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.

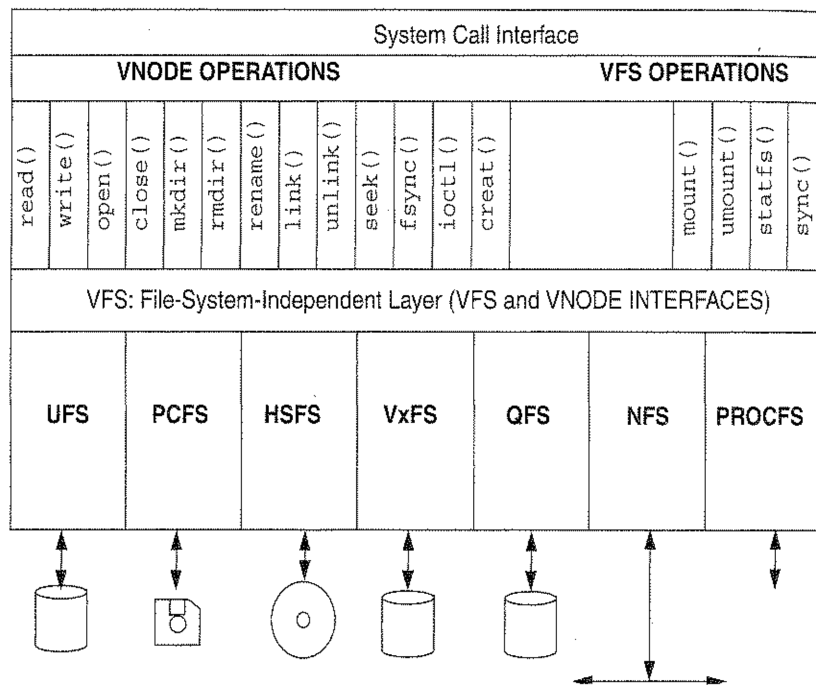

Figure 14.1 Solaris File System Framework

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.

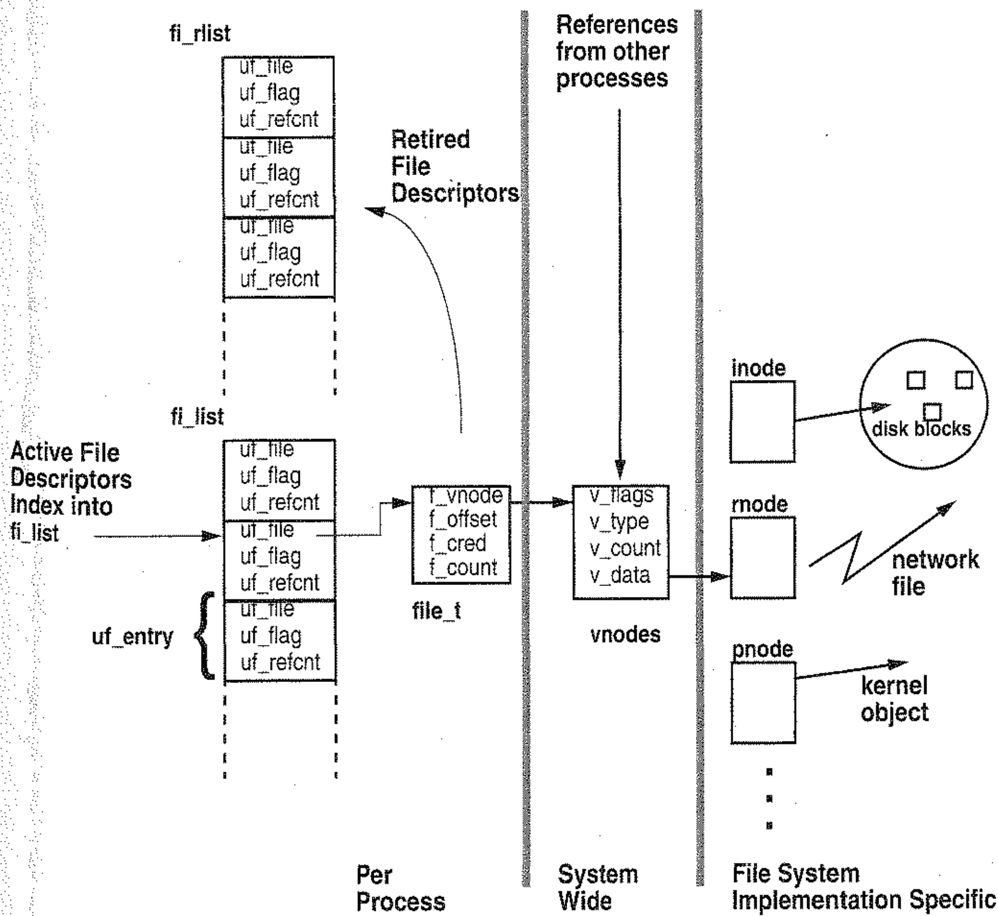

Figure 14.2 Structures Used for File Access

14.2.3 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).

