THE DESIGN AND IMPLEMENTATION OF THE FreeBSD® OPERATING SYSTEM

SECOND EDITION

MARSHALL KIRK McKUSICK
GEORGE V. NEVILLE-NEIL
ROBERT N.M. WATSON

The Design and Implementation of the

FreeBSD® Operating System

Second Edition

of the frequent locality of page faults. The drawback is that lookups often need to exclusively acquire the tree's lock to do the permutation causing lock contention between page-faulting threads sharing the same address space.

3. Having found a *vm_map_entry* that contains the faulting address, convert that address to an offset within the underlying *vm_object*. Calculate the offset within the *vm_object* as

```
object_offset = fault_address
    - vm_map_entry->start_address
    + vm_map_entry->object_offset
```

   Subtract off the start address to give the offset into the region mapped by the *vm_map_entry*. Add in the object_offset to give the absolute offset of the page within the *vm_object*.

4. Present the absolute object_offset to the underlying *vm_object*, which allocates a *vm_page* structure and uses its pager to fill the page. The *vm_object* then returns a pointer to the *vm_page* structure, which is mapped into the faulting location in the process address space.

Once the appropriate page has been mapped into the faulting location, the page-fault handler returns and reexecutes the faulting instruction.

Mapping to *Vm_objects*

A *vm_object* holds information about either a file or an area of anonymous memory. Whether a file is mapped by a single process in the system or by many processes in the system, it will always be represented by a single *vm_object*. Thus, the *vm_object* is responsible for maintaining all the state about those pages of a file that are resident. All references to that file will be described by *vm_map_entry* structures that reference the same *vm_object*. A *vm_object* never stores the same page of a file in more than one physical-memory page, so all mappings will get a consistent view of the file.

A *vm_object* stores the following information:

- A collection of the pages for that *vm_object* that are currently resident in main memory; a page may be mapped into multiple address spaces, but it is always claimed by exactly one *vm_object*
- A count of the number of *vm_map_entry* structures or other *vm_object*s that reference the *vm_object*
- The size of the file or anonymous area described by the *vm_object*
- The number of memory-resident pages held by the *vm_object*
- For shadow objects, a pointer to the next *vm_object* in the chain (shadow objects are described in Section 6.5)

- The type of ***pager*** for the *vm_object*; the pager is responsible for providing the data to fill a page and for providing a place to store the page when it has been modified (pagers are covered in Section 6.10)

There are three types of *vm_object*s in the system:

- *Named vm_object*s represent files; they may also represent hardware devices that are able to provide mapped memory such as frame buffers.
- *Anonymous vm_object*s represent areas of memory that are zero filled on first use; they are abandoned when they are no longer needed.
- *Shadow vm_object*s hold private copies of pages that have been modified; they are abandoned when they are no longer referenced.

Shadow and all anonymous *vm_object*s (other than POSIX *shmem*) are often referred to as "internal" *vm_object*s in the source code. The type of a *vm_object* is defined by the type of pager that it uses to fulfill page-fault requests.

A named *vm_object* uses the device pager if it maps a hardware device, the vnode pager if it is backed by a file in the filesystem, or the swap pager if it backs a POSIX *shmem* object. The device pager services a page fault by returning the appropriate physical address for the device being mapped. Since the device memory is separate from the main memory on the machine, it will never be selected by the pageout daemon. Thus, the device pager never has to handle a pageout request.

The vnode pager provides an interface to *vm_object*s that represent files in the filesystem. The vnode pager keeps a reference to a vnode that represents the file being mapped in the *vm_object*. The vnode pager services a pagein request by doing a read on the vnode; it services a pageout request by doing a write to the vnode. Thus, the file itself stores the modified pages. In cases where it is not appropriate to modify the file directly, such as an executable that does not want to modify its initialized data pages, the kernel must interpose a shadow *vm_object* between the *vm_map_entry* and the *vm_object* representing the file; see Section 6.5.

Anonymous or POSIX *shmem vm_object*s use the swap pager. An anonymous or POSIX *shmem vm_object* services pagein requests by getting a page of memory from the free list and zeroing that page. When a pageout request is made for a page for the first time, the swap pager is responsible for finding an unused page in the swap area, writing the contents of the page to that space, and recording where that page is stored. If a pagein request comes for a page that had been previously paged out, the swap pager is responsible for finding where it stored that page and reading back the contents into a free page in memory. A later pageout request for that page will cause the page to be written out to the previously allocated location.

