplied vnode
VOP_SYMLINK	Creates a symbolic link between the two path names
VOP_WRITE	Writes the range supplied for the given vnode

```
extern int fop_access(vnode_t *vp, int mode, int flags, cred_t *cr);

Checks to see if the user (represented by the cred structure) has permission to do an
operation. Mode is made up of some combination (bitwise OR) of VREAD, VWRITE, and VEXEC.
These bits are shifted to describe owner, group, and "other" access.

extern int fop_addmap(vnode_t *vp, offset_t off, struct as *as, caddr_t addr,
                      size_t len, uchar_t prot, uchar_t maxprot, uint_t flags,
                      cred_t *cr);

Increments the map count.

extern int fop_close(vnode_t *vp, int flag, int count, offset_t off, cred_t *cr);

Closes the file given by the supplied vnode. When this is the last close, some file sys-
tems use vop_close() to initiate a writeback of outstanding dirty pages by checking the
reference count in the vnode.

extern int fop_cmp(vnode_t *vp1, vnode_t *vp2);

Compares two vnodes. In almost all cases, this defaults to fs_cmp() which simply does a:
return (vp1 == vp2);

NOTE: NFS/NFS3 and Cachefs have their own CMP routines, but they do
exactly what fs_cmp() does. Procfs appears to be the only exception. It looks like it
follows a chain.

extern int fop_create(vnode_t *dvp, char *name, vattr_t *vap, vcexcl_t excl, int mode,
                      vnode_t **vp, cred_t *cr, int flag);

Creates a file with the supplied path name.

extern int fop_delmap(vnode_t *vp, offset_t off, struct as *as, caddr_t addr,
                    size_t len, uint_t prot, uint_t maxprot, uint_t flags, cred_t *cr);

Decrements the map count.

extern void fop_dispose(vnode_t *vp, struct page *pp, int flag, int dn, cred_t *cr);

Frees the given page from the vnode.

extern int fop_dump(vnode_t *vp, caddr_t addr, int lbdn, int dblks);

Dumps data when the kernel is in a frozen state.

extern int fop_dumpctl(vnode_t *vp, int action, int *blkp);

Prepares the file system before and after a dump.
```

continues

```
extern int fop_fid(vnode_t *vp, struct fid *fidp);
```

Puts a unique (by node, file system, and host) vnode/xxx_node identifier into fidp. Used for NFS file-handles.

```
extern int fop_frlock(vnode_t *vp, int cmd, struct flock64 *bfp, int flag,
                      offset_t off, struct flk_callback *flk_cbp, cred_t *cr);
```

Does file and record locking for the supplied vnode. Most file systems either map this to fs_frlock() or do some special case checking and call fs_frlock() directly. As you might expect, fs_frlock() does all the dirty work.

```
extern int fop_fsync(vnode_t *vp, int syncflag, cred_t *cr);
```

Flushes out any dirty pages for the supplied vnode.

```
extern int fop_getattr(vnode_t *vp, vattr_t *vap, int flags, cred_t *cr);
```

Gets the attributes for the supplied vnode.

```
extern int fop_getpage(vnode_t *vp, offset_t off, size_t len, uint_t protp,
                       struct page **plarr, size_t plsz, struct seg *seg,
                       caddr_t addr, enum seg_rw rw, cred_t *cr);
```

Gets pages in the range offset and length for the vnode from the backing store of the file system. Does the real work of reading a vnode. This method is often called as a result of read(), which causes a page fault in seg_map, which calls vop_getpage.

```
extern int fop_getsecattr(vnode_t *vp, vsecattr_t *vsap, int flag, cred_t *cr);
```

Gets security access control list attributes.

```
extern void fop_inactive(vnode_t *vp, cred_t *cr);
```

Frees resources and releases the supplied vnode. The file system can choose to destroy the vnode or put it onto an inactive list, which is managed by the file system implementation.

```
extern int fop_ioctl(vnode_t *vp, int cmd, intptr_t arg, int flag, cred_t *cr,
                     int *rvalp);
```

Performs an I/O control on the supplied vnode.

```
extern int fop_link(vnode_t *targetvp, vnode_t *sourcevp, char *targetname, cred_t
*cr);
```

Creates a hard link to the supplied vnode.

```
extern int fop_lookup(vnode_t *dvp, char *name, vnode_t **vpp, int flags, vnode_t
*rdir,
                      cred_t *cr);
```

Looks up the name in the directory vnode dvp with the given dirname and returns the new vnode in vpp. The vop_lookup() does file-name translation for the open, stat system calls.

```
extern int fop_map(vnode_t *vp, offset_t off, struct as *as, caddr_t *addrp, size_t len,
                   uchar_t prot, uchar_t maxprot, uint_t flags, cred_t *cr);
```

Maps a range of pages into an address space by doing the appropriate checks and calling as_map().

continues

extern int fop_mkdir(vnode_t *dvp, char *name, vattr_t *vap, vnode_t **vpp, cred_t *cr);

Makes a directory in the directory vnode (dvp) with the given name (dirname) and returns the new vnode in vpp.

extern int fop_vnevent(vnode_t *vp, vnevent_t vnevent);

Interface for reporting file events. File systems need not implement this method.

extern int fop_open(vnode_t **vpp, int mode, cred_t *cr);

Opens a file referenced by the supplied vnode. The open() system call has already done a vop_lookup() on the path name, which returned a vnode pointer and then calls to vop_open(). This function typically does very little, since most of the real work was performed by vop_lookup(). Also called by file systems to open devices as well as by anything else that needs to open a file or device.

extern int fop_pageio(vnode_t *vp, struct page *pp, u_offset_t io_off, size_t io_len, int flag, cred_t *cr);

Paged I/O support for file system swap files.

extern int fop_pathconf(vnode_t *vp, int cmd, ulong_t *valp, cred_t *cr);

Establishes file system parameters with the pathconf system call.

extern int fop_poll(vnode_t *vp, short events, int anyyet, short *reventsp, struct pollhead **phpp);

File system support for the poll() system call.

extern int fop_putpage(vnode_t *vp, offset_t off, size_t len, int, cred_t *cr);

Writes pages in the range offset and length for the vnode to the backing store of the file system. Does the real work of writing a vnode.

extern int fop_read(vnode_t *vp, uio_t *uiop, int ioflag, cred_t *cr, caller_context_t *ct);

Reads the range supplied for the given vnode. vop_read() typically maps the requested range of a file into kernel memory and then uses vop_getpage() to do the real work.

extern int fop_readdir(vnode_t *vp, uio_t *uiop, cred_t *cr, int *eofp);

Reads the contents of a directory.

extern int fop_readlink(vnode_t *vp, uio_t *uiop, cred_t *cr);

Follows the symlink in the supplied vnode.

extern int fop_realvp(vnode_t *vp, vnode_t **vpp);

Gets the real vnode from the supplied vnode.

extern int fop_remove(vnode_t *dvp, char *name, cred_t *cr);

Removes the file for the supplied vnode.

extern int fop_rename(vnode_t *sourcedvp, char *sourcename, vnode_t *targetdvp, char *targetname, cred_t *cr);

Renames the file named (by sourcename) in the directory given by sourcedvp to the new name (targetname) in the directory given by targetdvp.

continues

extern int fop_rmdir(vnode_t *dvp, char *name, vnode_t *vp, cred_t *cr);

Removes the name in the directory given by dvp.

extern int fop_rwlock(vnode_t *vp, int write_lock, caller_context_t *ct);

Holds the reader/writer lock for the supplied vnode. This method is called for each vnode, with the rwflag set to 0 inside a read() system call and the rwflag set to 1 inside a write() system call. POSIX semantics require only one writer inside write() at a time. Some file system implementations have options to ignore the writer lock inside vop_rwlock().

extern void fop_rwunlock(vnode_t *vp, int write_lock, caller_context_t *ct);

Releases the reader/writer lock for the supplied vnode.

extern int fop_seek(vnode_t *vp, offset_t oldoff, offset_t *newoffp);

Checks the FS-dependent bounds of a potential seek.
NOTE: VOP_SEEK() doesn't do the seeking. Offsets are usually saved in the file_t structure and are passed down to VOP_READ/VOP_WRITE in the uiostructure.

**extern int fop_setattr(vnode_t *vp, vattr_t *vap, int flags, cred_t *cr,
caller_context_t *cr);**

Sets the file attributes for the supplied vnode.

extern int fop_setfl(vnode_t *vp, int oldflags, int newflags, cred_t *cr);

Sets the file system-dependent flags (typically for a socket) for the supplied vnode.

extern int fop_setsecattr(vnode_t *vp, vsecattr_t *vsap, int flag, cred_t *cr);

Sets security access control list attributes.

extern int fop_shrlock(vnode_t *vp, int cmd, struct shrlock *shr, int flag, cred_t *cr);

ONC shared lock support.

**extern int fop_space(vnode_t vp*, int cmd, struct flock64 *bfp, int flag,
offset_t off, cred_t *cr, caller_context_t *ct);**

Frees space for the supplied vnode.

**extern int fop_symlink(vnode_t *vp, char *linkname, vattr_t *vap, char *target,
cred_t *cred);**

Creates a symbolic link between the two path names.

**extern int fop_write(vnode_t *vp, uio_t *uiop, int ioflag, cred_t *cr,
caller_context_t *ct);**

Writes the range supplied for the given vnode. The write system call typically maps the requested range of a file into kernel memory and then uses vop_putpage() to do the real work.

See usr/src/uts/common/sys/vnode.h

14.6.5 Support Functions for Vnodes

Following is a list of the public functions available for obtaining information from within the private part of the `vnode`.

```
int vn_is_readonly(vnode_t *);

Is the vnode write protected?

int vn_is_opened(vnode_t *, v_mode_t);

Is the file open?

int vn_is_mapped(vnode_t *, v_mode_t);

Is the file mapped?

int vn_can_change_zones(vnode_t *vp);

Check if the vnode can change zones: used to check if a process can change zones. Mainly
used for NFS.

int vn_has_flocks(vnode_t *);

Do file/record locks exist for this vnode?

int vn_has_mandatory_locks(vnode_t *, int);

Does the vnode have mandatory locks in force for this mode?

int vn_has_cached_data(vnode_t *);

Does the vnode have cached data associated with it?

struct vfs *vn_mountedvfs(vnode_t *);

Returns the vfs mounted on this vnode if any

int vn_ismntpt(vnode_t *);

Returns true (non-zero) if this vnode is mounted on, zero otherwise

                                                  See usr/src/uts/common/sys/vnode.h
```

14.6.6 The Life Cycle of a Vnode

A vnode is an in-memory reference to a file. It is a transient structure that lives in memory when the kernel references a file within a file system.

A vnode is allocated by `vn_alloc()` when a first reference to an existing file is made or when a file is created. The two common places in a file system implementation are within the `VOP_LOOKUP()` method or within the `VOP_CREAT()` method.