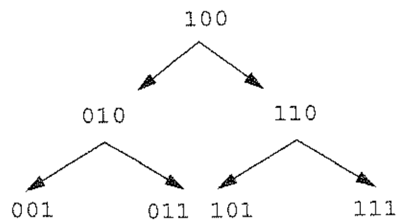

Figure 14.3 File Descriptor Integer Space as an *Infix* Binary Tree

- **(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:

$$N = xxxx10...0$$

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:

$$N - 1 = xxxx01...1:$$

$$N \,\&\, (N - 1) = xxxx000000$$
$$N \mid (N - 1) = xxxx111111$$
$$N \wedge (N - 1) = \quad 111111$$

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:

$$(C1)\ CSIZE(N) = (N - 1) \wedge (N \mid (N - 1))$$

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

$$(C1a)\ CSIZE(N) = N \wedge (N \mid (N + 1))$$
$$(L1a)\ LPARENT(N) = (N \,\&\, (N + 1)) - 1$$
$$(R1a)\ RPARENT(N) = N \mid (N + 1)$$

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 fds 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
```

14.5.6 `vfs` Information Available with `MDB`

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

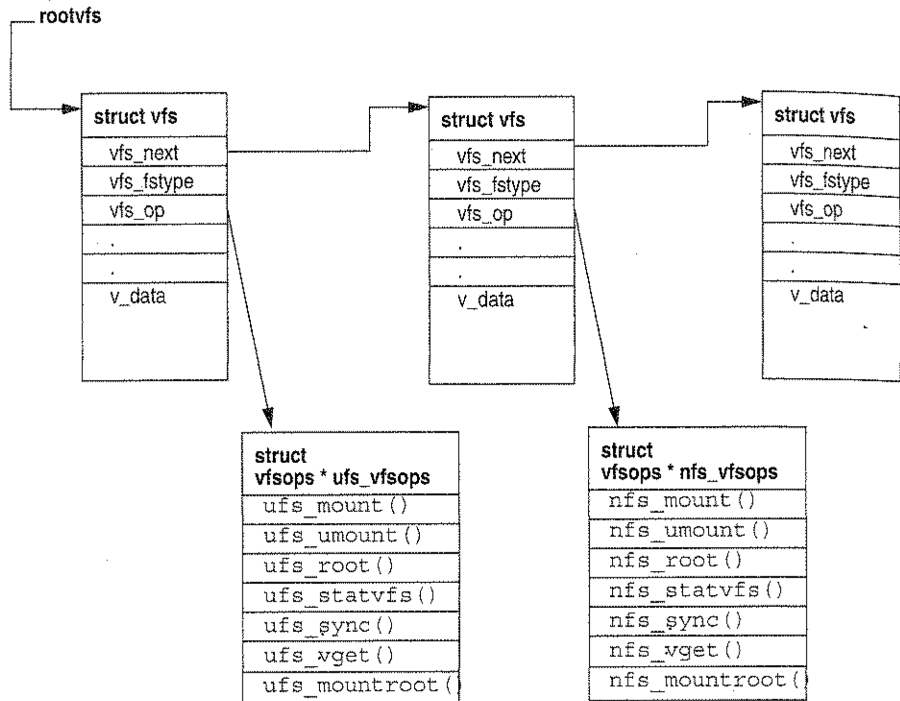

Figure 14.6 The Mounted `vfs` List

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
```