Shadow *vm_object*s also use the swap pager. They work just like anonymous or POSIX *shmem vm_object*s, except that the swap pager does not need to provide

their initial pages. The initial pages are created by the *vm_fault*() routine by copying existing pages in response to copy-on-write faults.

Further details on the pagers are given in Section 6.10.

Vm_objects

Each virtual-memory *vm_object* has a pager type, pager handle, and pager private data associated with it. The *vm_object*s that map files have a vnode-pager type associated with them. The handle for the vnode-pager type is a pointer to the vnode on which to do the I/O, and the private data is the size of the vnode at the time that the mapping was done. Every vnode that maps a file has a *vm_object* associated with it. When a fault occurs for a file that is mapped into memory, the *vm_object* associated with the file can be checked to see whether the faulted page is resident. If the page is resident, it can be used. If the page is not resident, a new page is allocated, and the vnode pager is requested to fill the new page.

Caching in the virtual-memory system is done by a *vm_object* that is associated with a file or region that it represents. Each *vm_object* contains pages that are the cached contents of its associated file or region. All *vm_object*s are reclaimed as soon as their reference count drops to zero. Pages associated with reclaimed *vm_object*s are moved to the free list. Each *vm_object* that represents anonymous memory is reclaimed as part of cleaning up a process as it exits. However, *vm_object*s that refer to files are persistent. When the reference count on a vnode drops to zero, it is stored on a ***least recently used*** (***LRU***) list known as the vnode cache; vnodes are described in Section 7.3. The vnode does not release its *vm_object* until the vnode is reclaimed and reused for another file. Unless there is pressure on the memory, the *vm_object* associated with the vnode will retain its pages. If the vnode is reactivated and a page fault occurs before the associated page is freed, that page can be used rather than being reread from disk.

This cache is similar to the text cache found in earlier versions of BSD in that it provides performance improvements for short-running but frequently executed programs. Frequently executed programs include those used to list the contents of directories, show system status, or perform the intermediate steps involved in compiling a program. For example, consider a typical application that is made up of multiple source files. Each of several compiler steps must be run on each file in turn. The first time that the compiler is run, the executable files associated with its various components are read in from the disk. For each file compiled thereafter, the previously created executable files are found, as well as any previously read header files, alleviating the need to reload them from disk each time.

Vm_objects to Pages

When the system is first booted, the kernel looks through the physical memory on the machine to find out how many pages are available. After the physical memory that will be dedicated to the kernel has been deducted, all the remaining pages of physical memory are described by *vm_page* structures. These *vm_page* structures

are all initially placed on the memory free list. As the system starts running and processes begin to execute, they generate page faults. Each page fault is matched to the *vm_object* that covers the faulting piece of address space. The first time that a piece of a *vm_object* is faulted, it must allocate a page from the free list and must initialize that page either by zero-filling it or by reading its contents from the filesystem. That page then becomes associated with the *vm_object*. Thus, each *vm_object* has its current set of *vm_page* structures linked to it.

If memory becomes scarce, the paging daemon will search for pages that have not been used actively. Before these pages can be used by a new *vm_object*, they must be removed from all the processes that currently have them mapped, and any modified contents must be saved by the *vm_object* that owns them. Once cleaned, the pages can be removed from the *vm_object* that owns them and can be placed on the free list for reuse. The details of the paging system are described in Section 6.12.

6.5 Shared Memory

In Sections 6.2 and 6.4, we explained how the address space of a process is organized. This section shows the additional data structures needed to support shared address space between processes. Traditionally, the address space of each process was completely isolated from the address space of all other processes running on the system. The only exception was read-only sharing of program text. All interprocess communication was done through well-defined channels that passed through the kernel: pipes, sockets, files, and special devices. The benefit of this isolated approach is that, no matter how badly a process destroys its own address space, it cannot affect the address space of any other process running on the system. Each process can precisely control when data are sent or received; it can also precisely identify the locations within its address space that are read or written. The drawback of this approach is that all interprocess communication requires at least two system calls: one from the sending process and one from the receiving process. For high volumes of interprocess communication, especially when small packets of data are being exchanged, the overhead of the system calls dominates the communications cost.

Shared memory provides a way to reduce interprocess-communication costs dramatically. Two or more processes that wish to communicate map the same piece of read–write memory into their address space. Once all the processes have mapped the memory into their address space, any changes to that piece of memory are visible to all the other processes, without any intervention by the kernel. Thus, interprocess communication can be achieved without any system-call overhead other than the cost of the initial mapping. The drawback to this approach is that, if a process that has the memory mapped corrupts the data structures in that memory, all the other processes mapping that memory also see the corrupted data structures. In addition, there is the complexity faced by the application developer who must develop data structures to control access to the shared memory and must

cope with the race conditions inherent in manipulating and controlling such data structures that are being accessed concurrently.

