around the segment drivers, so that subsystems need not know what segment driver is used for a memory range. The address space object shown in Figure 9.7 is linked from the process's address space and contains pointers to the segments that constitute the address space.

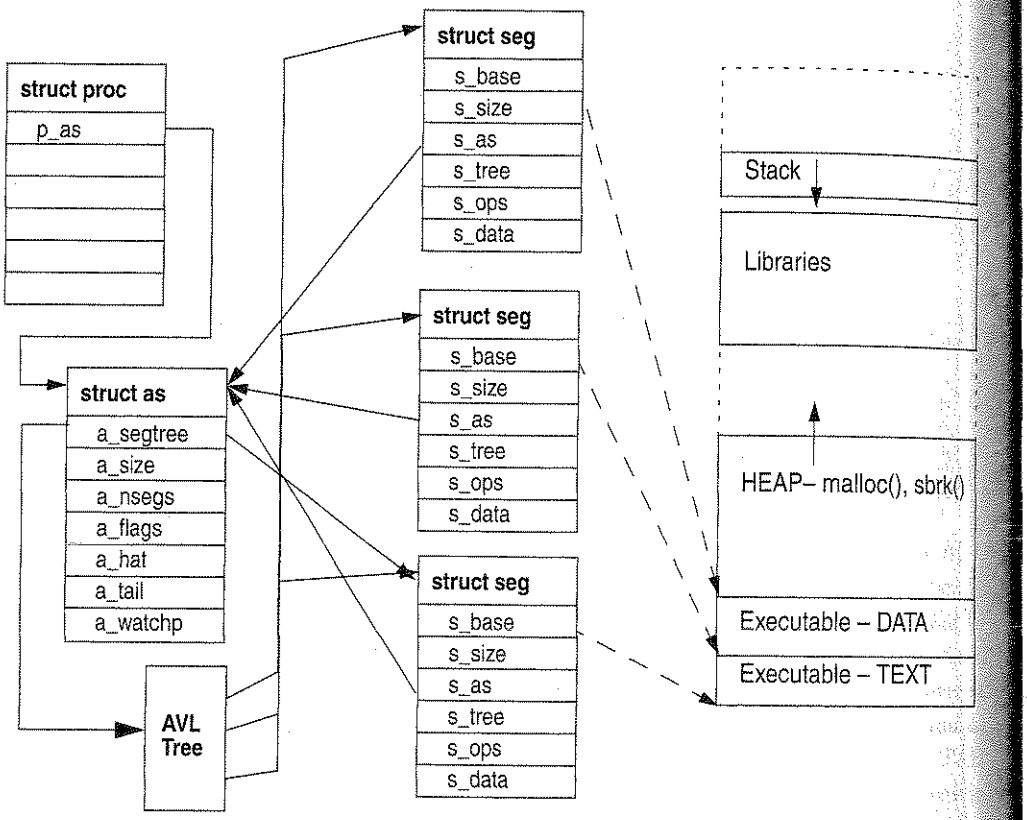

Figure 9.7 The Address Space

The address space subsystem manages the following functions:

- Duplication of address spaces, for `fork()`
- Destruction of address spaces, for `exit()`
- Creation of new segments within an address space
- Removal of segments from an address space
- Setting and management of page protection for an address space

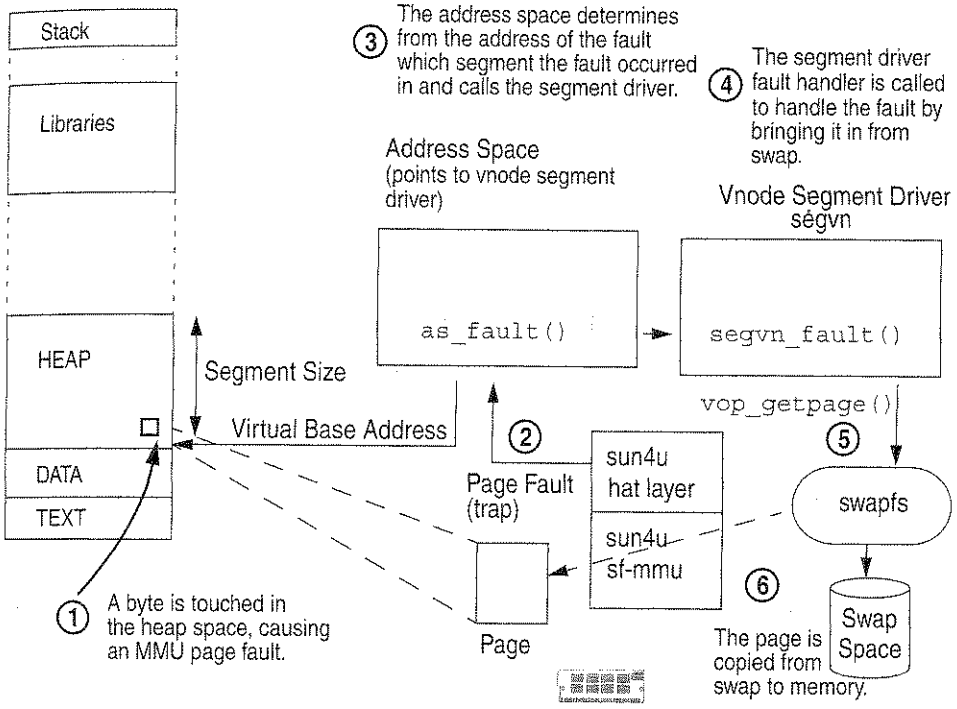

Figure 9.8 Virtual Address Space Page Fault Example

5. The segment driver then reads the page in from the backing store by calling the `getpage()` function of the backing store's `vnode`.
6. The backing store for this segment is the swap device, so the swap device `getpage()` function is called to read in the page from the swap device.

Once this process is completed, the process can continue execution. The following example using the DTrace `vm.d` script shows the logical flow for a zero-fill on demand page fault.

```
0  -> as_fault                                  4210
0   | as_fault:as_fault                         4210
0    -> as_segat                               4211
0    <- as_segat                               4212
0    -> segvn_fault                            4213
0      -> anonmap_alloc                        4216
0        -> anon_create                        4219
0        <- anon_create                        4224
0      <- anonmap_alloc                        4225
0      -> anon_array_enter                     4227
```