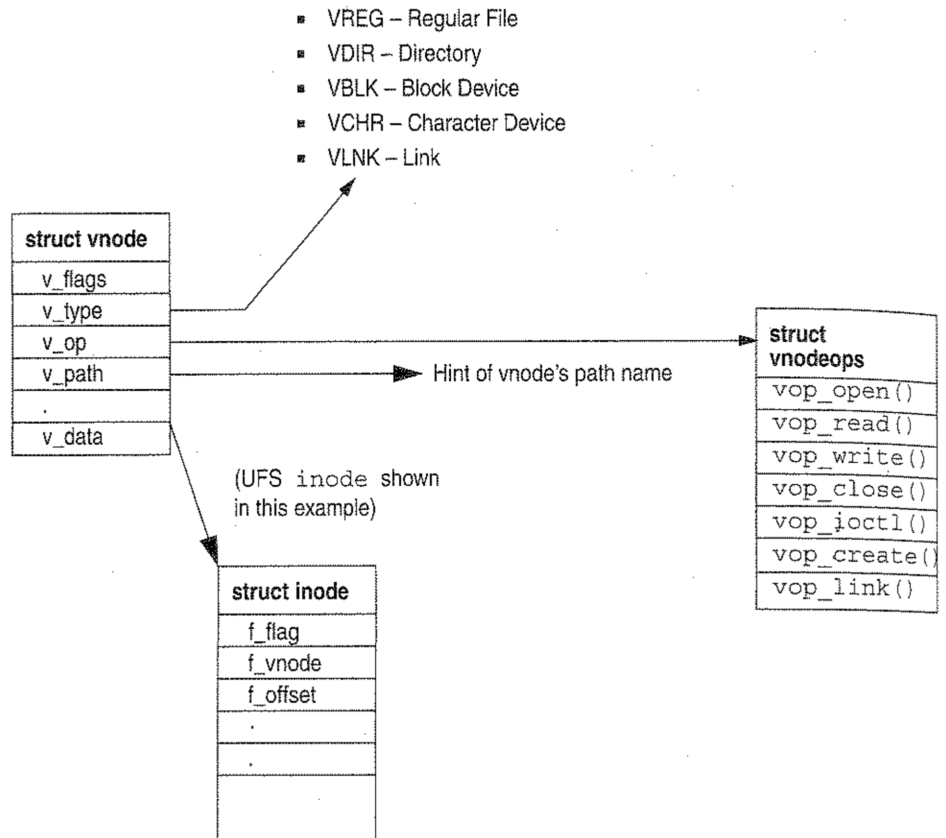

Figure 14.7 The vnode Object

- **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

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.