When a file descriptor is opened to a file, the reference count for that `vnode` is incremented. The `vnode` is always in memory when the reference count is greater

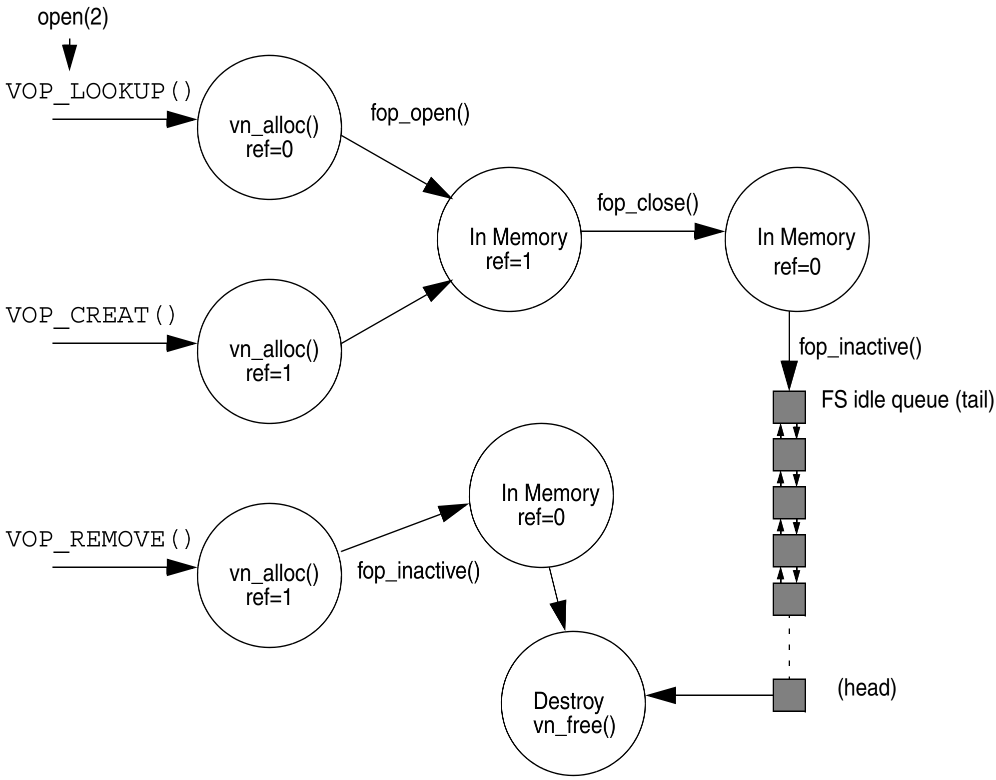

Figure 14.8 The Life Cycle of a `vnode` Object

than zero. The reference count may drop back to zero after the last file descriptor has been closed, at which point the file system framework calls the file system's `VOP_INACTIVE()` method.

Once a `vnode`'s reference count becomes zero, it is a candidate for freeing. Most file systems won't free the vnode immediately, since to recreate it will likely require a disk I/O for a directory read or an over-the-wire operation. For example, the UFS keeps a list of inactive inodes on an "inactive list" (see Section 15.3.1). Only when certain conditions are met (for example, a resource shortage) is the `vnode` actually freed.

Of course, when a file is deleted, its corresponding in-memory `vnode` is freed. This is also performed by the `VOP_INACTIVE()` method for the file system: Typically, the `VOP_INACTIVE()` method checks to see if the link count for the `vnode` is zero and then frees it.

14.6.7 vnode Creation and Destruction

The allocation of a vnode must be done by a call to the appropriate support function. The functions for allocating, destroying, and reinitializing vnodes are shown below.

```
vnode_t *vn_alloc(int kmflag);

Allocate a vnode and initialize all of its structures.

void vn_free(vnode_t *vp);

Free the allocated vnode.

void vn_reinit(vnode_t *vp);

(Re)initializes a vnode.
                                        See usr/src/uts/common/sys/vnode.h
```

14.6.8 The vnode Reference Count

A vnode is created by the file system at the time a file is first opened or created and stays active until the file system decides the vnode is no longer needed. The vnode framework provides an infrastructure that keeps track of the number of references to a vnode. The kernel maintains the reference count by means of the VN_HOLD() and VN_RELE() macros, which increment and decrement the v_count field of the vnode. The vnode stays valid while its reference count is greater than zero, so a subsystem can rely on a vnode's contents staying valid by calling VN_HOLD() before it references a vnode's contents. It is important to distinguish a vnode reference from a lock; a lock ensures exclusive access to the data, and the reference count ensures persistence of the object.

When a vnode's reference count drops to zero, VN_RELE() invokes the VOP_INACTIVE() method for that file system. Every subsystem that references a vnode is required to call VN_HOLD() at the start of the reference and to call VN_RELE() at the end of each reference. Some file systems deconstruct a vnode when its reference count falls to zero; others hold on to the vnode for a while so that if it is required again, it is available in its constructed state. UFS, for example, holds on to the vnode for a while after the last release so that the virtual memory system can keep the inode and cache for a file, whereas PCFS frees the vnode and all of the cache associated with the vnode at the time VOP_INACTIVE() is called.

14.6.9 Interfaces for Paging vnode Cache

Solaris OS unifies file and memory management by using a vnode to represent the backing store for virtual memory (see Chapter 8). A page of memory represents a particular vnode and offset. The file system uses the memory relationship to implement caching for vnodes within a file system. To cache a vnode, the file system has the memory system create a page of physical memory that represents the vnode and offset.

The virtual memory system provides a set of functions for cache management and I/O for vnodes. These functions allow the file systems to cluster pages for I/O and handle the setup and checking required for synchronizing dirty pages with their backing store. The functions, described below, set up pages so that they can be passed to device driver block I/O handlers.

```
int pvn_getdirty(struct page *pp, int flags);
```

Queries whether a page is dirty. Returns 1 if the page should be written back (the iolock is held in this case), or 0 if the page has been dealt with or has been unlocked.

```
void pvn_plist_init(struct page *pp, struct page **pl, size_t plsz,
                    u_offset_t off, size_t io_len, enum seg_rw rw);
```

Releases the iolock on each page and downgrades the page lock to shared after new pages have been created or read.

```
void pvn_read_done(struct page *plist, int flags);
```

Unlocks the pages after read is complete. The function is normally called automatically by pageio_done() but may need to be called if an error was encountered during a read.

```
struct page *pvn_read_kluster(struct vnode *vp, u_offset_t off,
                              struct seg *seg, caddr_t addr, u_offset_t *offp,
                              size_t *lenp, u_offset_t vp_off, size_t vp_len,
                              int isra);
```

Finds the range of contiguous pages within the supplied address / length that fit within the provided vnode offset / length that do not already exist. Returns a list of newly created, exclusively locked pages ready for I/O. Checks that clustering is enabled by calling the segop_kluster() method for the given segment. On return from pvn_read_kluster, the caller typically zeroes any parts of the last page that are not going to be read from disk, sets up the read with pageio_setup for the returned offset and length, and then initiates the read with bdev_strategy().Once the read is complete, pvn_plist_init() can release the I/O lock on each page that was created.

```
void pvn_write_done(struct page *plist, int flags);
```

Unlocks the pages after write is complete. For asynchronous writes, the function is normally called automatically by pageio_done() when an asynchronous write completes. For synchronous writes, pvn_write_done() is called after pageio_done to unlock written pages. It may also need to be called if an error was encountered during a write.

```
struct page *pvn_write_kluster(struct vnode *vp, struct page *pp,
                               u_offset_t *offp, size_t *lenp, u_offset_t vp_off,
                               size_t vp_len, int flags);
```

Finds the contiguous range of dirty pages within the supplied offset and length. Returns a list of dirty locked pages ready to be written back. On return from pvn_write_kluster(), the caller typically sets up the write with pageio_setup for the returned offset and length, then initiates the write with bdev_strategy(). If the write is synchronous, then the caller should call pvn_write_done() to unlock the pages. If the write is asynchronous, then the io_done routine calls pvn_write_done when the write is complete.

continues

```
int pvn_vplist_dirty(struct vnode *vp, u_offset_t off,
                     int (*putapage)(vnode_t *, struct page *, u_offset_t *,
                     size_t *, int, cred_t *),
                     int flags, struct cred *cred);

Finds all dirty pages in the page cache for a given vnode that have an offset greater
than the supplied offset and calls the supplied putapage() routine. pvn_vplist_dirty()
is often used to synchronize all dirty pages for a vnode when vop_putpage is called with
a zero length.

int pvn_getpages(int (*getpage)(vnode_t *, u_offset_t, size_t, uint_t *,
                                struct page *[], size_t, struct seg *,
                                caddr_t, enum seg_rw, cred_t *),
                 struct vnode *vp, u_offset_t off, size_t len,
                 uint_t *protp, struct page **pl, size_t plsz,
                 struct seg *seg, caddr_t addr, enum seg_rw rw,
                 struct cred *cred);

Handles common work of the VOP_GETPAGE routines when more than one page must be returned
by calling a file-system-specific operation to do most of the work. Must be called with
the vp already locked by the VOP_GETPAGE routine.

void pvn_io_done(struct page *plist);

Generic entry point used to release the "shared/exclusive" lock and the "p_iolock" on
pages after i/o is complete.

void pvn_vpzero(struct vnode *vp, u_offset_t vplen, size_t zbytes);

Zeros-out zbytes worth of data. Caller should be aware that this routine may enter back
into the fs layer (xxx_getpage). Locks that the xxx_getpage routine may need should not
be held while calling this.
                                                 See usr/src/uts/common/sys/pvn.h
```

14.6.10 Block I/O on vnode Pages

The block I/O subsystem supports I/O initiation to and from vnode pages. It schedules I/O from the device drivers directly to and from a page without buffering the data in the buffer cache. These functions are typically used in the implementation of vop_getpage() and vop_putpage() to do the physical I/O on behalf of the file system. Three functions, shown below, initiate I/O between a physical page and a device.

```
struct buf *pageio_setup(struct page *, size_t, struct vnode *, int);

Sets up a block buffer for I/O on a page of memory so that it bypasses the block buffer
cache by setting the B_PAGEIO flag and putting the page list on the b_pages field.

extern int bdev_strategy(struct buf *);

Initiates an I/O on a page, using the block I/O device.

void pageio_done(struct buf *);

Waits for the block device I/O to complete.
                                                 See usr/src/uts/common/sys/bio.h
```

14.6.11 `vnode` Information Obtainable with `mdb`

You can use mdb to traverse the vnode cache, inspect a vnode object, view the path name, and examine linkages between vnodes.

With the centralized vn_alloc(), a central vnode cache holds all the vnode structures. It is a regular kmem cache and can be traversed with mdb and the generic kmem cache walker.

```
sol10# mdb -k
> ::walk vn_cache
ffffffff80f24040
ffffffff80f24140
ffffffff80f24240
ffffffff8340d940
...
```

Similarly, you can inspect a vnode object.

```
sol10# mdb -k
> ::walk vn_cache
ffffffff80f24040
ffffffff80f24140
ffffffff80f24240
ffffffff8340d940
...
> ffffffff8340d940::print vnode_t
{
    v_lock = {
        _opaque = [ 0 ]
    }
    v_flag = 0x10000
    v_count = 0x2
    v_data = 0xffffffff8340e3d8
    v_vfsp = 0xffffffff816a8f00
    v_stream = 0
    v_type = 1 (VREG)
    v_rdev = 0xffffffffffffffff
    v_vfsmountedhere = 0
    v_op = 0xffffffff805fe300
    v_pages = 0
    v_npages = 0
    v_msnpages = 0
    v_scanfront = 0
    v_scanback = 0
    v_filocks = 0
    v_shrlocks = 0
    v_nbllock = {
        _opaque = [ 0 ]
    }
    v_cv = {
        _opaque = 0
    }
    v_locality = 0
    v_femhead = 0
    v_path = 0xffffffff8332d440 "/zones/gallery/root/var/svc/log/work-inetd:default.log"
```

continues

```
    v_rdcnt = 0
    v_wrcnt = 0x1
    v_mmap_read = 0
    v_mmap_write = 0
    v_mpssdata = 0
    v_scantime = 0
    v_mset = 0
    v_msflags = 0
    v_msnext = 0
    v_msprev = 0
    v_mslock = {
        _opaque = [ 0 ]
    }
}
```