continues

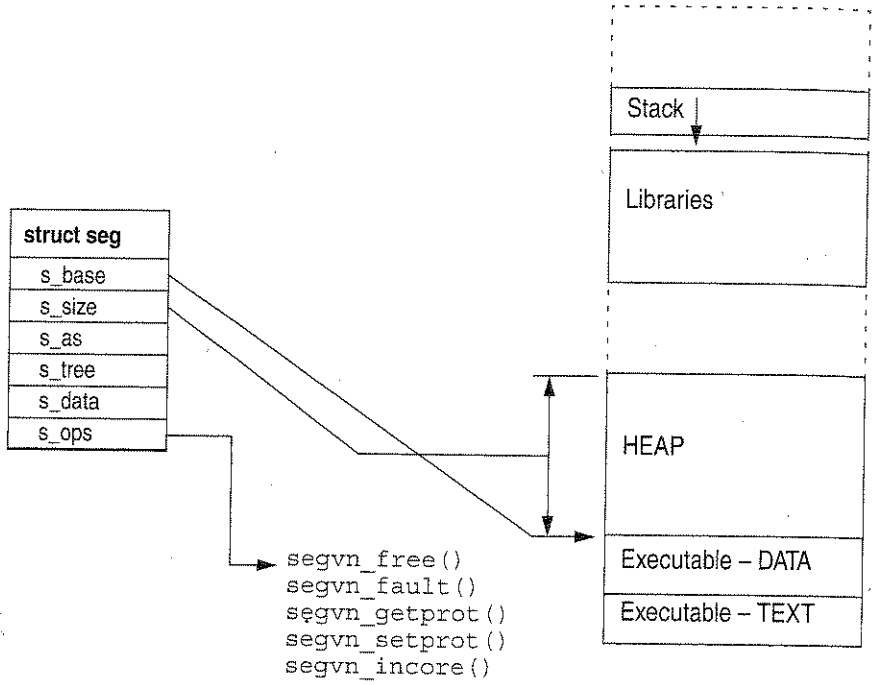

Figure 9.9 Segment Interface

For example, when a file is mapped into an address space with `mmap()`, the address space map routine `as_map()` is called with `segvn_create()` (the vnode segment driver constructor) as an argument, which in turn calls into the `seg_vn` segment driver to create the mapping. The segment object is created and inserted into the segment list for the address space (struct `as`), and from that point on, the address space can perform operations on the mapping without knowing what the underlying segment is.

The address space routines can operate on the segment without knowing what type of segment is underlying by calling the segment operation macros. For example, if the address space layer wants to call the fault handler for a segment, it calls `SEGOP_FAULT()`, which invokes the segment-specific page fault method, as shown below.

```
#define SEGOP_FAULT(h, s, a, l, t, rw) \
        (*(s)->s_ops->fault)((h), (s), (a), (l), (t), (rw))
```

See vm/seg.h

The Solaris kernel is implemented with a range of segment drivers for various functions. The different types of drivers are shown in Table 9.4. Most of the process address space mapping—including executable text, data, heap, stack and memory-mapped files—is performed with the `vnode` segment driver, `seg_vn`. Other types of mappings that don't have `vnodes` associated with them require different segment drivers. The other segment drivers are typically associated with kernel memory mappings or hardware devices, such as graphics adapters.

Table 9.5 describes segment driver methods implemented in Solaris 10.

Table 9.4 Solaris 10 Segment Drivers

Segment	Function
`seg_vn`	The `vnode` mappings into process address spaces are managed with the `seg_vn` device driver. Executable text and data, shared libraries, mapped files, heap and stack (heap and stack are anonymous memory) are all mapped with `seg_vn`.
`seg_kmem`	The segment from which the bulk of nonpageable kernel memory is allocated. (See Chapter 11.)
`seg_kp`	The segment from which pageable kernel memory is allocated. Only a very small amount of the kernel is pageable; kernel thread stacks and TNF buffers are the main consumers of pageable kernel memory.
`seg_kpm`	A mapping of all physical memory into the kernel's address space on 64-bit systems—to facilitate fast mapping of pages. The file systems use this facility to read and write to pages to avoid excessive map/unmap operations.
`seg_spt`	Shared page table segment driver. Fast System V shared memory is mapped into process address space from this segment driver. Memory allocated from this driver is also known as Intimate Shared Memory (ISM).
`seg_map`	The kernel uses the `seg_map` driver to map files (`vnodes`) into the kernel's address space, to implement file system caching.
`seg_dev`	Mapped hardware devices.
`seg_mapdev`	Mapping support for mapped hardware devices, through the `ddi_mapdev(9F)` interface.
`seg_lock`	Mapping support for hardware graphics devices that are mapped between user and kernel address space.
`seg_drv`	Mapping support for mapped hardware graphics devices.
`seg_nf`	Nonfaulting kernel memory driver.

Table 9.5 Solaris Segment Driver Methods

Method	Description
`advise()`	Provides a hint to optimize memory accesses to this segment. For example, sequential advice given to mapped files causes read-ahead to occur.
`checkprot()`	Checks that the requested access type (read, write, exec) is allowed within the protection level of the pages within the segment.
`dump()`	Dumps the segment to the dump device; used for crash dumps.
`dup()`	Duplicates the current memory segment, including all of the page mapping entries to the new segment pointer provided.
`fault()`	Handles a page fault for a segment. The arguments describe the segment, the virtual address of the page fault, and the type of fault.
`faulta()`	Starts a page fault on a segment and address asynchronously. Used for read-ahead or prefaulting of data as a performance optimization for I/O.
`free()`	Destroys a segment.
`getmemid()`	Gets a unique identifier for the memory segment.
`getoffset()`	Queries the segment driver for the offset into the underlying device for the mapping. (Not meaningful on all segment drivers.)
`getpolicy()`	Get the MPO Lgroup Policy for the supplied address
`getprot()`	Asks the segment driver for the protection levels for the memory range.
`gettype()`	Queries the driver for the sharing modes of the mapping.
`getvp()`	Gets the `vnode` pointer for the `vnode`, if there is one, behind this mapping.
`incore()`	Queries to find out how many pages are in physical memory for a segment.
`kluster()`	Asks the segment driver if it is OK to cluster I/O operations for pages within this segment.
`lockop()`	Locks or unlocks the pages for a range of memory mapped by a segment.
`pagelock()`	Locks a single page within the segment.
`setpagesize()`	Advises the page size for the address range