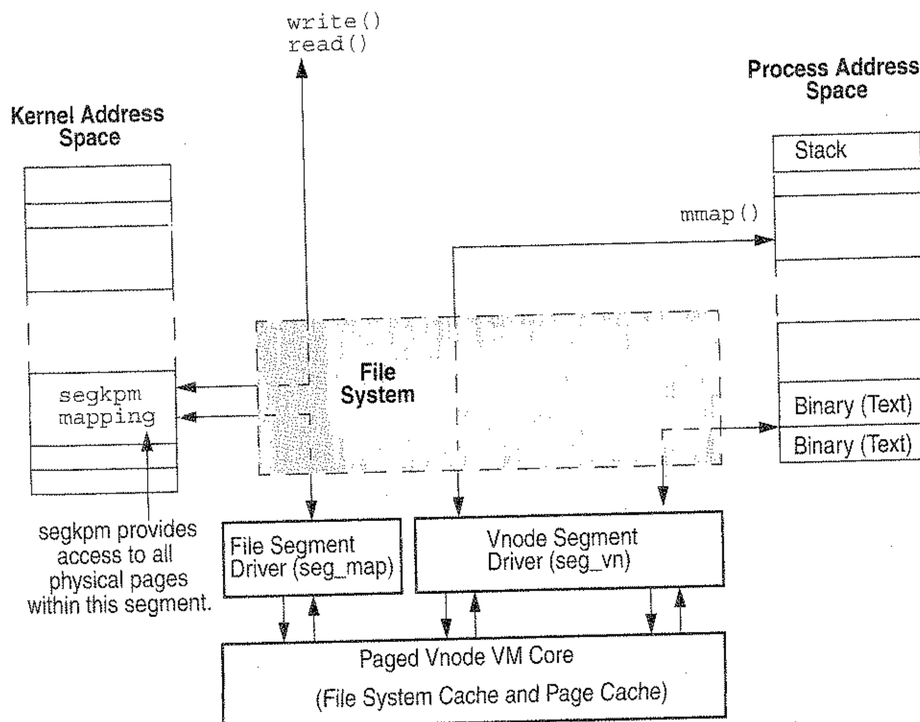

Figure 14.9 The `read()`/`write()` vs. `mmap()` Methods for File I/O

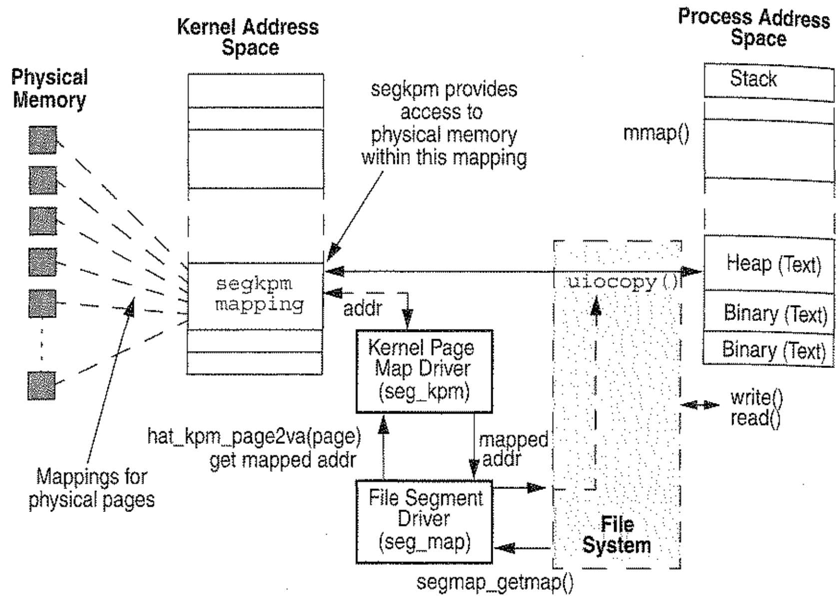

Figure 14.10 File System Data Movement with `seg_map/seg_kpm`

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 inter-

- **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.

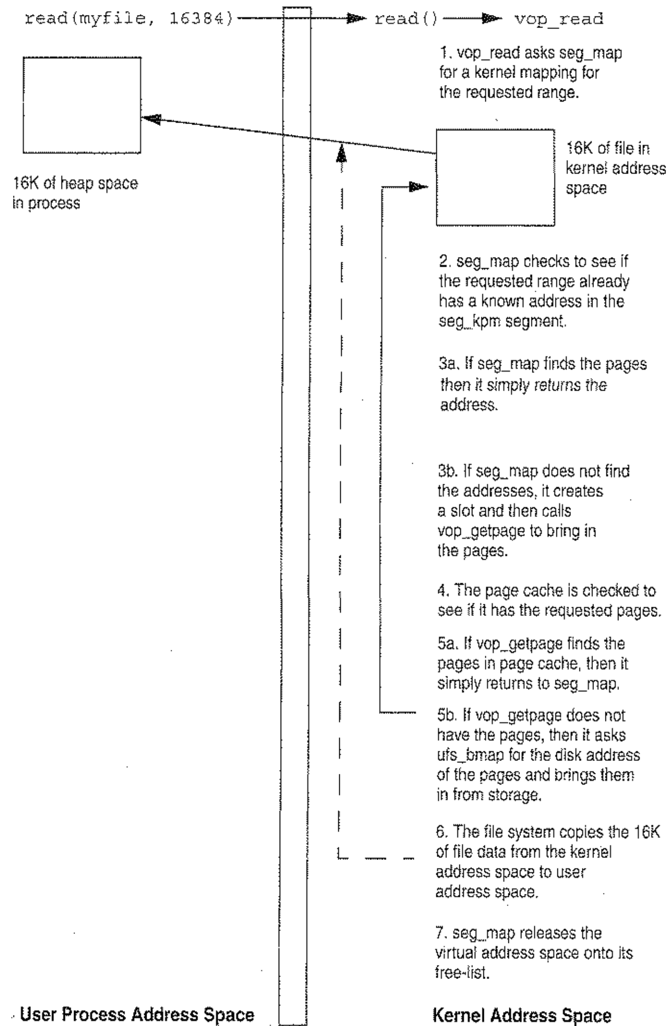

Figure 14.11 vop_read() segmap Interaction

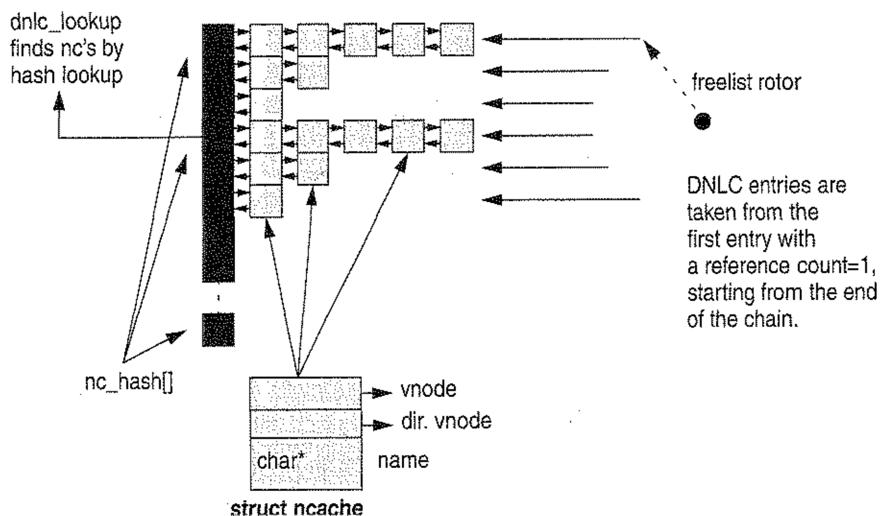

Figure 14.13 Solaris DNLC

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