With other mdb d-commands, you can view the vnode's path name (a guess, cached during vop_lookup), the linkage between vnodes, which processes have them open, and vice versa.

```
> ffffffff8340d940::vnode2path
/zones/gallery/root/var/svc/log//network-inetd:default.log
> ffffffff8340d940::whereopen
file ffffffff832d4bd8
ffffffff83138930
> ffffffff83138930::ps
S    PID   PPID   PGID    SID    UID      FLAGS             ADDR NAME
R    845      1    845    845      0 0x42000400 ffffffff83138930 inetd
> ffffffff83138930::pfiles
FD   TYPE            VNODE INFO
   0  CHR ffffffff857c8580 /zones/gallery/root/dev/null
   1  REG ffffffff8340d940 /zones/gallery/root/var/svc/log//network-inetd:default.log
   2  REG ffffffff8340d940 /zones/gallery/root/var/svc/log//network-inetd:default.log
   3 FIFO ffffffff83764940
   4 DOOR ffffffff836d1680 [door to 'nscd' (proc=ffffffff835ecd10)]
   5 DOOR ffffffff83776800 [door to 'svc.configd' (proc=ffffffff8313f928)]
   6 DOOR ffffffff83776900 [door to 'svc.configd' (proc=ffffffff8313f928)]
   7 FIFO ffffffff83764540
   8  CHR ffffffff83776500 /zones/gallery/root/dev/sysevent
   9  CHR ffffffff83776300 /zones/gallery/root/dev/sysevent
  10 DOOR ffffffff83776700 [door to 'inetd' (proc=ffffffff83138930)]
  11  REG ffffffff833fcac0 /zones/gallery/root/system/contract/process/template
  12 SOCK ffffffff83215040 socket: AF_UNIX /var/run/.inetd.uds
  13  CHR ffffffff837f1e40 /zones/gallery/root/dev/ticotsord
  14  CHR ffffffff837b6b00 /zones/gallery/root/dev/ticotsord
  15 SOCK ffffffff85d106c0 socket: AF_INET6 :: 48155
  16 SOCK ffffffff85cdb000 socket: AF_INET6 :: 20224
  17 SOCK ffffffff83543440 socket: AF_INET6 :: 5376
  18 SOCK ffffffff8339de80 socket: AF_INET6 :: 258
  19  CHR ffffffff85d27440 /zones/gallery/root/dev/ticlts
  20  CHR ffffffff83606100 /zones/gallery/root/dev/udp
  21  CHR ffffffff8349ba00 /zones/gallery/root/dev/ticlts
  22  CHR ffffffff8332f680 /zones/gallery/root/dev/udp
  23  CHR ffffffff83606600 /zones/gallery/root/dev/ticots
  24  CHR ffffffff834b2d40 /zones/gallery/root/dev/ticotsord
  25  CHR ffffffff8336db40 /zones/gallery/root/dev/tcp
  26  CHR ffffffff83626540 /zones/gallery/root/dev/ticlts
  27  CHR ffffffff834f1440 /zones/gallery/root/dev/udp
  28  CHR ffffffff832d5940 /zones/gallery/root/dev/ticotsord
  29  CHR ffffffff834e4b80 /zones/gallery/root/dev/ticotsord
```

continues

```
  30 SOCK ffffffff83789580 socket: AF_INET 0.0.0.0 514
  31 SOCK ffffffff835a6e80 socket: AF_INET6 :: 514
  32 SOCK ffffffff834e4d80 socket: AF_INET6 :: 5888
  33  CHR ffffffff85d10ec0 /zones/gallery/root/dev/ticotsord
  34  CHR ffffffff83839900 /zones/gallery/root/dev/tcp
  35 SOCK ffffffff838429c0 socket: AF_INET 0.0.0.0 11904
```

14.6.12 DTrace Probes in the vnode Layer

DTrace provides probes for file system activity through the `vminfo` provider and, optionally, through deeper tracing with the `fbt` provider. All the `cpu_vminfo` statistics are updated from `pageio_setup()` (see Section 14.6.10).

The `vminfo` provider probes correspond to the fields in the "vm" named `kstat`: a probe provided by `vminfo` fires immediately before the corresponding `vm` value is incremented. Table 14.4 lists the probes available from the VM provider; these are further described in Section 6.11 in *Solaris™ Performance and Tools*. A probe takes the following arguments.

arg0. The value by which the statistic is to be incremented. For most probes, this argument is always 1, but for some it may take other values; these probes are noted in Table 14.4.

arg1. A pointer to the current value of the statistic to be incremented. This value is a 64-bit quantity that is incremented by the value in `arg0`. Dereferencing this pointer allows consumers to determine the current count of the statistic corresponding to the probe.

For example, the following paging activity that is visible with `vmstat` indicates page-in from the file system (`fpi`).

```
sol8# vmstat -p 3
     memory           page          executable      anonymous      filesystem
   swap  free  re  mf  fr  de  sr  epi  epo  epf  api  apo  apf  fpi  fpo  fpf
 1512488 837792 160 20 12   0   0    0    0    0 8102    0    0   12   12   12
 1715812 985116 7  82   0   0   0    0    0    0 7501    0    0   45    0    0
 1715784 983984 0   2   0   0   0    0    0    0 1231    0    0   53    0    0
 1715780 987644 0   0   0   0   0    0    0    0 2451    0    0   33    0    0

sol10$ dtrace -n fspgin'{@[execname] = count()}'
dtrace: description 'fspgin' matched 1 probe
  svc.startd                                                        1
  sshd                                                              2
  ssh                                                               3
  dtrace                                                            6
  vmstat                                                            8
  filebench                                                        13
```

See Section 6.11 in *Solaris™ Performance and Tools* for examples of how to use `dtrace` for memory analysis.

Table 14.4 DTrace VM Provider Probes and Descriptions

Probe Name	Description
`anonfree`	Fires whenever an unmodified anonymous page is freed as part of paging activity. Anonymous pages are those that are not associated with a file; memory containing such pages include heap memory, stack memory, or memory obtained by explicitly mapping zero(7D).
`anonpgin`	Fires whenever an anonymous page is paged in from a swap device.
`anonpgout`	Fires whenever a modified anonymous page is paged out to a swap device.
`as_fault`	Fires whenever a fault is taken on a page and the fault is neither a protection fault nor a copy-on-write fault.
`cow_fault`	Fires whenever a copy-on-write fault is taken on a page. `arg0` contains the number of pages that are created as a result of the copy-on-write.
`dfree`	Fires whenever a page is freed as a result of paging activity. Whenever `dfree` fires, exactly one of `anonfree`, `execfree`, or `fsfree` will also subsequently fire.
`execfree`	Fires whenever an unmodified executable page is freed as a result of paging activity.
`execpgin`	Fires whenever an executable page is paged in from the backing store.
`execpgout`	Fires whenever a modified executable page is paged out to the backing store. If it occurs at all, most paging of executable pages will occur in terms of `execfree`; `execpgout` can only fire if an executable page is modified in memory—an uncommon occurrence in most systems.
`fsfree`	Fires whenever an unmodified file system data page is freed as part of paging activity.
`fspgin`	Fires whenever a file system page is paged in from the backing store.
`fspgout`	Fires whenever a modified file system page is paged out to the backing store.
`kernel_asflt`	Fires whenever a page fault is taken by the kernel on a page in its own address space. Whenever `kernel_asflt` fires, it will be immediately preceded by a firing of the `as_fault` probe.
`maj_fault`	Fires whenever a page fault is taken that results in I/O from a backing store or swap device. Whenever `maj_fault` fires, it will be immediately preceded by a firing of the `pgin` probe.
`pgfrec`	Fires whenever a page is reclaimed off the free page list.

Below is an example of tracing a generic vnode layer with DTrace.

```
dtrace:::BEGIN
{
        printf("%-15s %-10s %51s %2s %8s %8s\n",
                "Event", "Device", "Path", "RW", "Size", "Offset");
        self->trace = 0;
        self->path = "";
}


fbt::fop_*:entry
/self->trace == 0/
{
        /* Get vp: fop_open has a pointer to vp */
        self->vpp = (vnode_t **)arg0;
        self->vp = (vnode_t *)arg0;
        self->vp = probefunc == "fop_open" ? (vnode_t *)*self->vpp : self->vp;

        /* And the containing vfs */
        self->vfsp = self->vp ? self->vp->v_vfsp : 0;

        /* And the paths for the vp and containing vfs */
        self->vfsvp = self->vfsp ? (struct vnode *)((vfs_t *)self->vfsp)->vfs_vnodecov-
ered : 0;
        self->vfspath = self->vfsvp ? stringof(self->vfsvp->v_path) : "unknown";

        /* Check if we should trace the root fs */
        ($1 == "/all" ||
         ($1 == "/" && self->vfsp && \
         (self->vfsp == `rootvfs))) ? self->trace = 1 : self->trace;

        /* Check if we should trace the fs */
        ($1 == "/all" || (self->vfspath == $1)) ? self->trace = 1 : self->trace;
}

/*
 * Trace the entry point to each fop
 *
 */
fbt::fop_*:entry
/self->trace/
{
       self->path = (self->vp != NULL && self->vp->v_path) ? stringof(self->vp->v_path)
: "unknown";
        self->len = 0;
        self->off = 0;

        /* Some fops has the len in arg2 */
        (probefunc == "fop_getpage" || \
         probefunc == "fop_putpage" || \
         probefunc == "fop_none") ? self->len = arg2 : 1;

        /* Some fops has the len in arg3 */
        (probefunc == "fop_pageio" || \
         probefunc == "fop_none") ? self->len = arg3 : 1;

        /* Some fops has the len in arg4 */
        (probefunc == "fop_addmap" || \
         probefunc == "fop_map" || \
         probefunc == "fop_delmap") ? self->len = arg4 : 1;
```

continues

```
        /* Some fops has the offset in arg1 */

        (probefunc == "fop_addmap" || \
         probefunc == "fop_map" || \
         probefunc == "fop_getpage" || \
         probefunc == "fop_putpage" || \
         probefunc == "fop_seek" || \
         probefunc == "fop_delmap") ? self->off = arg1 : 1;

        /* Some fops has the offset in arg3 */
        (probefunc == "fop_close" || \
         probefunc == "fop_pageio") ? self->off = arg3 : 1;

        /* Some fops has the offset in arg4 */
        probefunc == "fop_frlock" ? self->off = arg4 : 1;

        /* Some fops has the pathname in arg1 */
        self->path = (probefunc == "fop_create" || \
         probefunc == "fop_mkdir" || \
         probefunc == "fop_rmdir" || \
         probefunc == "fop_remove" || \
         probefunc == "fop_lookup") ?
                strjoin(self->path, strjoin("/", stringof(arg1))) : self->path;
        printf("%-15s %-10s %51s %2s %8d %8d\n",
                probefunc,
                "-", self->path, "-", self->len, self->off);
        self->type = probefunc;
}

fbt::fop_*:return
/self->trace == 1/
{
        self->trace = 0;
}


/* Capture any I/O within this fop */
io:::start
/self->trace/
{
        printf("%-15s %-10s %51s %2s %8d %8u\n",
                self->type, args[1]->dev_statname,
                self->path, args[0]->b_flags & B_READ ? "R" : "W",
                args[0]->b_bcount, args[2]->fi_offset);

}

sol10# ./voptrace.d /tmp
Event           Device                                      Path  RW     Size   Offset
fop_putpage     -          /tmp/bin/i386/fastsu                    -     4096     4096
fop_inactive    -          /tmp/bin/i386/fastsu                    -        0        0
fop_putpage     -          /tmp/WEB-INF/lib/classes12.jar          -     4096   204800
fop_inactive    -          /tmp//WEB-INF/lib/classes12.jar         -        0        0
fop_putpage     -          /tmp/s10_x86_sparc_pkg.tar.Z            -     4096  7655424
fop_inactive    -          /tmp/s10_x86_sparc_pkg.tar.Z            -        0        0
fop_putpage     -          /tmp/xanadu/WEB-INF/lib/classes12.jar  -     4096   782336
fop_inactive    -          /tmp/xanadu/WEB-INF/lib/classes12.jar  -        0        0
fop_putpage     -          /tmp/bin/amd64/filebench                -     4096    36864
```

14.7 File System I/O

Two distinct methods perform file system I/O:

- read(), write(), and related system calls
- Memory-mapping of a file into the process's address space

Both methods are implemented in a similar way: Pages of a file are mapped into an address space, and then paged I/O is performed on the pages within the mapped address space. Although it may be obvious that memory mapping is performed when we memory-map a file into a process's address space, it is less obvious that the read() and write() system calls also map a file before reading or writing it. The major differences between these two methods lie in where the file is mapped and who does the mapping; a process calls mmap() to map the file into its address space for memory mapped I/O, and the kernel maps the file into the kernel's address space for read and write. The two methods are contrasted in Figure 14.9.