continues

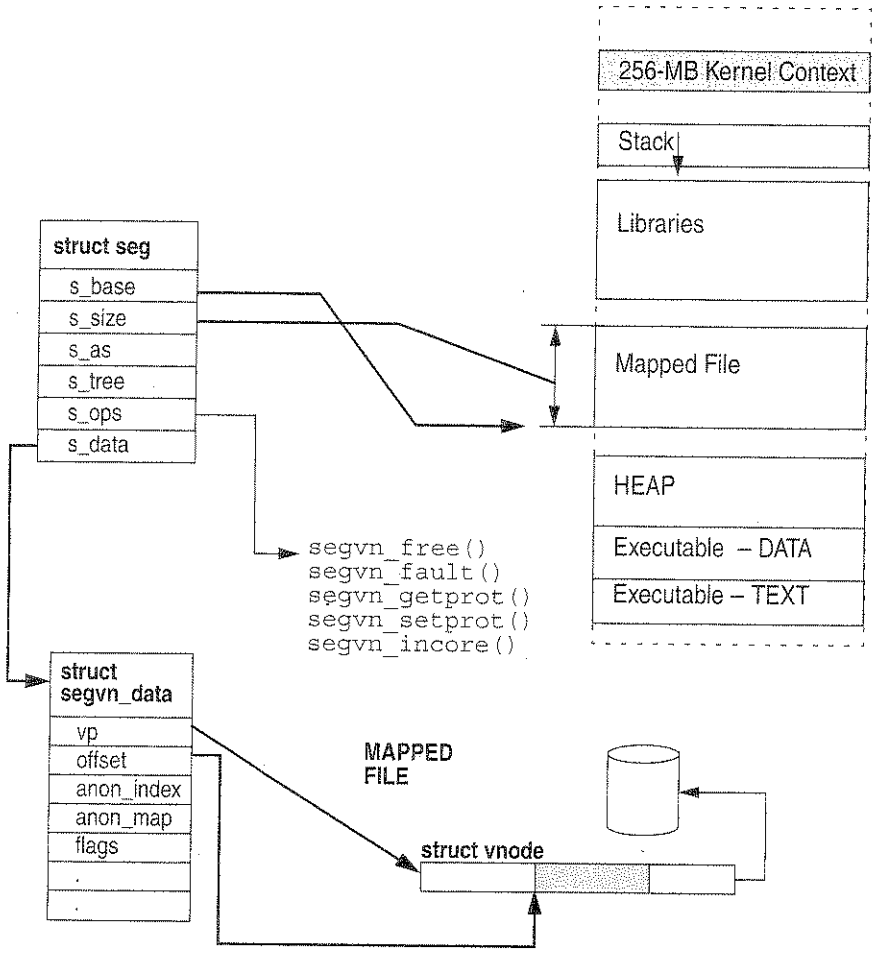

Figure 9.10 The `seg_vn` Segment Driver Vnode Relationship

Having established a valid hardware mapping for our file, we can look at how our mapped file is effectively read into the address space. The hardware MMU can generate traps for memory accesses to the memory within that segment. These traps will be routed to our `seg_vn` driver through the `as_fault()` routine. (See Section 9.4.4.) The first time we access a memory location within our segment, the `segvn_fault()` page fault handling routine is called. This fault handler recognizes our segment as a mapped file (by looking in the `segvn_data` structure) and simply calls into the vnode's file system (in this case, with `ufs_getpage()`) to read in a page-sized chunk from the file system. The subsequent access to memory that is now backed by physical memory simply results in a normal memory access.

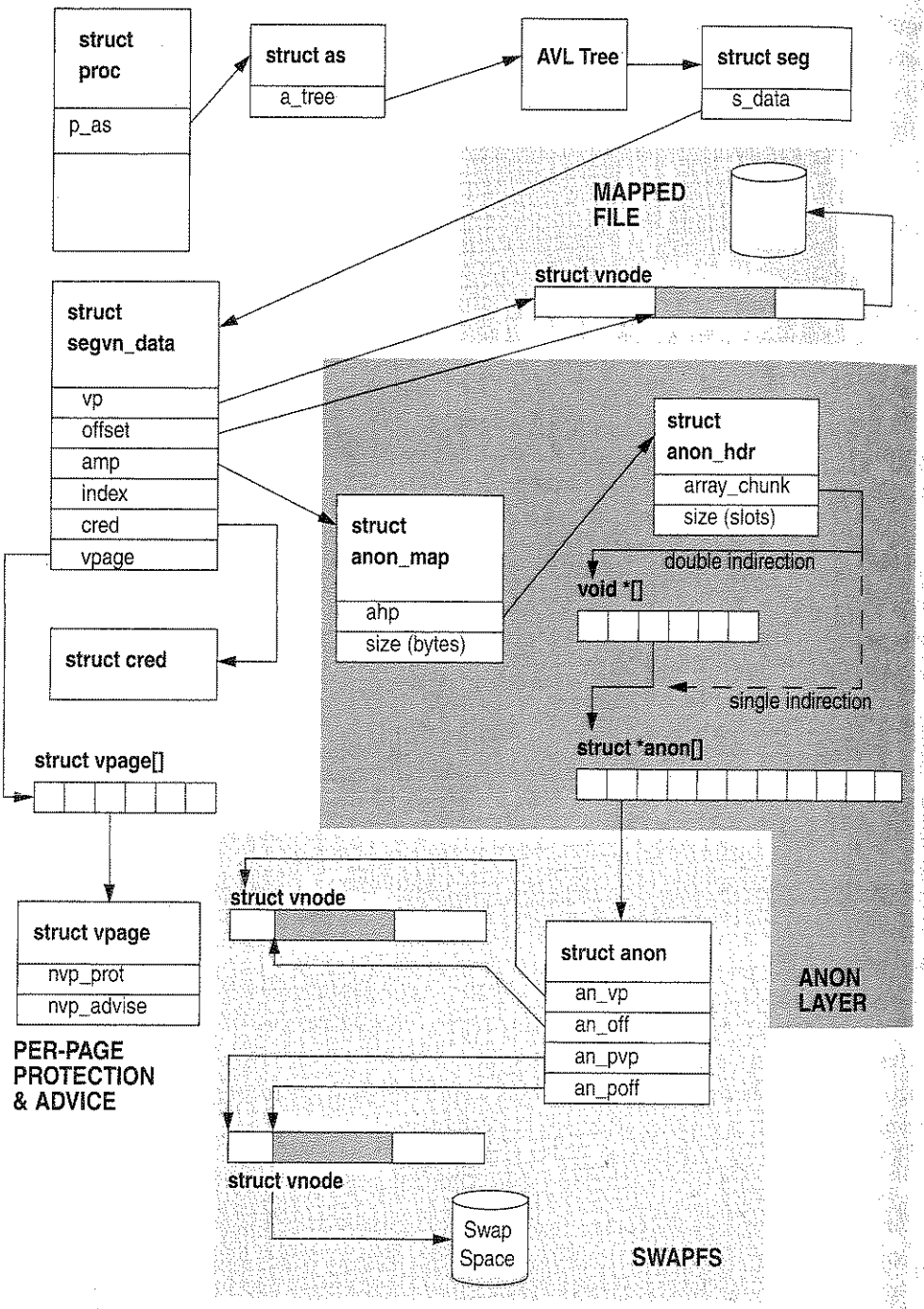

Figure 9.11 Anonymous Memory Data Structures

When we create a private segment, we reserve swap and allocate anon structures. At this stage, that's all that happens until a real memory page is created as a result of a ZFOD or copy-on-write (COW). When a physical page is faulted in, it is identified by vnode/offset, which for anonymous memory is the virtual swap device for the page.

Anonymous pages in Solaris are assigned a swapfs vnode and offsets when the segment driver calls anon_alloc() to get a new anonymous page. The anon_alloc() function calls into swapfs through swapfs_getvp() and then calls swapfs_getpage() to create a new page with swapfs vnode/offset. The anon structure members, an_vp and an_off, which identify the backing store for this page, are initialized to reference the vnode and offset within the swapfs virtual swap device.

Figure 9.12 shows how the anon slot points into swapfs. At this stage, we still don't need any physical swap space—the amount of virtual swap space available was decremented when the segment reserved virtual swap space—but because we haven't had to swap the pages out to physical swap, no physical swap space has been allocated.

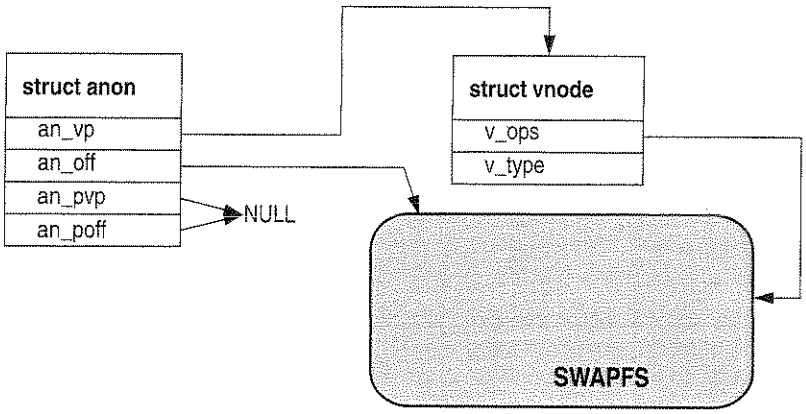

Figure 9.12 Anon Slot Initialized to Virtual Swap before Page-Out

It's not until the first page-out request occurs—because the page scanner must want to push a page to swap—that real swap is assigned. At this time, the page scanner looks up the vnode for the page and then calls its putpage() method. The page's vnode is a swapfs vnode, and hence swapfs_putpage() is called to swap this page out to the swap device. The swapfs_putpage() routine allocates a

page-sized block of physical swap and then sets the physical vnode an_pvp and an_poff fields in the anon slot to point to the physical swap device. The page is pushed to the swap device. At this point we allocate physical swap space. Figure 9.13 shows the anon slot *after* the page has been swapped out.