Some UNIX variants have a kernel-based semaphore mechanism to provide the needed serialization of access to the shared memory. However, both getting and setting such semaphores require system calls. The overhead of using such semaphores is comparable to that of using the traditional interprocess-communication methods. Unfortunately, these semaphores have all the complexity of shared memory, yet confer little of its speed advantage. The primary reason to introduce the complexity of shared memory is for the commensurate speed gain. If this gain is to be obtained, most of the data-structure locking needs to be done in the shared memory segment itself. The kernel-based semaphores should be used for only those rare cases where there is contention for a lock and one process must wait. Consequently, modern interfaces, such as POSIX Pthreads, are designed such that the semaphores can be located in the shared memory region. The common case of setting or clearing an uncontested semaphore can be done by the user process, without calling the kernel. There are two cases where a process must perform a system call. If a process tries to set an already-locked semaphore, it must call the kernel to block until the semaphore is available. This system call has little effect on performance because the lock is contested, so it is impossible to proceed, and the kernel must be invoked to do a context switch anyway. If a process clears a semaphore that is wanted by another process, it must call the kernel to awaken that process. Since most locks are uncontested, the applications can run at full speed without kernel intervention.

Mmap Model

When two processes wish to create an area of shared memory, they must have some way to name the piece of memory that they wish to share, and they must be able to describe its size and initial contents. The system interface describing an area of shared memory accomplishes all these goals by using files as the basis for describing a shared memory segment. A process creates a shared memory segment by using

```
void *addr = mmap(
    void *addr,      /* base address */
    size_t len,      /* length of region */
    int prot,        /* protection of region */
    int flags,       /* mapping flags */
    int fd,          /* file to map */
    off_t offset);   /* offset to begin mapping */
```

to map the file referenced by descriptor *fd*, starting at file offset *offset* into its address space, starting at *addr* and continuing for *len* bytes with access permission *prot*. The *flags* parameter allows a process to specify whether it wants to make a *shared* or *private* mapping. Changes made to a shared mapping are written back to the file and are visible to other processes. Changes made to a private mapping

are not written back to the file and are not visible to other processes. Two processes that wish to share a piece of memory request a shared mapping of the same file into their address space. Thus, the existing and well-understood filesystem namespace identifies shared objects. The contents of the file are used as the initial value of the memory segment. All changes made to the mapping are reflected back into the contents of the file, so long-term state can be maintained in the shared memory region, even across invocations of the sharing processes.

Some applications want to use shared memory purely as a short-term interprocess-communication mechanism. They need an area of memory that is initially zeroed and whose contents are abandoned when they are done using it. Such processes want neither to pay the relatively high startup cost associated with paging in the contents of a file to initialize a shared memory segment nor to pay the shutdown costs of writing modified pages back to the file when they are done with the memory. Although FreeBSD does provide the limited and quirky naming scheme of the System V *shmem* interface as a rendezvous mechanism for such short-term shared memory (see Section 7.2), the designers ultimately decided that all naming of memory objects for *mmap* should use the filesystem namespace. To provide an efficient mechanism for short-term shared memory, mappings that do not require stability across system reboots use the MAP_NOSYNC flag to avoid the overhead of periodic syncing of dirty pages. When this flag is specified, dirty pages are only written to the filesystem when memory is in high demand.

When a mapping is no longer needed, it can be removed using

```
munmap(void *addr, size_t len);
```

The *munmap* system call removes any mappings that exist in the address space, starting at *addr* and continuing for *len* bytes. There are no constraints between previous mappings and a later *munmap*. The specified range may be a subset of a previous *mmap*, or it may encompass an area that contains many *mmap*'ed files. When a process exits, the system does an implied *munmap* over its entire address space.

During its initial mapping, a process can set the protections on a page to allow reading, writing, and/or execution. The process can change these protections later by using

```
mprotect(const void *address, int length, int protection);
```

This feature can be used by debuggers when they are trying to track down a memory-corruption bug. By disabling writing on the page containing the data structure that is being corrupted, the debugger can trap all writes to the page and verify that they are correct before allowing them to occur.

Traditionally, programming for real-time systems has been done with specially written operating systems. In the interests of reducing the costs of real-time applications and of using the skills of the large body of UNIX programmers, companies developing real-time applications now use UNIX-based systems for writing