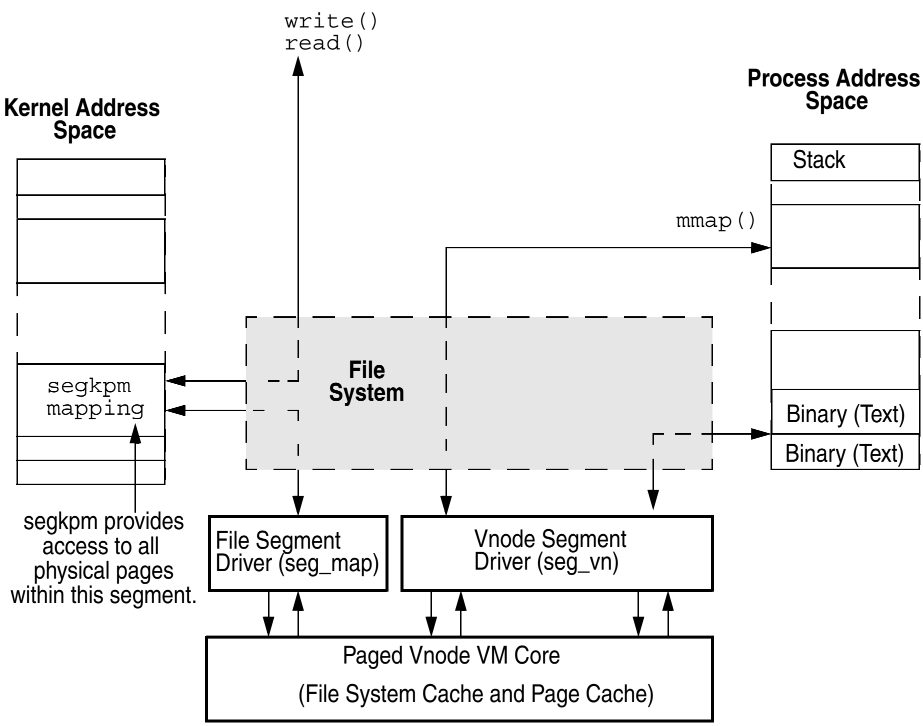

Figure 14.9 The read()/write() vs. mmap() Methods for File I/O

14.7.1 Memory Mapped I/O

A request to memory-map a file into an address space is handled by the file system vnode method vop_map() and the seg_vn memory segment driver (see Section 14.7.4). A process requests that a file be mapped into its address space. Once the mapping is established, the address space represented by the file appears as regular memory and the file system can perform I/O by simply accessing that memory.

Memory mapping of files hides the real work of reading and writing the file because the seg_vn memory segment driver quietly works with the file system to perform the I/Os without the need for process-initiated system calls. I/O is performed, in units of pages, upon reference to the pages mapped into the address space; reads are initiated by a memory access; writes are initiated as the VM system finds dirty pages in the mapped address space.

The system call mmap() calls the file system for the requested file with the vnode's vop_map() method. In turn, the file system calls the address space map function for the current address space, and the mapping is created. The protection flags passed into the mmap() system call are reduced to the subset allowed by the file permissions. If mandatory locking is set for the file, then mmap() returns an error.

Once the file mapping is created in the process's address space, file pages are read when a fault occurs in the address space. A fault occurs the first time a memory address within the mapped segment is accessed because at this point, no physical page of memory is at that location. The memory management unit causes a hardware trap for that memory segment; the memory segment calls its fault function to handle the I/O for that address. The segvn_fault() routine handles a fault for a file mapping in a process address space and then calls the file system to read in the page for the faulted address, as shown below.

```
segvn_fault (hat, seg, addr, len, type, rw) {

        for ( page = all pages in region ) {

                advise = lookup_advise (page);  /* Look up madvise settings for page */
                if (advise == MADV_SEQUENTIAL)
                        free_all_pages_up_to (page);

                /* Segvn will read at most 64k ahead */
                if ( len > PVN_GETPAGE_SZ)
                        len = PVN_GETPAGE_SZ;

                vp = segvp (seg);
                vpoff = segoff (seg);
```

continues

```
                /* Read 64k at a time if the next page is not in memory,
                 * else just a page
                 */
                if (hat_probe (addr+PAGESIZE)==TRUE)
                        len=PAGESIZE;

                /* Ask the file system for the next 64k of pages if the next*/
                VOP_GETPAGE(vp, vp_off, len,
                        &vpprot, plp, plsz, seg, addr + (vp_off - off), arw, cred)
        }
}
                                                    See usr/src/uts/common/vm/seg_vn.c
```

For each page fault, seg_vn reads in an 8-Kbyte page at the fault location. In addition, seg_vn initiates a read-ahead of the next eight pages at each 64-Kbyte boundary. Memory mapped read-ahead uses the file system cluster size (used by the read() and write() system calls) unless the segment is mapped MA_SHARED or memory advice MADV_RANDOM is set.

Recall that you can provide paging advice to the pages within a memory mapped segment by using the madvise system call. The madvise system call and (as in the example) the advice information are used to decide when to free behind as the file is read.

Modified pages remain unwritten to disk until the fsflush daemon passes over the page, at which point they will be written out to disk. You can also use the memcntl() system call to initiate a synchronous or asynchronous write of pages.

14.7.2 read() and write() System Calls

The vnode's vop_read() and vop_write() methods implement reading and writing with the read() and write() system calls. As shown in Figure 14.10, the seg_map segment driver directly accesses a page by means of the seg_kpm mapping of the system's physical pages within the kernel's address space during the read() and write() system calls. The read and write file system calls copy data to or from the process during a system call to a portion of the file that is mapped into the kernel's address space by seg_kpm. The seg_map driver maintains a cache of addresses between the vnode/offset and the virtual address where the page is mapped.

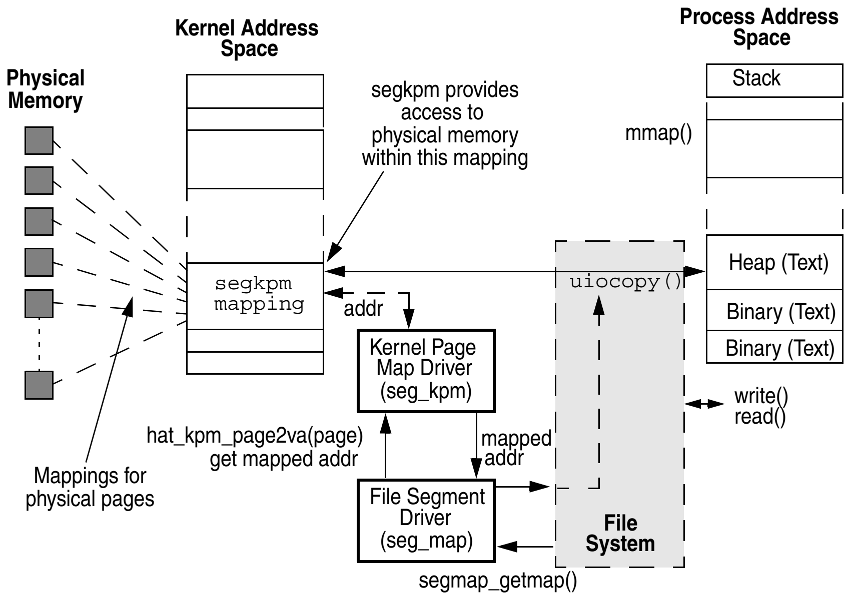

Figure 14.10 File System Data Movement with `seg_map/seg_kpm`

14.7.3 The `seg_kpm` Driver

The `seg_kpm` driver provides a fast mapping for physical pages within the kernel's address space. It is used by file systems to provide a virtual address when copying data to and from the user's address space for file system I/O. The use of this `seg_kpm` mapping facility is new for Solaris 10.

Since the available virtual address range in a 64-bit kernel is always larger than physical memory size, the entire physical memory can be mapped into the kernel. This eliminates the need to map/unmap pages every time they are accessed through `segmap`, significantly reducing code path and the need for TLB shoot-downs. In addition, `seg_kpm` can use large TLB mappings to minimize TLB miss overhead.

14.7.4 The `seg_map` Driver

The `seg_map` driver maintains the relationship between pieces of files into the kernel address space and is used only by the file systems. Every time a `read` or `write` system call occurs, the `seg_map` segment driver locates the virtual address space where the page of the file can be mapped. The system call can then copy the data to or from the user address space.

The `seg_map` segment provides a full set of segment driver interfaces (see Section 9.5); however, the file system directly uses a small subset of these interfaces without going through the generic segment interface. The subset handles the bulk of the work that is done by the `seg_map` segment for file read and write operations. The functions used by the file systems are shown on page 714.

The `seg_map` segment driver divides the segment into block-sized slots that represent blocks in the files it maps. The `seg_map` block size for the Solaris kernel is 8,192 bytes. A 128-Mbyte `segkmap` segment would, for example, be divided into 128-MB/8-KB slots, or 16,384 slots. The `seg_map` segment driver maintains a hash list of its page mappings so that it can easily locate existing blocks. The list is based on file and offsets. One list entry exists for each slot in the `segkmap` segment. The structure for each slot in a `seg_map` segment is defined in the `<vm/segmap.h>` header file, shown below.

```
/*
 * Machine independent per instance kpm mapping structure
 */
struct kpme {
        struct kpme     *kpe_next;
        struct kpme     *kpe_prev;
        struct page     *kpe_page;      /* back pointer to (start) page */
};
                                             See usr/src/uts/common/vm/kpm.h


/*
 * Each smap struct represents a MAXBSIZE sized mapping to the
 * <sm_vp, sm_off> given in the structure.  The location of the
 * the structure in the array gives the virtual address of the
 * mapping. Structure rearranged for 64bit sm_off.
 */
struct  smap {
        kmutex_t        sm_mtx;         /* protect non-list fields */
        struct  vnode   *sm_vp;         /* vnode pointer (if mapped) */
        struct  smap    *sm_hash;       /* hash pointer */
        struct  smap    *sm_next;       /* next pointer */
        struct  smap    *sm_prev;       /* previous pointer */
        u_offset_t      sm_off;         /* file offset for mapping */
        ushort_t        sm_bitmap;      /* bit map for locked translations */
        ushort_t        sm_refcnt;      /* reference count for uses */
        ushort_t        sm_flags;       /* smap flags */
        ushort_t        sm_free_ndx;    /* freelist */
#ifdef  SEGKPM_SUPPORT
        struct kpme     sm_kpme;        /* segkpm */
#endif
};
                                          See usr/src/uts/common/vm/segmap.h
```

The key `smap` structures are

- **`sm_vp`.** The file (`vnode`) this slot represents (if slot not empty)
- **`sm_hash`, `sm_next`, `sm_prev`.** Hash list reference pointers
- **`sm_off`.** The file (`vnode`) offset for a block-sized chunk in this slot in the file

- **sm_bitmap.** Bitmap to maintain translation locking
- **sm_refcnt.** The number of references to this mapping caused by concurrent reads

The important fields in the smap structure are the file and offset fields, sm_vp and sm_off. These fields identify which page of a file is represented by each slot in the segment.

An example of the interaction between a file system read and segmap is shown in Figure 14.11.