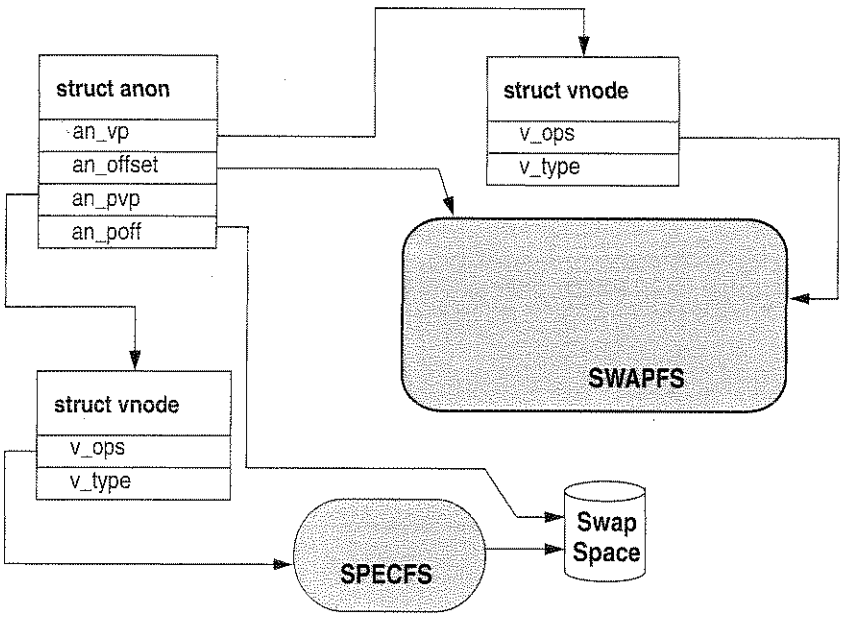

Figure 9.13 Physical Swap after a Page-Out Occurs

When we exhaust physical swap space, we simply ignore the putpage() request for a page, resulting in memory performance problems that are very hard to analyze. A failure does not occur when physical swap space fills; during reservation, we ensured that we had sufficient available virtual swap space, comprising both physical memory and physical swap space. In this case, the swapfs_putpage() simply leaves the page in memory and does not push a page to physical swap. This means that once physical swap is 100 percent allocated, we begin effectively locking down the remaining pages in physical memory. For this reason, it's often a bad idea to run with 100 percent physical swap allocation (swap -l shows 0 blocks free) because we might start locking down the wrong pages in memory and our working set might not correctly match the pages we really want in memory.

watchpoint, and the `si_code` field contains one of `TRAP_RWATCH`, `TRAP_WWATCH`, or `TRAP_XWATCH`, indicating read, write, or execute access, respectively. The `si_trapafter` field is zero unless `WA_TRAPAFTER` is in effect for this watched area; nonzero indicates that the current instruction is not the instruction that incurred the watchpoint trap. The `si_pc` field contains the virtual address of the instruction that incurred the trap. Figure 9.14 illustrates watchpoint data structures.

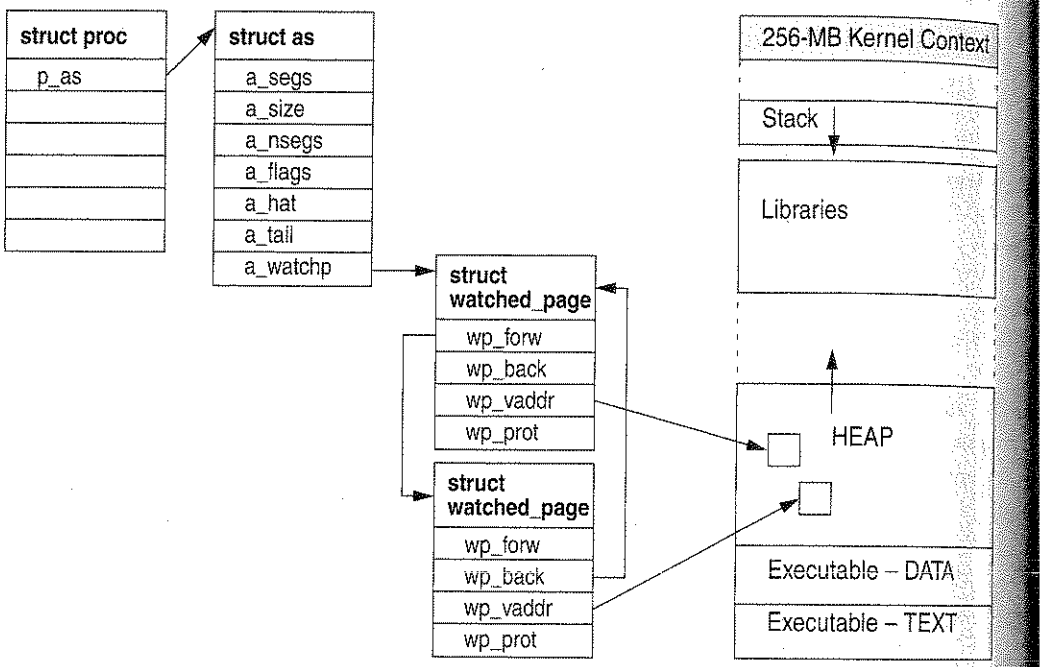

Figure 9.14 Watchpoint Data Structures

9.10 Changes to Support Large Pages

Solaris provides support for large MMU pages, as part of the Multiple Page Sizes for Solaris (MPSS) infrastructure. In this section, we discuss enhancements to page allocation and segment drivers.

9.10.1 System View of a Large Page

Solaris implements large pages in way that localizes changes to to only a few layer of the virtual memory system, and that does not slow down operations and applications not using large pages. The file systems and I/O layers need no knowledge of