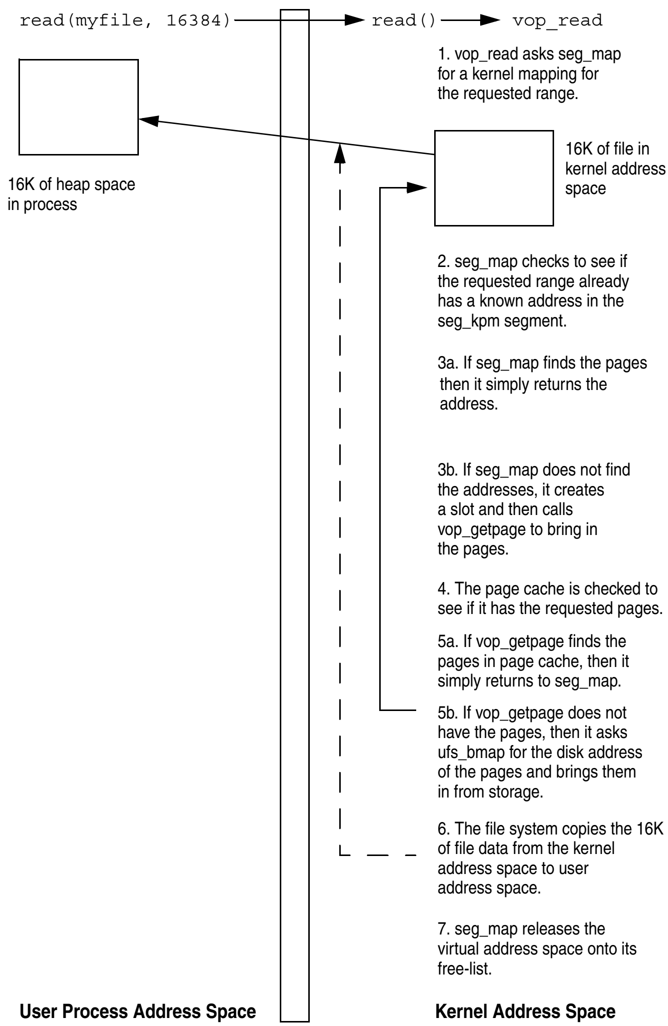

Figure 14.11 vop_read() segmap Interaction

A read system call invokes the file-system-dependent vop_read function. The vop_read method calls into the seg_map segment to locate a virtual address in the kernel address space via segkpm for the file and offset requested with the segmap_getmapflt() function. The seg_map driver determines whether it already has a slot for the page of the file at the given offset by looking into its hashed list of mapping slots. Once a slot is located or created, an address for the page is located, and segmap then calls back into the file system with vop_getpage() to soft-initiate a page fault to read in a page at the virtual address of the seg_map slot. While the segmap_getmapflt() routine is still running, the page fault is initiated by a call to segmap_fault(), which in turn calls back into the file system with vop_getpage().

The file system's vop_getpage() routine handles the task of bringing the requested range of the file (vnode, offset, and length) from disk into the virtual address and length passed into the vop_getpage() function.

Once the page is read by the file system, the requested range is copied back to the user by the uio_move() function. Then, the file system releases the slot associated with that block of the file with the segmap_release() function. At this point, the slot is not removed from the segment because we may need the same file and offset later (effectively caching the virtual address location); instead, it is added onto a seg_map free list so it can be reclaimed or reused later.

Writing is a similar process. Again, segmap_getmap() is called to retrieve or create a mapping for the file and offset, the I/O is done, and the segmap slot is released. An additional step is involved if the file is being extended or a new page is being created within a hole of a file. This additional step calls the segmap_pagecreate() function to create and lock the new pages, then calls segmap_pageunlock() to unlock the pages that were locked during the page_create().

The key segmap functions are shown below.

```
caddr_t segmap_getmapflt(struct seg *seg,
                         struct vnode *vp,
                         u_offset_t off,
                         size_t len,
                         int forcefault,
                         enum seg_rw rw);

Retrieves an address in the kernel's address space for a range of the file at the given
offset and length. segmap_getmap allocates a MAXBSIZE big slot to map the vnode vp in
the range <off, off + len). off doesn't need to be MAXBSIZE aligned. The return address
is always MAXBSIZE aligned. If forcefault is nonzero and the MMU translations haven't
yet been created, segmap_getmap will call segmap_fault(..., F_INVAL, rw) to create
them.

int segmap_release(struct seg *seg, caddr_t addr, uint_t flags);

Releases the mapping for a given file at a given address.
```

continues

```
int segmap_pagecreate(struct seg *seg, caddr_t addr, size_t len, int softlock);

Creates new page(s) of memory and slots in the seg_map segment for a given file. Used
for extending files or writing to holes during a write. This function creates pages
(without using VOP_GETPAGE) and loads up translations to them. If softlock is TRUE, then
set things up so that it looks like a call to segmap_fault with F_SOFTLOCK.  Returns 1
if a page is created by calling page_create_va(), or 0 otherwise.

All fields in the generic segment (struct seg) are considered to be read-only for "seg-
map" even though the kernel address space (kas) may not be locked; hence, no lock is
needed to access them.

void segmap_pageunlock(struct seg *seg, caddr_t addr, size_t len, enum seg_rw rw);

Unlocks pages in the segment that was locked during segmap_pagecreate().

                                                   See usr/src/uts/common/vm/segmap.h
```

We can observe the `seg_map` slot activity with the kstat statistics that are collected for the `seg_map` segment driver. These statistics are visible with the `kstat` command, as shown below.

```
sol10$ kstat -n segmap
module: unix                            instance: 0
name:   segmap                          class:    vm
        crtime                          42.268896913
        fault                           352197
        faulta                          0
        free                            1123987
        free_dirty                      50836
        free_notfree                    2073
        get_nofree                      0
        get_nomtx                       0
        get_reclaim                     5644590
        get_reuse                       1356990
        get_unused                      0
        get_use                         386
        getmap                          7005644
        pagecreate                      1375991
        rel_abort                       0
        rel_async                       291640
        rel_dontneed                    291640
        rel_free                        7054
        rel_write                       304570
        release                         6694020
        snaptime                        1177936.33212098
        stolen                          0
```

Table 14.5 describes the `segmap` statistics.

Table 14.5 Statistics from the seg_map Segment Driver

Field Name	Description
fault	The number of times segmap_fault was called, usually as a result of a read or write system call.
faulta	The number of times the segmap_faulta function was called. It is called to initiate asynchronous paged I/O on a file.
getmap	The number of times the segmap_getmap function was called. It is called by the read and write system calls each time a read or write call is started. It sets up a slot in the seg_map segment for the requested range on the file.
get_use	The number of times a valid mapping was found in seg_map, which was also already referenced by another user.
get_reclaim	The number of times a valid mapping was found in seg_map, which was otherwise unused.
get_reuse	The number of times getmap deleted the mapping in a non-empty slot and created a new mapping for the file and offset requested.
get_unused	Not used—always zero.
get_nofree	The number of times a request for a slot was made and none was available on the internal free list of slots. This number is usually zero because each slot is put on the free list when release is called at the end of each I/O. Hence, ample free slots are usually available.
rel_async	The slot was released with a delayed I/O on it.
rel_write	The slot was released as a result of a write system call.
rel_free	The slot was released, and the VM system was told that the page may be needed again but to free it and retain its file/offset information. These pages are placed on the cache list tail so that they are not the first to be reused.
rel_abort	The slot was released and asked to be removed from the seg_map segment as a result of a failed aborted write.
rel_dontneed	The slot was released, and the VM system was told to free the page because it won't be needed again. These pages are placed on the cache list head so they will be reused first.
released	The slot was released and the release was not affected by rel_abort, rel_async, or rel_write.
pagecreate	Pages were created in the segmap_pagecreate function.

continues

Table 14.5 Statistics from the `seg_map` Segment Driver (*continued*)

Field Name	Description
`free_notfree`	An attempt was made to free a page which was still mapped
`free_dirty`	Pages that were dirty were freed from `segmap`.
`free`	Pages that were clean were freed from `segmap`.
`stolen`	A `smap` slot was taken during a `getmap`.
`get_nomtx`	This field is not used.

14.7.5 Interaction between segmap and segkpm

The following three examples show the code flow through the file system into `segmap` for three important cases:

1. The requested `vnode`/offset has a cached slot in `seg_map`, and the physical page is in the page cache.
2. The requested `vnode`/offset does not have a cached slot in `seg_map`, but the physical page is in the page cache.
3. The requested vnode/offset is not in either.

```
Hit in page cache and segmap:
-> ufs_read                          read() Entry point into UFS
  -> segmap_getmapflt                Locate the segmap slot for the vnode/off
    -> hat_kpm_page2va               Identify the virtual address for the vnode/off
    <- hat_kpm_page2va
  <- segmap_getmapflt
  -> uiomove                         Copy the data from the segkpm address to userland
  <- uiomove
  -> segmap_release                  Release the segmap slot
    -> hat_kpm_vaddr2page            Locate the page by looking up its address
    <- hat_kpm_vaddr2page
    -> segmap_smapadd                Add the segmap slot to the reuse pool
    <- segmap_smapadd
  <- segmap_release
<- ufs_read
                                                          See examples/segkpm.d
```

```
Hit in page cache, miss in segmap:
-> ufs_read                          read() Entry point into UFS
  -> segmap_getmapflt                Locate the segmap slot for the vnode/off
    -> get_free_smp                  Find a segmap slot that can be reused
      -> grab_smp                    Flush out the old segmap slot identity
        -> segmap_hashout
        <- segmap_hashout
        -> hat_kpm_page2va           Identify the virtual address for the vnode/off
        <- hat_kpm_page2va
```

continues

```
      <- grab_smp
      -> segmap_pagefree             Put the page back on the cachelist
      <- segmap_pagefree
    <- get_free_smp
    -> segmap_hashin                 Set up the segmap slot for the new vnode/off
    <- segmap_hashin
    -> segkpm_create_va              Create a virtual address for this vnode/off
    <- segkpm_create_va
    -> ufs_getpage                   Find the page already in the page-cache
    <- ufs_getpage
    -> hat_kpm_mapin                 Reuse a mapping for the page in segkpm
    <- hat_kpm_mapin
  <- segmap_getmapflt
  -> uiomove                         Copy the data from the segkpm address to userland
  <- uiomove
  -> segmap_release                  Add the segmap slot to the reuse pool
    -> hat_kpm_vaddr2page
    <- hat_kpm_vaddr2page
    -> segmap_smapadd
    <- segmap_smapadd
  <- segmap_release
<- ufs_read
                                                           See examples/segkpm.d
```

```
Miss in page cache, miss in segmap:
-> ufs_read                          read() Entry point into UFS
  -> segmap_getmapflt                Locate the segmap slot for the vnode/off
    -> get_free_smp                  Find a segmap slot that can be reused
      -> grab_smp                    Flush out the old segmap slot identity
        -> segmap_hashout
        <- segmap_hashout
        -> hat_kpm_page2va           Identify the virtual address for the vnode/off
        <- hat_kpm_page2va
        -> hat_kpm_mapout            Unmap the old slot's page(s)
        <- hat_kpm_mapout
      <- grab_smp
      -> segmap_pagefree
      <- segmap_pagefree
    <- get_free_smp
    -> segmap_hashin                 Set up the segmap slot for the new vnode/off
    <- segmap_hashin
    -> segkpm_create_va              Create a virtual address for this vnode/off
    <- segkpm_create_va
    -> ufs_getpage                   Call the file system getpage() to read in the page
      -> bdev_strategy               Initiate the physical read
      <- bdev_strategy
    <- ufs_getpage
    -> hat_kpm_mapin                 Create a mapping for the page in segkpm
      -> sfmmu_kpm_mapin
        -> sfmmu_kpm_getvaddr
        <- sfmmu_kpm_getvaddr
      <- sfmmu_kpm_mapin
      -> sfmmu_kpme_lookup
      <- sfmmu_kpme_lookup
      -> sfmmu_kpme_add
      <- sfmmu_kpme_add
    <- hat_kpm_mapin
  <- segmap_getmapflt
  -> uiomove                         Copy the data from the segkpm address to userland
  <- uiomove
  -> segmap_release                  Add the segmap slot to the reuse pool
```

continues

```
    -> get_smap_kpm
      -> hat_kpm_vaddr2page
      <- hat_kpm_vaddr2page
    <- get_smap_kpm
    -> segmap_smapadd
    <- segmap_smapadd
  <- segmap_release
<- ufs_read
                                                          See examples/segkpm.d
```

14.8 File Systems and Memory Allocation

File system caching has been implemented as an integrated part of the Solaris virtual memory system since as far back as SunOS 4.0. This has the great advantage of dynamically using available memory as a file system cache. While this integration has many positive advantages (like being able to speed up some I/O-intensive applications by as much as 500 times), there were some historic side effects: Applications with a lot of file system I/O could swamp the memory system with demand for memory allocations, pressuring the memory system so much that memory pages were aggressively stolen from important applications. Typical symptoms of this condition were that everything seemed to "slow down" when file I/O was occurring and that the system reported it was constantly out of memory. In Solaris 2.6 and 7, the paging algorithms were updated to steal only file system pages unless there was a real memory shortage, as part of the feature named "priority paging." This meant that although there was still significant pressure from file I/O and high "scan rates," applications didn't get paged out or suffer from the pressure. A healthy Solaris 7 system still reported it was out of memory, but performed well.