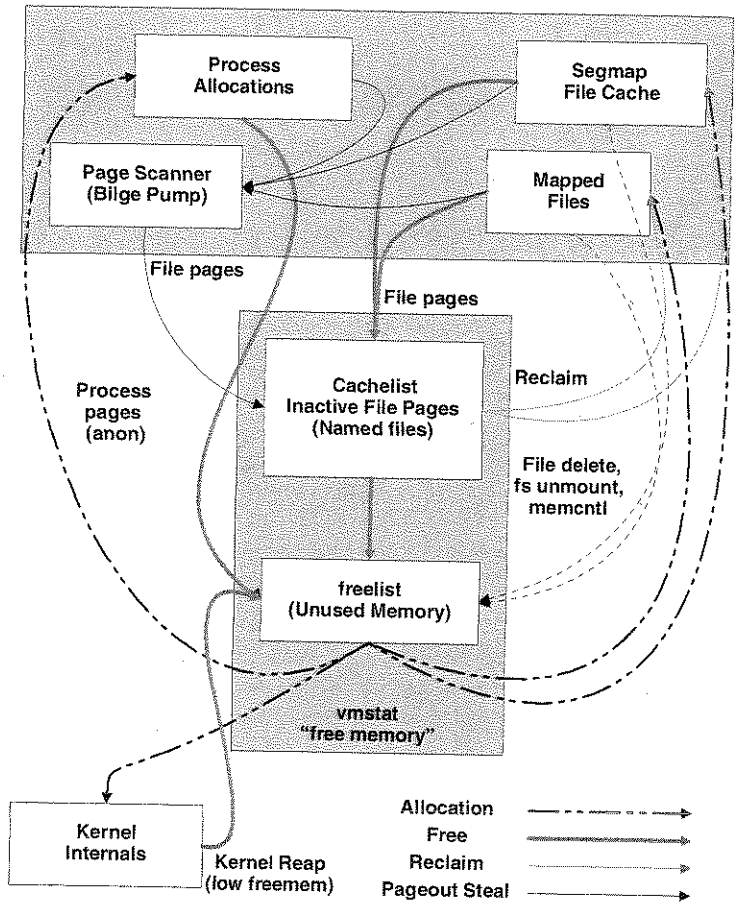

Figure 10.1 Life Cycle of Physical Memory

- **Anonymous/process allocations.** Anonymous memory, the most common form of allocation from the freelist, 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 freelist when it is unmapped or if it is stolen by the page scanner daemon.
- **File system "page cache."** The page cache is used for caching of file data for file systems other than the ZFS file system. The file system page cache grows on demand to consume available physical memory as a file cache and

10.2 Pages: The Basic Unit of Solaris Memory

Pages are the fundamental unit of physical memory in the Solaris memory management subsystem. In this section, we discuss how pages are structured, how they are located, and how free lists manage pools of pages within the system.

Physical memory is divided into pages. Every active (not free) page in the Solaris kernel is a mapping between a file (vnode) and memory; the page can be identified with a vnode pointer and the page size offset within that vnode. A page's identity is its vnode/offset pair. The vnode/offset pair is the backing store for the page and represents the file and offset that the page is mapping. The page structure and associated lists are shown in Figure 10.2.

The hardware address translation (HAT) and address space layers manage the mapping between a physical page and its virtual address space (which is described in Chapter 12). The key property of the vnode/offset pair is reusability; that is, we can reuse each physical page for another task by simply synchronizing its contents in RAM with its backing store (the vnode and offset) before the page is reused.

For example, we can reuse a page of heap memory from a process by simply copying the contents to its vnode and offset, which in this case will copy the contents to the swap device. The same mechanism is used for caching files, and we simply use the vnode/offset pair to reference the file that the page is caching. If we were to reuse a page of memory that was caching a regular file, then we simply synchronize the page with its backing store (if the page has been modified) or just reuse the page if it is not modified and does not need resyncing with its backing store.

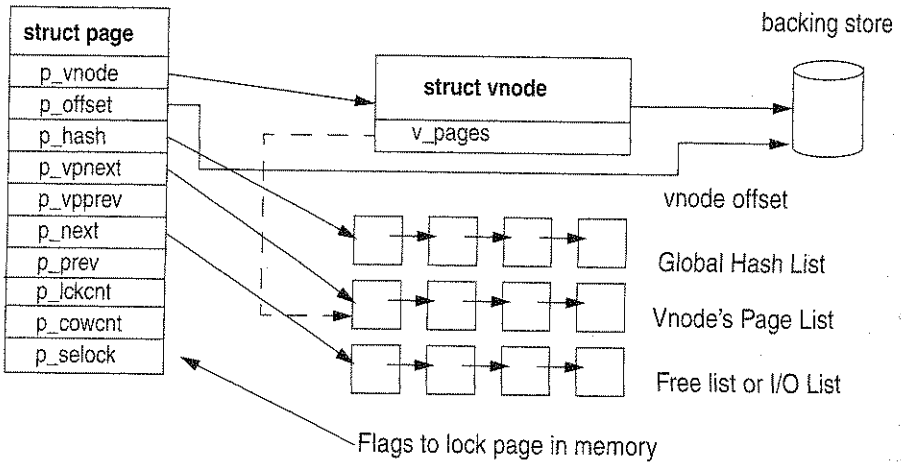

Figure 10.2 The Page Structure

10.2.1 The Page Hash List

The VM system hashes pages with identity (a valid vnode/offset pair) onto a global hash list so that they can be located by vnode and offset. Three page functions search the global page hash list: `page_find()`, `page_lookup()`, and `page_lookup_nowait()`. These functions take a vnode and offset as arguments and return a pointer to a page structure if found.

The global hash list is an array of pointers to linked lists of pages. The functions use a hash to index into the `page_hash` array to locate the list of pages that contains the page with the matching vnode/offset pair. Figure 10.3 shows how the `page_find()` function indexes into the `page_hash` array to locate a page matching a given vnode/offset.

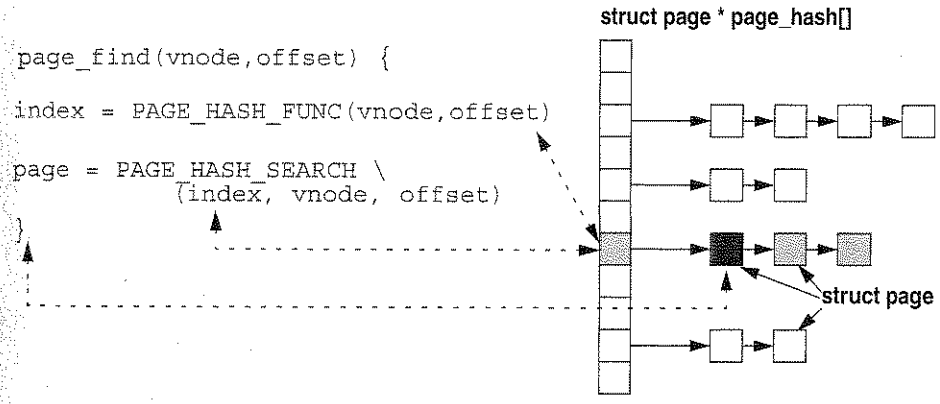

Figure 10.3 Locating Pages by Their Vnode/Offset Identity

`page_find()` locates a page as follows:

1. It calculates the slot in the `page_hash` array containing a list of potential pages by using the `PAGE_HASH_FUNC` macro, shown below.

```
#define PAGE_HASHSZ     page_hashsz
#define PAGE_HASHAVELEN         4
#define PAGE_HASH_FUNC(vp, off) \
        ((((uintptr_t)(off) >> PAGESHIFT) + \
                ((uintptr_t)(off) >> (PAGESHIFT + PH_SHIFT_SIZE)) + \
                ((uintptr_t)(vp) >> 3) + \
                ((uintptr_t)(vp) >> (3 + PH_SHIFT_SIZE)) + \
                ((uintptr_t)(vp) >> (3 + 2 * PH_SHIFT_SIZE))) & \
                (PAGE_HASHSZ - 1))
```