14.8.1 Solaris 8—Cyclic Page Cache

Starting with Solaris 8, we significantly enhanced the architecture to solve the problem more effectively. We changed the file system cache so that it steals memory from itself, rather than from other parts of the system. Hence, a system with a large amount of file I/O will remain in a healthy virtual memory state—with large amounts of visible free memory and, since the page scanner doesn't need to run, with no aggressive scan rates. Since the page scanner isn't constantly required to free up large amounts of memory, it no longer limits file-system-related I/O throughput. Other benefits of the enhancement are that applications that need to allocate a large amount of memory can do so by efficiently consuming it directly from the file system cache. For example, starting Oracle with a 50-Gbyte SGA now takes less than a minute, compared to the 20–30 minutes with the prior implementation.

14.8.2 The Old Allocation Algorithm

To keep this explanation relatively simple, let's briefly look at what used to happen with Solaris 7, even with priority paging.

The file system consumes memory from the free lists every time a new page is read from disk (or wherever) into the file system. The more pages read, the more pages depleted from the system's free list (the central place where memory is kept for reuse). Eventually (sometimes rather quickly), the free memory pool is depleted. At this point, if there is enough pressure, further requests for new memory pages are blocked until the free memory pool is replenished by the page scanner. The page scanner scans inefficiently through all of memory, looking for pages it can free, and slowly refills the free list, but only by enough to satisfy the immediate request. Processes resume for a short time, and then stop as they again run short on memory. The page scanner is a bottleneck in the whole memory life cycle.

In Figure 14.12, we can see the file system's cache mechanism (`segmap`) consuming memory from the free list until the list is depleted. After those pages are

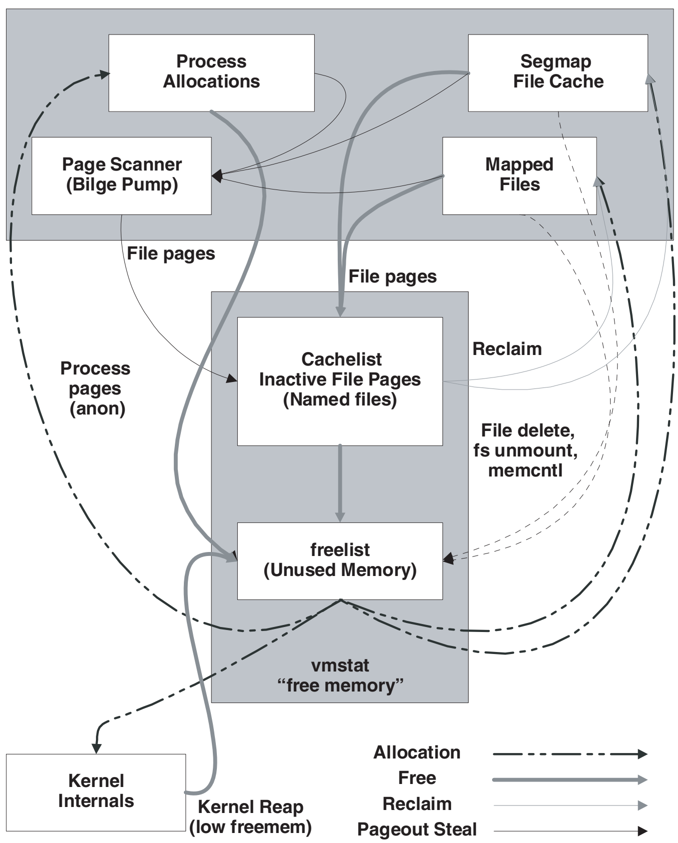

Figure 14.12 Life Cycle of Physical Memory

used, they are kept around, but they are only immediately accessible by the file system cache in the direct reuse case; that is, if a file system cache hit occurs, then they can be "reclaimed" into `segmap` to avoid a subsequent physical I/O. However, if the file system cache needs a new page, there is no easy way of finding these pages; rather, the page scanner is used to stumble across them. The page scanner effectively "bilges out" the system, blindly looking for new pages to refill the free list. The page scanner has to fill the free list at the same rate at which the file system is reading new pages—and thus is a single point of constraint in the whole design.

14.8.3 The New Allocation Algorithm

The new algorithm uses a central list to place the inactive file cache (that which isn't immediately mapped anywhere), so that it can easily be used to satisfy new memory requests. This is a very subtle change, but one with significant demonstrable effects. First, the file system cache now appears as a single age-ordered FIFO: Recently read pages are placed at the tail of the list, and new pages are consumed from the head. While on the list, the pages remain as valid cached portions of the file, so if a read cache hit occurs, they are simply removed from wherever they are on the list. This means that pages that are accessed often (cache hit often) are frequently moved to the tail of the list, and only the oldest and least used pages migrate to the head as candidates for freeing.

The cache list is linked to the free list, such that if the free list is exhausted, then pages are taken from the head of the cache list and their contents discarded. New page requests are requested from the free list, but since this list is often empty, allocations occur mostly from the head of the cache list, consuming the oldest file system cache pages. The page scanner doesn't need to get involved, thus eliminating the paging bottleneck and the need to run the scanner at high rates (and hence, not wasting CPU either).

If an application process requests a large amount of memory, it too can take from the cache list via the free list. Thus, an application can take a large amount of memory from the file system cache without needing to start the page scanner, resulting in substantially faster allocation.

14.8.4 Putting It All Together: The Allocation Cycle

The most significant central pool physical memory is the free list. Physical memory is placed on the free list in page-size chunks when the system is first booted and then consumed as required. Three major types of allocations occur from the free list, as shown in Figure 14.12.

Anonymous/Process Allocations. Anonymous memory, the most common form of allocation from the free list, is used for most of a process's memory allocation, including heap and stack. Anonymous memory also fulfills shared memory mappings allocations. A small amount of anonymous memory is also used in the kernel for items such as thread stacks. Anonymous memory is pageable and is returned to the free list when it is unmapped or if it is stolen by the page scanner daemon.

File System Page Cache. The page cache caches file data for file systems. The file system page cache grows on demand to consume available physical memory as a file cache and caches file data in page-size chunks. Pages are consumed from the free list as files are read into memory. The pages then reside in one of three places: on the `segmap` cache, in a process's address space to which they are mapped, or on the cache list.

The cache list is the heart of the page cache. All unmapped file pages reside on the cache list. Working in conjunction with the cache list are mapped files and the segmap cache.

Think of the `segmap` file cache as the fast first-level file system read/write cache. `segmap` is a cache that holds file data read and written through the read and write system calls. Memory is allocated from the free list to satisfy a read of a new file page, which then resides in the `segmap` file cache. File pages are eventually moved from the `segmap` cache to the cache list to make room for more pages in the `segmap` cache.

The cachelist is typically 12% of the physical memory size on SPARC systems. The `segmap` cache works in conjunction with the system cache list to cache file data. When files are accessed—through the read and write system calls—up to 12% of the physical memory file data resides in the `segmap` cache and the remainder is on the cache list.

Memory mapped files also allocate memory from the free list and remain allocated in memory for the duration of the mapping or unless a global memory shortage occurs. When a file is unmapped (explicitly or with `madvise`), file pages are returned to the cache list.

The cache list operates as part of the free list. When the free list is depleted, allocations are made from the oldest pages in the cache list. This allows the file system page cache to grow to consume all available memory and to dynamically shrink as memory is required for other purposes.

Kernel Allocations. The kernel uses memory to manage information about internal system state, for example, memory that holds the list of processes in the system. The kernel allocates memory from the free list for these purposes with its own allocators: `vmem` and `slab`. However, unlike process and file allocations, the kernel seldom returns memory to the free list; memory is allocated and freed between kernel subsystems and the kernel allocators. Memory is consumed from the free list only when the total kernel allocation grows.

Memory allocated to the kernel is mostly nonpageable and so cannot be managed by the system page scanner daemon. Memory is returned to the system free list proactively by the kernel's allocators when a global memory shortage occurs. See Chapter 11.

14.9 Path-Name Management

All but a few of the `vnode` methods operate on `vnode` pointers rather than on path names. Before calling file system `vnode` methods, the `vnode` framework first converts path names and file descriptors into `vnode` references. File descriptors may be directly translated into `vnodes` for the files they referenced, whereas path names must be converted into `vnodes` by a lookup of the path-name components and a reference to the underlying file. The file-system-independent `lookuppn()` function converts path names to `vnodes`. An additional wrapper, `lookupname()`, converts path names from user-mode system calls.

14.9.1 The `lookuppn()` Method

Given a path name, the `lookuppn()` method attempts to return a pointer to the `vnode` the path represents. If the vnode is already available, then a new reference to the vnode is established. If no vnode is available, one is created. The `lookuppn()` function decomposes the components of the path name, separating them by "/" and ".", and calls the file-system-specific `vop_lookup()` method (see below) for each component of the path name.

If the path name begins with a "/", path-name traversal starts at the user's root directory. Otherwise, it starts at the `vnode` pointed to by the user's current directory. `lookuppn()` traverses the path one component at a time, using the `vop_lookup()` `vnode` method.

If a directory `vnode` has `v_vfsmountedhere` set, then it is a mount point. If `lookuppn()` encounters a mount point while going down the file system tree, then it follows the `vnode`'s `v_vfsmountedhere` pointer to the mounted file system and calls the `vfs_root()` method to obtain the root `vnode` for the file system. Path-name traversal then continues from this point.

If `lookuppn()` encounters a root `vnode` (`VROOT` flag in `v_flag` set) when following "..", then `lookuppn()` follows the `vfs_vnodecovered` pointer in the `vnode's` associated `vfs` to obtain the covered `vnode`.

If `lookuppn()` encounters a symbolic link, then it calls the `vn_readlink()` `vnode` method to obtain the symbolic link. If the symbolic link begins with a "/", the path-name traversal is restarted from the root directory; otherwise, the traversal continues from the last directory. The caller of `lookuppn()` specifies whether the last component of the path name is to be followed if it is a symbolic link.

This procedure continues until the path name is exhausted or an error occurs. When `lookuppn()` completes, it returns a `vnode` representing the desired file.

14.9.2 The `vop_lookup()` Method

The `vop_lookup()` method searches a directory for a path-name component matching the supplied path name. The `vop_lookup()` method accepts a directory `vnode` and a string path-name component as an argument and returns a `vnode` pointer to the `vnode` representing the file. If the file cannot be located, then `ENOENT` is returned.

Many regular file systems will first check the directory name lookup cache, and if an entry is found there, the entry is returned. If the entry is not found in the directory name cache, then a real lookup of the file is performed.

14.9.3 The `vop_readdir()` Method

The `vop_readdir()` method reads chunks of the directory into a `uio` structure. Each chunk can contain as many entries as will fit within the size supplied by the `uio` structure. The `uio_resid` structure member shows the size of the `getdents` request in bytes, which is divided by the size of the directory entry made by the `vop_readdir()` method to calculate how many directory entries to return.

Directories are read from disk with the buffered kernel file functions `fbread` and `fbwrite`. These functions, described below, are provided as part of the generic file system infrastructure.

```
/*
 * A struct fbuf is used to get a mapping to part of a file using the
 * segkmap facilities.  After you get a mapping, you can fbrelse() it
 * (giving a seg code to pass back to segmap_release), you can fbwrite()
 * it (causes a synchronous write back using the file mapping information),
 * or you can fbiwrite it (causing indirect synchronous write back to
 * the block number given without using the file mapping information).
 */

struct fbuf {
        caddr_t fb_addr;
        uint_t  fb_count;
};
```

continues

```
extern int fbread(struct vnode *, offset_t, uint_t, enum seg_rw, struct fbuf **);

Returns a pointer to locked kernel virtual address for the given <vp, off> for len
bytes. The read may not cross a boundary of MAXBSIZE (8192) bytes.

extern void fbzero(struct vnode *, offset_t, uint_t, struct fbuf **);

Similar to fbread(), but calls segmap_pagecreate(), not segmap_fault(), so that SOFT-
LOCK can create the pages without using VOP_GETPAGE(). Then, fbzero() zeroes up to the
length rounded to a page boundary.

extern int fbwrite(struct fbuf *);

Direct write.

extern int fbiwrite(struct fbuf *, struct vnode *, daddr_t bn, int bsize);

Writes directly and invalidates pages.

extern int fbdwrite(struct fbuf *);

Delayed write.

extern void fbrelse(struct fbuf *, enum seg_rw);

Releases fbp.
                                                   See usr/src/uts/common/sys/fbuf.h
```

14.9.4 Path-Name Traversal Functions

Several path-name manipulation functions assist with decomposition of path names. The path-name functions use a path-name structure, shown below, to pass around path-name components.

```
/*
 * Pathname structure.
 * System calls that operate on path names gather the path name
 * from the system call into this structure and reduce it by
 * peeling off translated components.  If a symbolic link is
 * encountered the new path name to be translated is also
 * assembled in this structure.
 *
 * By convention pn_buf is not changed once it's been set to point
 * to the underlying storage; routines which manipulate the path name
 * do so by changing pn_path and pn_pathlen.  pn_pathlen is redundant
 * since the path name is null-terminated, but is provided to make
 * some computations faster.
 */
typedef struct pathname {
        char    *pn_buf;                /* underlying storage */
        char    *pn_path;               /* remaining pathname */
        size_t  pn_pathlen;             /* remaining length */
        size_t  pn_bufsize;             /* total size of pn_buf */
} pathname_t;
                                                See usr/src/uts/common/sys/pathname.h
```

The path-name functions are shown below.

```
void pn_alloc(struct pathname *pnp);

Allocates a new path-name buffer.Structure is typically an automatic variable in call-
ing routine for convenience.May sleep in the call to kmem_alloc() and so must not be
called from interrupt level.

int pn_get(char *str, enum uio_seg seg, struct pathname *pnp);

Copies path-name string from user and mounts arguments into a struct path name.

int pn_set(struct pathname *pnp, char *path);

Sets a path name to the supplied string.

int pn_insert(struct pathname *pnp, struct pathname *sympnp, size_t complen);

Combines two argument path names by putting the second argument before the first in the
first's buffer.  This isn't very general; it is designed specifically for symbolic link
processing. This function copies the symlink in-place in the path name.  This is to
ensure that vnode path caching remains correct.  At the point where this is called (from
lookuppnvp), we have called pn_getcomponent(), found it is a symlink, and are now
replacing the contents.  The complen parameter indicates how much of the path name to
replace. If the symlink is an absolute path, then we overwrite the entire contents of
the pathname.

int pn_getsymlink(vnode_t *vp, struct pathname *pnp, cred_t *crp);

Follows a symbolic link for a path name.

int pn_getcomponent(struct pathname *pnp, char *component);

Extracts the next delimited path-name component.

void pn_setlast(struct pathname *pnp);

Sets pn_path to the last component in the path name, updating pn_pathlen.  If pathname
is empty or degenerate, leaves pn_path pointing at NULL char.The path name is explicitly
null-terminated so that any trailing slashes are effectively removed.

void pn_skipslash(struct pathname *pnp);

Skips over consecutive slashes in the path name.

int pn_fixslash(struct pathname *pnp);

Eliminates any trailing slashes in the path name.

int pn_addslash(struct pathname *pnp);

Add sa slash to the end of the path name, if it will fit.Return ENAMETOOLONG if it
won't.

void pn_free(struct pathname *pnp);

Frees a struct path name.
                                          See usr/src/uts/common/sys/pathname.h
```

14.10 The Directory Name Lookup Cache

The directory name lookup cache (DNLC) is based on BSD 4.2 code. It was ported to Solaris 2.0 and threaded and has undergone some significant revisions. Most of the enhancements to the DNLC have been performance and threading, but a few visible changes are noteworthy. Table 14.6 summarizes the important changes to the DNLC.

Table 14.6 Solaris DNLC Changes

Year	OS Rev	Comment
1984	BSD 4.2	14-character name maximum
1990	Solaris 2.0	31-character name maximum
1994	Solaris 2.4	Performance (new locking/search algorithm)
1998	Solaris 7	Variable name length
2001	Solaris 8	Directory caching and negative entry caching

14.10.1 DNLC Operation

Each time we open a file, we call the `open()` system call with a path name. That path name must be translated to a vnode by the process of reading the directory and finding the corresponding name that matches the requested name. To prevent us from having to reread the directory every time we translate the path name, we cache the containing directory `vnode`/file-name name and the corresponding `vnode` mappings in the directory name lookup cache. The cache is managed as an LRU cache, so that most frequently used directory entries are kept in the cache.

The Solaris DNLC replaces the original SVR4 DNLC algorithm. It yielded a significant improvement in scalability. The Solaris 2.4 DNLC algorithm removed LRU list lock contention by eliminating the LRU list completely. In addition, the list takes into account the number of references to a `vnode` and whether the `vnode` has any pages in the page cache. This design allows the DNLC to cache the most relevant `vnodes`, rather than just the most frequently looked-up `vnodes`.

Figure 14.13 illustrates the Solaris DNLC.

The lookup algorithm uses a rotor pointing to a hash chain; the rotor switches chains for each invocation of `dnlc_enter()` that needs a new entry. The algorithm starts at the end of the chain and takes the first entry that has a `vnode` reference count of `1` or no pages in the page cache. In addition, during lookup, entries are moved to the front of the chain so that each chain is sorted in LRU order.

The DNLC was enhanced to use the kernel memory allocator to allocate a variable length string for the name; this change removed the 31-character limit. In the

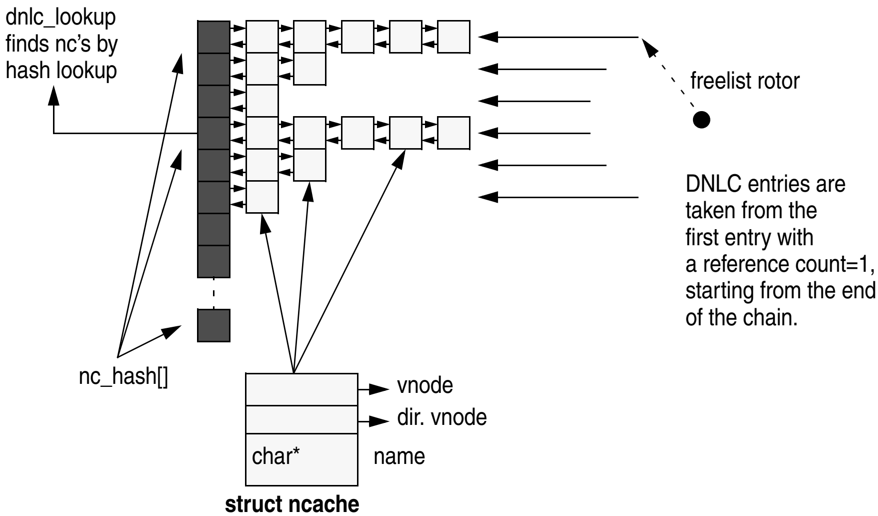

Figure 14.13 Solaris DNLC

Solaris 7 DNLC structure, shown in Figure 14.13, note that the name field has changed from a static structure to a pointer.

The number of entries in the DNLC is controlled by the `ncsize` parameter, which is initialized to 4 * (`max_nprocs` + `maxusers`) + 320 at system boot.

Most of the DNLC work is done with two functions: `dnlc_enter()` and `dnlc_lookup()`. When a file system wants to look up the name of a file, it first checks the DNLC with the `dnlc_lookup()` function, which queries the DNLC for an entry that matches the specified file name and directory `vnode`. If no entry is found, `dnlc_lookup` fails and the file system reads the directory from disk. When the file name is found, it is entered into the DNLC with the `dnlc_enter()` function. The DNLC stores entries on a hashed list (`nc_hash[]`) by file name and directory `vnode` pointer. Once the correct `nc_hash` chain is identified, the chain is searched linearly until the correct entry is found.

The original BSD DNLC had 8 `nc_hash` entries, which was increased to 64 in SunOS 4.x. Solaris 2.0 sized the `nc_hash` list at boot, attempting to make the average length of each chain no more than 4 entries. It used the total DNLC size, `ncsize`, divided by the average length to establish the number of `nc_hash` entries. Solaris 2.3 had the average length of the chain dropped to 2 in an attempt to increase DNLC performance; however, other problems, related to the LRU list locking and described below, adversely affected performance.

Each entry in the DNLC is also linked to an LRU list, in order of last use. When a new entry is added into the DNLC, the algorithm replaces the oldest entry from the LRU list with the new file name and directory `vnode`. Each time a lookup is done, the DNLC also takes the entry from the LRU and places it at the end of the list so that it won't be reused immediately. The DNLC uses the LRU list to attempt to keep most-used references in the cache. Although the DNLC list had been made short, the LRU list still caused contention because it required that a single lock be held around the entire chain.

14.10.2 Primary DNLC Support Functions

The primary DNLC support functions are summarized below.

```
void    dnlc_enter(vnode_t *dvp, char *name, vnode_t *vp, cred_t *cr);

Enters a new ncache entry into the DNLC for the given name and directory vnode pointer.
If an entry already exists for the name and directory pointer, the function returns with
no action.


void    dnlc_update(vnode_t *dvp, char *name, vnode_t *vp, cred_t *cr);

Enters a new ncache entry into the DNLC for the given name and directory vnode pointer.
If an entry already exists for the name and directory pointer but the vnode is differ-
ent, then the entry is overwritten. Otherwise, the function returns with no action.

vnode_t *dnlc_lookup(vnode_t *dvp, char *name, cred_t *cr);

Locates an ncache entry that matches the supplied name and directory vnode pointer.
Returns a pointer to the vnode for that entry or returns NULL.


void    dnlc_purge(void);

Called by the vfs framework when an umountall() is called.


void    dnlc_purge_vp(vnode_t *vp);

Purges all entries matching the vnode supplied.


int     dnlc_purge_vfsp(vfs_t *vfs, int);

Purges all entries matching the vfs supplied.


void    dnlc_remove(vnode_t *vp, char *name);

Removes the entry matching the supplied name and directory vnode pointer.


int     dnlc_fs_purge1(struct vnodeops *vop);

Purge 1 entry from the dnlc that is part of the file system(s) represented by 'vop'. The
purpose of this routine is to allow users of the dnlc to free a vnode that is being held
by the dnlc.If we find a vnode that we release which will result in freeing the under-
lying vnode (count was 1), return 1, 0 if no appropriate vnodes found.

                                              See usr/src/uts/common/sys/dnlc.h
```

14.10.3 DNLC Negative Cache

The DNLC has support for negative caching. Some applications repeatedly test for the existence or nonexistence of a file (for example, a lock file or a results file). In addition, many shell PATH variables list directories that don't exist. For these applications, caching the fact that the file doesn't exist (negative caching) is a performance boost.

The DNLC negative cache follows the NFS negative-cache solution. It defines a negative cache `vnode` that is initialized with the reference count set to `1` so that `VOP_INACTIVE()` never gets called on it.

```
vnode_t negative_cache_vnode;

#define DNLC_NO_VNODE &negative_cache_vnode

                                          See usr/src/uts/common/sys/dnlc.h
```

File systems were updated in Solaris 8 to use negative caching so that each `dnlc_lookup()` checks for a `DNLC_NO_VNODE` return. Negative cache entries will be added when directory lookups fail, and will be invalidated by `dnlc_update()` when a real file of that name is added.

14.10.4 DNLC Directory Cache

The directory cache adds a new set of interfaces to the DNLC to cache entire directories. The directory cache eliminates performance bottlenecks for directories with tens of thousands of files. This helps performance when the file name repeatedly changes and when new files are created. It removes the need to search the entire directory to find out if the file name already exists. It turns out that mail and news spool directories see this scenario all the time.