See vm/page.h

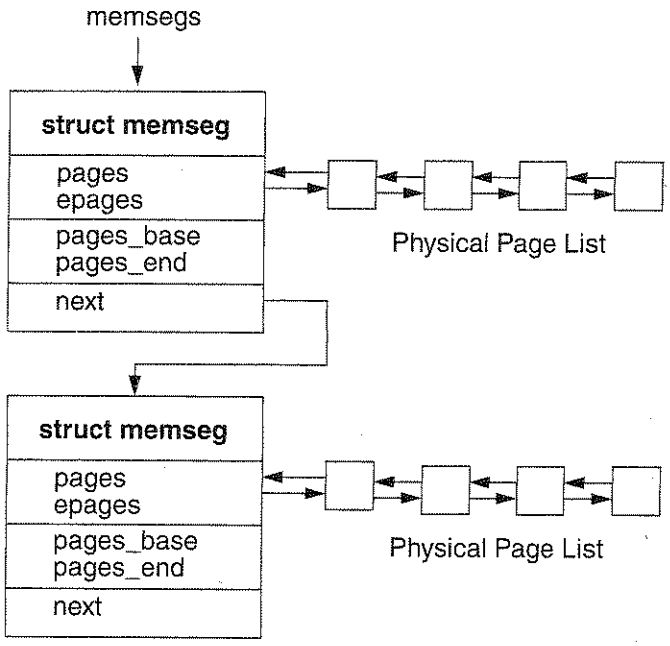

Figure 10.4 Contiguous Physical Memory Segments

added during system boot. They are also added and deleted dynamically when physical memory is added and removed while the system is running. Figure 10.4 shows the arrangement of the physical page lists into contiguous segments.

10.2.5 The Page-Level Interfaces

The Solaris virtual memory system implementation has grouped page management and manipulation into a central group of functions. These functions are used by the segment drivers and file systems to create, delete, and modify pages. The major page-level interfaces are shown in Table 10.1.

The `page_create_va()` function allocates pages. It takes the number of pages to allocate as an argument and returns a page list linked with the pages that have been taken from the free list. `page_create_va()` also takes a virtual address as an argument so that it can implement page coloring (discussed in Section 10.2.7). The new `page_create_va()` function subsumes the older `page_create()` function and should be used by all newly developed subsystems because `page_create()` may not correctly color the allocated pages.

11.1.6 The Kernel Address Space and Segments

The kernel address space is represented by the address space pointed to by the system object, kas. The segment drivers manage the manipulation of the segments within the kernel address space (see Figure 11.2).

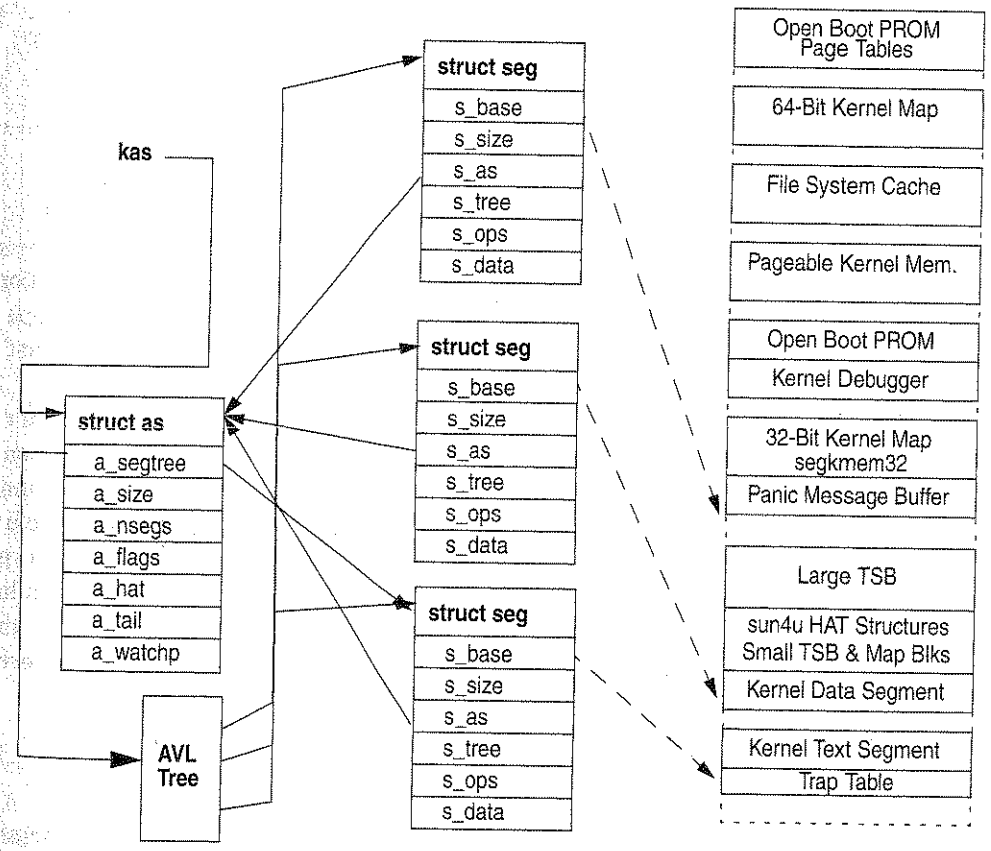

Figure 11.2 Kernel Address Space

The full list of segment drivers the kernel uses to create and manage kernel mappings is shown in Table 11.3. The majority of the kernel segments are manually calculated and placed for each platform, with the base address and offset hard-coded into a platform-specific header file. See Appendix A for a complete reference of platform-specific kernel allocation and address maps.