The DNLC structure is shown below.

```
/*
 * This structure describes the elements in the cache of recent
 * names looked up.
 *
 * Note namlen is a uchar_t to conserve space
 * and alignment padding. The max length of any
 * pathname component is defined as MAXNAMELEN
 * which is 256 (including the terminating null).
 * So provided this doesn't change, we don't include the null,
 * we always use bcmp to compare strings, and we don't start
 * storing full names, then we are ok. The space savings are worth it.
 */
```

continues

```
typedef struct ncache {
        struct ncache *hash_next;       /* hash chain, MUST BE FIRST */
        struct ncache *hash_prev;
        struct vnode *vp;               /* vnode the name refers to */
        struct vnode *dp;               /* vnode of parent of name */
        int hash;                       /* hash signature */
        uchar_t namlen;                 /* length of name */
        char name[1];                   /* segment name - null terminated */
} ncache_t;
                                          See usr/src/uts/common/sys/dnlc.h
```

File systems must provide a structure for use only by the DNLC directory caching code for each directory.

```
typedef struct dcanchor {
        void *dca_dircache;     /* opaque directory cache handle */
        kmutex_t dca_lock;      /* protects the pointer and cache */
} dcanchor_t;
```

All file systems have an in-memory *xx*node (for example, inode in `ufs`) that could contain such a structure. Following is an example of how a file system would use the directory cache interfaces.

```
fs_lookup(dir, name)
{
        Return entry if in regular dnlc
        dcap = dir->dcap;
        switch dnlc_dir_lookup(dcap, name, &handle)
        case DFOUND:
                use handle to get and return vnode
                break
        case DNOENT:
                return ENOENT
        }
        caching = 0;
        if want to cache directory {
                switch dnlc_dir_start(dcap, num_dir_entries)
                case DNOMEM:
                case DTOOBIG:
                        mark directory as non cache-able
                        break;
                case
                        caching = 1;
        }
        while not end of directory {
                if entry && caching
                        handle = ino and offset;
                        dnlc_dir_add_entry(dcap, entry_name, handle)
                if free space && caching
                        handle = offset;
                        dnlc_dir_add_space(dcap, length. handle)
                if entry matches
                        get vnode
```

continues

```
                if various errors
                        if caching
                                dnlc_dir_purge(dcap)
                        return error

        }
        if caching
                dnlc_dir_complete(dcap)
        return vnode or ENOENT
}
```

The following set of new `dnlc` interfaces will be provided to cache complete directory contents (both entries and free space).

```
Status returns from the directory cache interfaces

#define DOK             0       /* operation successful */
#define DNOCACHE        1       /* there is no cache */
#define DFOUND          2       /* entry found */
#define DNOENT          3       /* no entry found */
#define DTOOBIG         4       /* exceeds tunable dnlc_dir_max_size */
#define DNOMEM          5       /* no memory */

Interfaces for building and adding to the directory cache

int dnlc_dir_start(dcanchor_t *dcap, uint_t num_entries);

Requests that a directory be cached. This must be called initially to enable caching on
a directory. After a successful call, directory entries and free space can be added (see
below) until the directory is marked complete. num_entries is an estimate of the current
number of directory entries. The request is rejected with DNOCACHE if num_entries falls
below the tunable dnlc_dir_min_size (see below), and rejected with DTOOBIG if it's above
dnlc_dir_max_size.

Returns DOK, DNOCACHE, DTOOBIG, DNOMEM (see below)

int dnlc_dir_add_entry(dcanchor_t *dcap, char *name, uint64_t handle);

Adds an entry (name and handle) into the partial or complete cache. Handle is a file-
system-specific quantity that is returned on calls to dnlc_dir_lookup() - see below.
Handle for ufs holds the inumber and a directory entry offset.

Returns DOK, DNOCACHE, DTOOBIG

int dnlc_dir_add_space(dcanchor_t *dcap, uint_t len, uint64_t handle);

Add free space (length and file-system-specific handle) into the partial or complete
cache. Handle for ufs holds the directory entry offset

Returns DOK, DNOCACHE, DTOOBIG

void dnlc_dir_complete(dcanchor_t *dcap);

Indicates the previously partial cache is now complete

void dnlc_dir_purge(dcanchor_t *dcap);

Deletes the partial or complete cache
```

continues

```
Interface for reading the directory cache

int dnlc_dir_lookup(dcanchor_t *dcap, char *name, uint64_t *handlep);

Looks up a file in the cache. Handlep must be non-null, and will be set to point to the
file-system-supplied handle

Returns DFOUND, DNOENT, DNOCACHE

Interfaces for amending the cache

int dnlc_dir_update(dcanchor_t *dcap, char *name, uint64_t handle);

Update the handle for the given entry

Returns DFOUND, DNOENT, DNOCACHE

int dnlc_dir_rem_entry(dcanchor_t *dcap, char *name, uint64_t *handlep);

Remove an entry

Returns the handle if handlep non-null and DFOUND, DNOENT, DNOCACHE

int dnlc_dir_rem_space_by_len(dcanchor_t *dcap, uint_t len, uint64_t *handlep);

Find and remove a space entry with at least the given length and
Returns the handle, and DFOUND, DNOENT, DNOCACHE

int dnlc_dir_rem_space_by_handle(dcanchor_t *dcap, uint64_t handle);

Find and removes the free space with the given handle

Returns DFOUND, DNOENT, DNOCACHE

Interfaces for initializing and finishing with the directory cache anchor

void dnlc_dir_init(dcanchor_t *dcap);

Initializes the anchor. This macro clears the dca_dircache field and does a mutex_init
on the lock

void dnlc_dir_fini(dcanchor_t *dcap);

Called to indicate the anchor is no longer used. This macro asserts there's no cache and
mutex_destroys the lock.
```

Additional notes on the directory cache interface are as follows:

- Because of memory shortages, directory caches can be purged at any time. If the last directory cache is purged because of a memory shortage, then the directory cache is marked internally as “no memory.” Future returns will all be DNOCACHE until the next `dnlc_start_dir()`, which will return DNOMEM once. This memory shortage may only be transient. It’s up to the file system to handle this condition, but an attempt to immediately rebuild the cache will very likely lead to the same shortage of memory and to thrashing.
- It’s file system policy as to when and what size directories to cache.

- Directory caches are purged according to LRU basis when a plea to release memory comes from the `kmem` system. A `kmem_cache` is used for one data structure, and on the reclaim callback, the LRU directory cache is released. Directory caches are also purged on failure to get additional memory. Otherwise, directories are cached as much as memory allows.

14.10.5 DNLC Housekeeping Thread

The DNLC maintains a task queue. The `dnlc_reduce_cache()` activates the task queue when there are `ncsize` name cache entries, and it reduces the size to `dnlc_nentries_low_water`, which is by default one hundredth less than (or 99% of) `ncsize`. If `dnlc_nentries` hits `dnlc_max_nentries` (twice `ncsize`), then this means that `dnlc_reduce_cache()` is failing to keep up. In this case, we refuse to add new entries to the `dnlc` until the task queue catches up.

14.10.6 DNLC Statistics

Below is an example of DNLC statistics obtained with the `kstat` command.

```
sol10$ kstat -n dnlcstats
module: unix                            instance: 0
name:   dnlcstats                       class:    misc
        crtime                          70.644144966
        dir_add_abort                   0
        dir_add_max                     0
        dir_add_no_memory               0
        dir_cached_current              0
        dir_cached_total                269
        dir_entries_cached_current      0
        dir_fini_purge                  0
        dir_hits                        131992
        dir_misses                      1312735
        dir_reclaim_any                 23
        dir_reclaim_last                4
        dir_remove_entry_fail           0
        dir_remove_space_fail           0
        dir_start_no_memory             0
        dir_update_fail                 0
        double_enters                   310146
        enters                          22732358
        hits                            384680010
        misses                          2390823
        negative_cache_hits             6048394
        pick_free                       0
        pick_heuristic                  15613169
        pick_last                       632544
        purge_all                       0
        purge_fs1                       0
        purge_total_entries             5369737
        purge_vfs                       27052
        purge_vp                        3009
        snaptime                        4408540.56846945
```

14.11 The File System Flush Daemon

The fsflush process writes modified pages to disk at regular intervals. The fsflush process scans through physical memory looking for dirty pages. When it finds one, it initiates a write (or putpage) operation on that page.

The fsflush process is launched by default every second and looks for pages that have been modified (the modified bit is set in the page structure) more than 30 seconds ago. If a page has been modified, then a page-out is scheduled for that page, but without the free flag so that the page remains in memory. The fsflush daemon flushes both data pages and inodes by default. Table 14.7 describes the parameters that affect the behavior of fsflush.

Table 14.7 Parameters That Affect fsflush

Parameter	Description	Min	Solaris 10 Default
tune_t_fsflushr	This specifies the number of seconds between fsflush scans.	1	1
autoup	Pages older than autoup in seconds are written to disk.	1	30
doiflush	By default, fsflush flushes both inode and data pages. Set to 0 to suppress inode updates.	0	1
dopageflush	This is set to 0 to suppress page flushes.	0	1

14.12 File System Conversion to Solaris 10

If you are porting a file system source to Solaris 10, you can follow these steps to convert an older file system to the new Solaris 10 APIs.

1. Vnodes must be separated from FS-specific nodes (for example, inodes). Previously, most file systems embedded the vnode in the FS-specific node. The node should now have a pointer to the vnode. vnodes are allocated by the file system with vn_alloc() and freed with vn_free(). If the file system recycles vnodes (by means of a node cache), then vnodes can be reinitialized with vn_reinit().

   Note: Make sure the VTO{node}() and {node}TOV() routines and the corresponding FS-node macros are updated.

2. Change all references to the "private" vnode fields to use accessors. The only "public" fields are listed below.

```
kmutex_t        v_lock;                 /* protects vnode fields */
uint_t          v_flag;                 /* vnode flags (see below) */
uint_t          v_count;                /* reference count */
caddr_t         v_data;                 /* private data for fs */
struct vfs      *v_vfsp;                /* ptr to containing VFS */
struct stdata   *v_stream;              /* associated stream */
enum vtype      v_type;                 /* vnode type */
dev_t           v_rdev;                 /* device (VCHR, VBLK) */
```

Otherwise, information about the vnode can be accessed, as shown below.

```
For:                    Use:

v_vfsmountedhere        vn_ismntpt() or vn_mountedvfs()
v_op                    vn_setops(), vn_getops(), vn_matchops(),
                        vn_matchopval()
v_pages                 vn_has_cached_data()
v_filocks               vn_has_flocks(), vn_has_mandatory_locks()
```

3. The only significant change to the vfs structure is that the vfs_op field should not be used directly. Any references or accesses to that field *must* go through one of the following: vfs_setops(),vfs_getops(), vfs_matchops(), vfs_can_sync().
4. Create an FS definition structure (vfsdef_t). This is similar to, but replaces, the vfssw table entry.
5. Create the operation definition tables for vnode and vfs operations.
6. Update (or create) the FS initialization routine (called at module-loader time) to create the vfsops and vnodeops structures. You do this by calling vn_make_ops() and either vfs_setfsops() (or vfs_makefsops()), using the "operations definition table" (created above).
7. Update the following vnode operation routines (if applicable):

   Add a pointer to the caller_context structure to the argument list for the following FS-specific routines: *xxx*_read(), *xxx*_write(), *xxx*_space(), *xxx*_setattr(), *xxx*_rwlock(), *xxx*_rwunlock().

   Add a pointer to the cred structure to the argument list for the following FS-specific routine: *xxx*_shrlock().

   Important note: Because the compilers don't yet support "designated initializers," the compiler cannot strongly type-check the file-system-specific vnode/vfs operations through the registration system. It's important that any changes to the argument list be done very carefully.

8. vnode life cycle: When a vnode is created (fully initialized, after locks are dropped but before anyone can get to it), call vn_exists(vnode *vp). This notifies anyone with registered interest on this file system that a new vnode has been created. If just the vnode is to be torn down (still fully functional, but before any locks are taken), call vn_invalid(vnode_t *vp) so that anyone with registered interest can be notified that this vnode is about to go away.

14.13 MDB Reference

Table 14.8 File System MDB Reference

dcmd or walker	**Description**
dcmd dnlc	Print DNLC contents
dcmd fsinfo	Print mounted filesystems
dcmd inode	Display summarized inode_t
dcmd inode_cache	Search/display inodes from inode cache
dcmd vnode2path	Vnode address to pathname
dcmd vnode2smap	Translate vnode to smap
dcmd whereopen	Given a vnode, dumps procs which have it open
walk dnlc_space_cache	Walk the dnlc_space_cache cache
walk vfs	Walk file system list
walk vn_cache	Walk the vn_cache cache