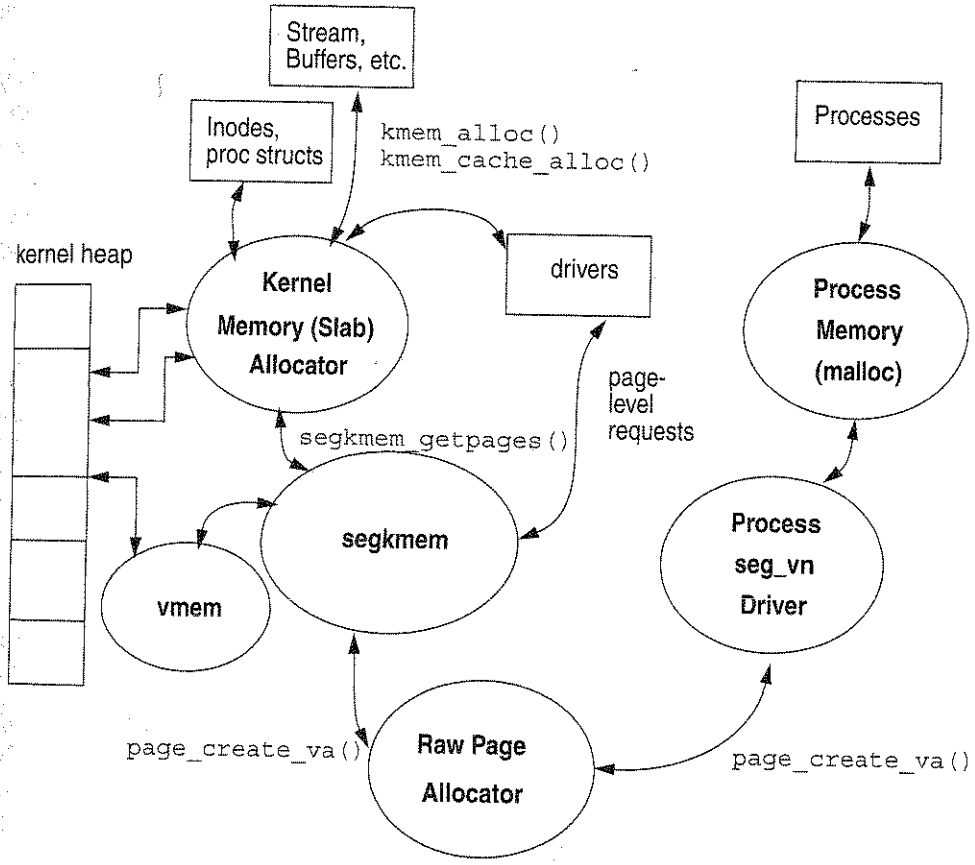

Figure 11.3 Different Levels of Memory Allocation

free and which parts are allocated so that we know where to satisfy new requests. To record the information, we use a general-purpose allocator to keep track of the start and length of the mappings that are allocated from the kernel map area. The allocator we use is the vmem allocator, which is used extensively for managing the kernel heap virtual address space, but since vmem is a universal resource allocator, it is also used for managing other resources (such as task, resource, and zone IDs).

We discuss the vmem allocator in detail in Section 11.3.

11.2.2 The Kernel Memory Segment Driver

The `segkmem` segment driver performs two major functions. It manages the creation of general-purpose memory segments in the kernel address space, and it also

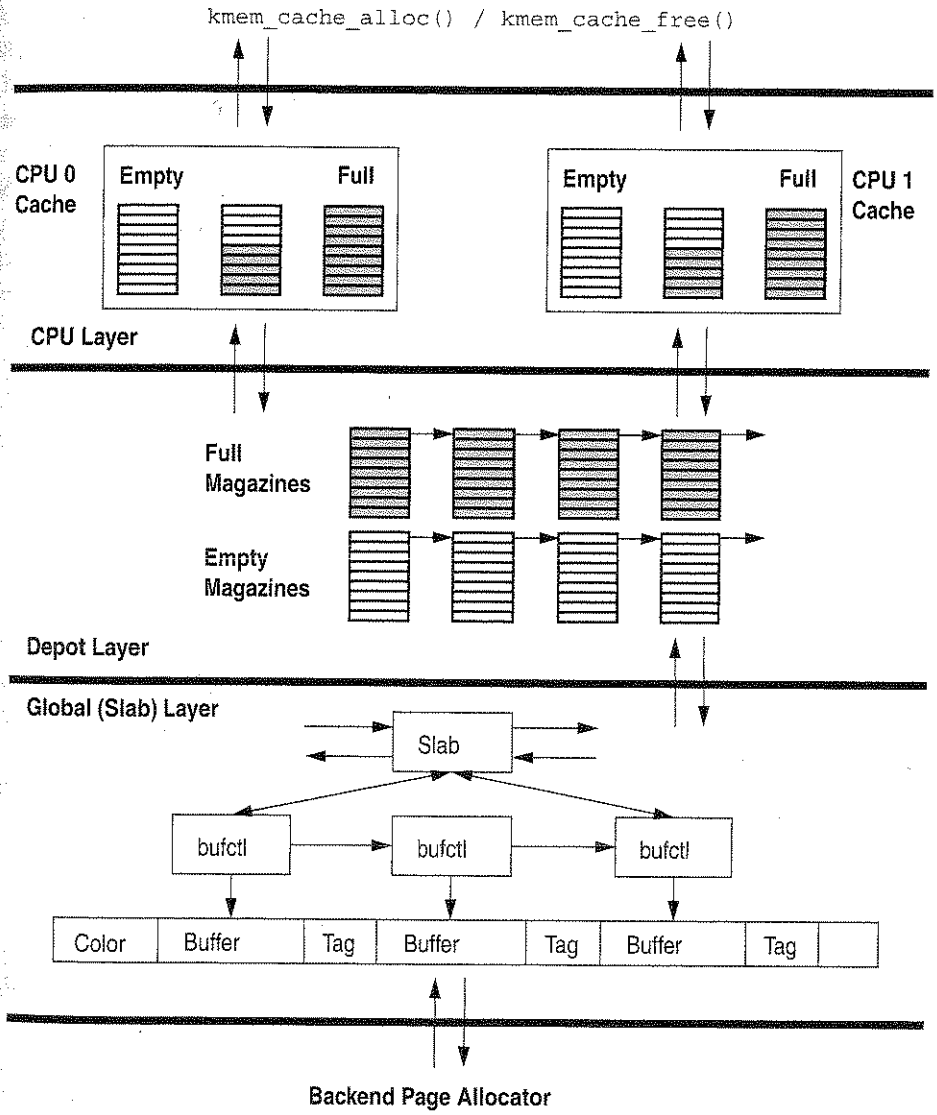

Figure 11.5 Slab Allocator Internal Implementation

11.2.3.5 The CPU Layer

The CPU layer caches groups of objects to minimize the number of times that an allocation will need to go down to the lower layers. This means that we can satisfy the majority of allocation requests without having to hold any global locks, thus dramatically improving the scalability of the allocator.

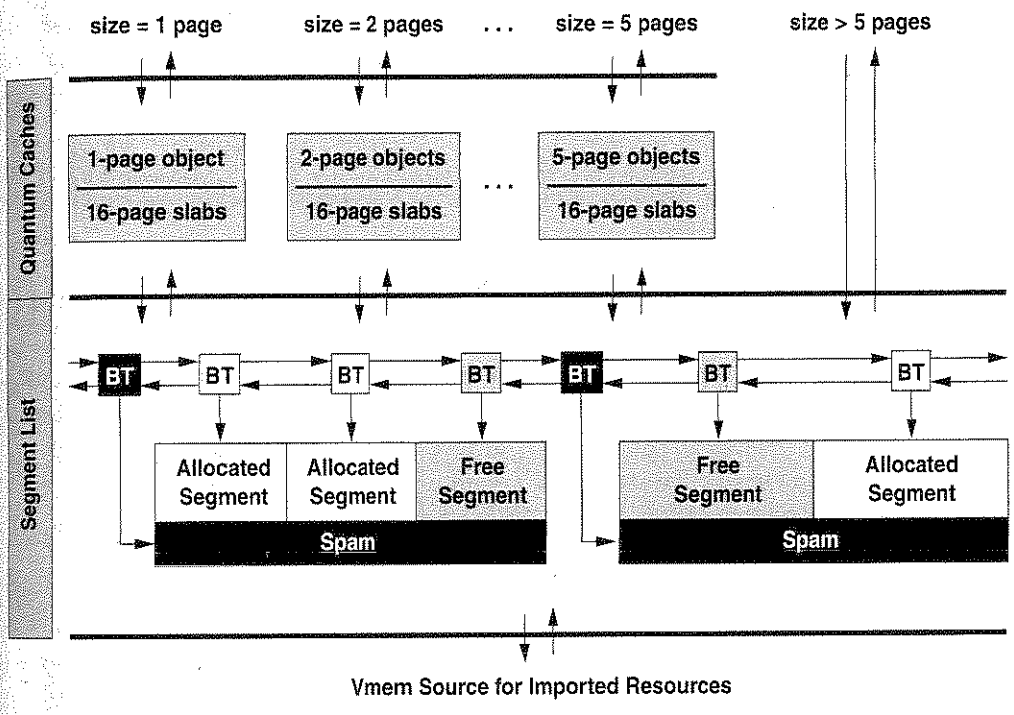

Figure 11.6 Structure of a Vmem Arena

Unfortunately, resource allocators can't use traditional boundary tags because the resource they're managing may not be memory (and therefore may not be able to hold information). In vmem we address this by using external boundary tags. For each segment in the arena we allocate a boundary tag to manage it, as shown in Figure 11.6. We'll see shortly that external boundary tags enable constant-time performance.

11.3.4.2 Allocating and Freeing Segments

Each arena has a segment list that links all of its segments in address order, as shown in Figure 11.6. Every segment also belongs to either a free list or an allocation hash chain, as described below. (The arena's segment list also includes span markers to keep track of span boundaries, so we can easily tell when an imported span can be returned to its source.)

We keep all free segments on power-of-two free lists; that is, free list[n] contains all free segments whose sizes are in the range [2n, 2n+1). To allocate a segment we search the appropriate free list for a segment large enough to satisfy the allocation. This approach, called segregated fit, actually approximates best-fit

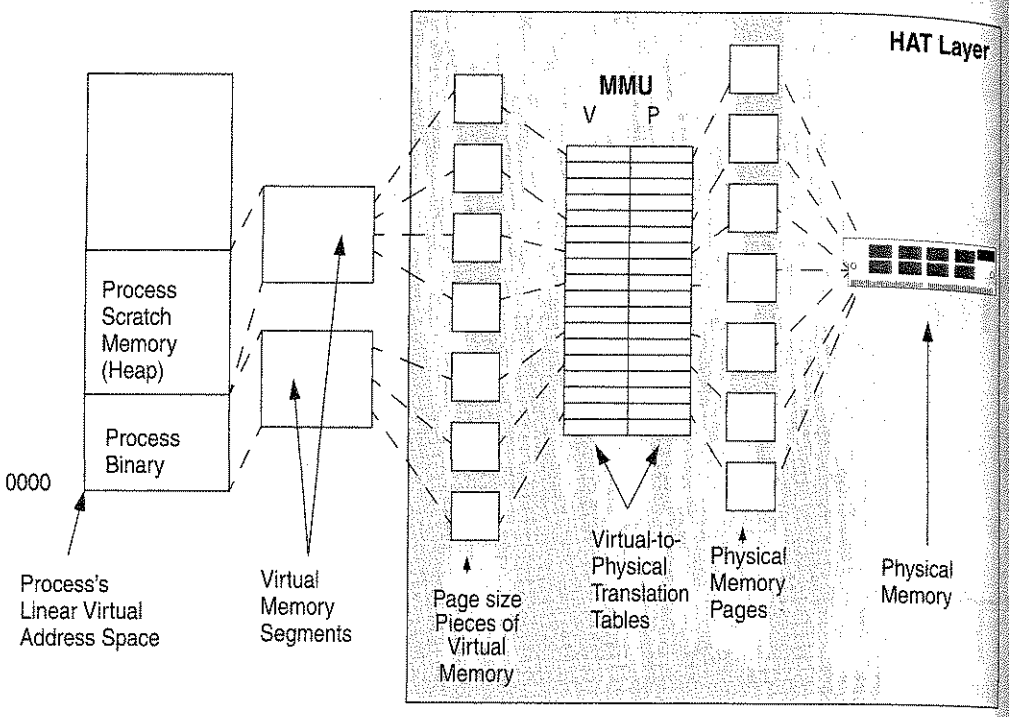

Figure 12.1 Role of the HAT Layer in Virtual-to-Physical Translation

and the action to be taken. We can call the HAT functions without knowing anything about the underlying MMU implementation; the arguments to the HAT functions are machine independent and usually consist of virtual addresses, lengths, page pointers, and protection modes.

Table 12.1 summarizes HAT functions.

Table 12.1 Machine-Independent HAT Functions

Function	Description
`hat_alloc()`	Allocates a HAT structure in the address space.
`hat_chgattr()`	Changes the protections for the supplied virtual address range.
`hat_clrattr()`	Clears the protections for the supplied virtual address range.
`hat_free_end()`	Informs the HAT layer that a process has exited.

continues

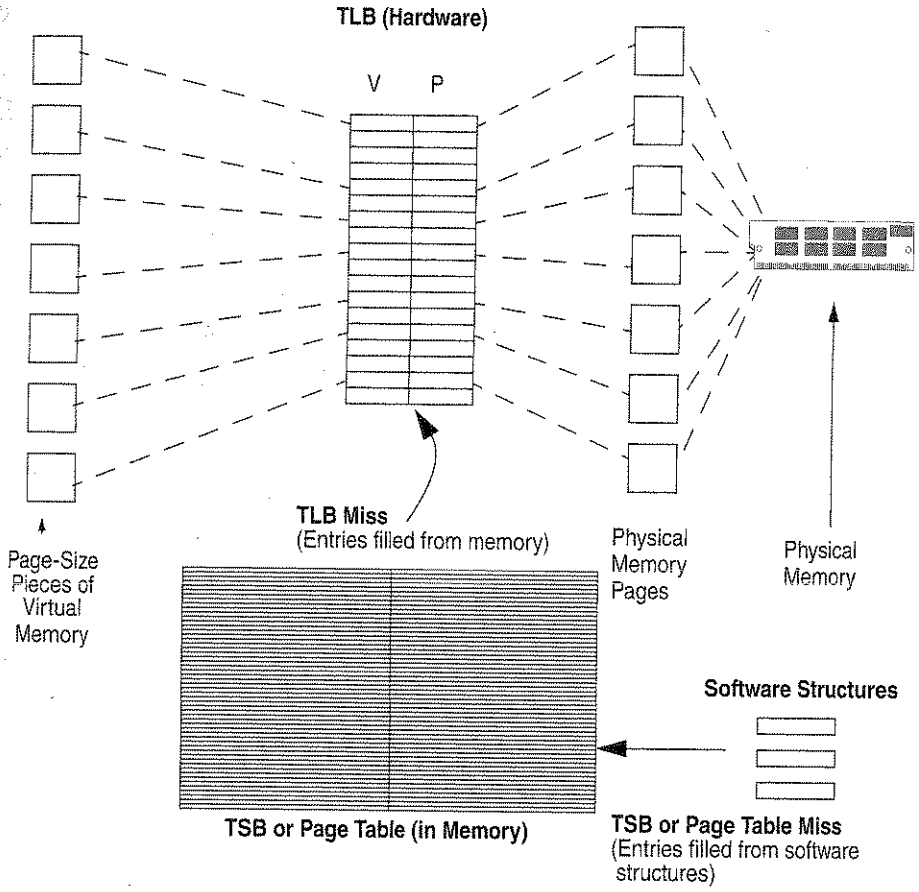

Figure 12.3 Virtual Address Translation Hardware and Software

ousted by other activity), the virtual-to-physical address is translated on-the-fly. If the TLB entry had been evicted, a TLB miss occurs, a hardware exception occurs, and the translation entry is looked up in the larger TSB.

The TSB is also limited in size, and in extreme circumstances a TSB miss can occur, requiring a lengthy search of the software structures linked to the process.

12.2.2 struct hat

The UltraSPARC `hat` structure is responsible for anchoring all HAT layer information and structures relating to a single process address space. These include the process' context ID (also known as the context number); a pointer to its `as` structure and TSBs; and various flags and status bits to name a few. Lets look at an