FreeBSD Virtual-Memory System

- The FreeBSD virtual-memory is based on Mach 2.0 with updates from Mach 2.5 and Mach 3.0.
- The BSD implementation of this virtual-memory first appeared in 4.4BSD and has been ported to FreeBSD with only minor changes.
- Some characteristics of the system:
  - → Based on object oriented design principles
  - → A clean division between machine dependent and non machine dependent parts
  - → Support for sharing of memory between processes
  - → Support for multiprocessor systems
- The virtual-memory system implements a protected address space into which can be mapped data sources (objects) such as files.
- Physical memory is used as a cache for recently used pages from these objects.

Layout of Virtual Address Space for a Process

Traditional Unix processes have 4 segments

Text: Program code, R/O (or R/W for debugging) mapping of a binary file

Data: Initiated data, R/W mapping of a binary file

BSS: Uninitiated data, R/W anonymous mapping (zero-fill)

Stack: R/W anonymous mapping (zero-fill)

- The BSS area can be extended using the C-library *malloc()* routine, which allocates dynamic memory at the heap.
- The *malloc()* routine uses the *sbrk* system call to request extra memory from the system.
- FreeBSD also has the *mmap* system call that allows for creation of new memory segments at (in principle) arbitrary virtual addresses.

FreeBSD Virtual Memory - Data Structures

The same basic data structures are used to describe both kernel and user process virtual memory.

The following data structures are used (see fig. 5.4):

vmspace: Highest level. Encompasses both the machine-dependent and the machine-independent data structures.

vm_map: Highest level data structure that describes the machine-independent virtual address space.

vm_pmap: Machine dependent data structure that is used only by the machine-dependent parts of the virtual memory system.

vm_map_entry: Describes a virtually contiguous range of addresses that share protection and inheritance attributes.

object: Describes the source of data for a range of addresses.

Shadow object Object that represents modified copy of original data.

vm_page: Represents the physical memory. One vm_page structure exist for each page frame in the physical memory

Kernel Memory Management

- The kernel's virtual addresses are permanently mapped into the high part of every process address space.
- When the system boots, the first task the kernel must do is to set up data structures to describe its address space.
- The address space is described by a **vm_map** and a list of **vm_map_entry** structures.
- **submap** Is a kernel-only construct used to isolate allocation for kernel subsystems.
- The kernel virtual addresses are defined in ascending address order by the constants K0 to K8 (See fig. 5.5).

K0-K1: Text and data segment

K2-K6: submap

K2-K3: Kernel malloc area

K4-K5: Network buffer area

K7-K8: I/O area

Kernel Address-Space Allocation

- Kernel virtual-memory ranges can be either *wired* (locked into memory) or pageable.
- Wired memory is allocated with *kmem_alloc()* or *kmem_malloc()*.

kmem_alloc() Returns zero-filled memory and may block if insufficient physical memory is available. Only used to allocate memory from a specific kernel submap.

kmem_malloc() Variant of kmem_alloc() with a nonblocking option. Nonblocking allocations fail if insufficient physical memory is available. Nonblocking allocations must be used in interrupt routines.

kmem_free() Deallocates kernel memory.

- Pageable kernel virtual memory can be allocated with *kmem_alloc_pageable()* or *kmem_alloc_wait()*.
- Currently, pageable kernel memory is used only for temporary storage of *exec* arguments.

Kernel malloc

- The kernel also provides a generalized interface to the low-level memory allocation routines that is very similar to the C library *malloc()* and *free()* routines.
- In user mode, temporary storage is usually allocated on the stack.
- Because the kernel has a very limited run-time stack, all temporary kernel memory must be dynamically allocated using malloc().

Kernel Zone Allocator (Slab Allocator)

Some commonly used data structures in the kernel such as process control blocks are not well handled by the general purpose malloc().

These structures share several characteristics:

- They tend to be large and hence wasteful of space if a power-of-2 memory allocator is used.
- They are often linked together in long lists. If allocation of each structure begins on a page boundary, linkage pointers in the blocks will all be on the same line in the hardware cache, causing each step in the list traversal to generate a cache miss.
- They often contain locks and lists that must be initialized before use. If there is a dedicated pool of memory for each structure, these substructures need to be initialized only once.

The zone allocator solves these problems by allocating typed blocks of a fixed size from a reserved memory pool.

A memory pool for every type of object that are to use the zone allocator must be created with the call *uma_zcreate()*.

Process Virtual-Address Space

- When a process is created, memory is allocated for the text, data, bss and stack segments.
- The address space for a process is described by a *vmspace* structure that points to a list of *vm_map_entry* structures (see fig. 5.6)
- Each *vm_map_entry* structure describes a region of virtual address space residing between a *start* address and an *end* address. It also have a pointer, to the object that provides the initial data for the region, and an offset that describes where within the object the data begins.

Page-Fault Dispatch

When a process references a page that is not currently resident, a page fault occurs. The page-fault handler in the kernel is presented with the virtual address that caused the fault.

The fault is handled with the following four steps:

1. Find the vmspace structure for the faulting process.
2. Traverse the *vm_map_entry* list to locate the segment that includes the faulting segment. If the address is not found it was an illegal reference and a *segmentation fault* signal is generated.
3. When the correct vm_map_entry is found, the address is converted to an offset within the underlying object.
4. Present the absolute object offset to the underlying object, which allocates a *vm_page* structure and calls its pager to fill the page frame with data.

Objects

- Objects are used to hold information about either a file or about an area of anonymous memory.
- Whether a file is mapped by a single process or by many processes, it will always be represented by a single object. All *vm_map_entry* structures referencing the same file will point to the same *object* structure.
- An object stores the following information:
  - → A list of vm_page structures for all pages in the object that are currently resident in main memory.
  - → The number of page frames held by the object.
  - → A reference count on the number of *vm_map_entry* structures and other objects that reference the object.
  - → The size of the file or anonymous data area described by the object.
  - → Pointer to shadow objects.
  - → Pointer to *pager* for the object.

Types of Objects

There are three types of objects in the system:

Named objects represent files or certain hardware devices.

Anonymous objects represent areas of memory that are zero filled at first use and abandoned when they are no longer needed.

Shadow objects hold private copies of pages that have been modified; they are abandoned when they are no longer referenced.

- The type of an object is defined by the type of pager the object uses.
- A *named object* uses the *vnode* pager if it is backed by a file and the device pager if it maps a hardware device.
- Anonymous objects use the *swap pager*.
- Shadow objects also use the *swap pager*.

Objects to Pages

When the system boots, memory is first assigned to the kernel.

All remaining memory can be used for page frames.

- Every page frame is described by a *vm_page* structure that point to the physical memory address for the page.
- Initially, all the *vm_page* structures are placed on the memory *free list* and marked as *free*.
- The first time a page frame is allocated to an object, its *vm_page* structure is moved to the object's vm_page list and marked as *active*.
- If a page is already present in memory, it can be located using a hashing mechanism that maps <object, offset> to *vm_page*.

Mmap - Shared Memory

A process can add a new memory area to its address space using the system call *mmap*.

The mmap system call maps a file into the process virtual address space.

```
mmap(caddr_t addr, size_t len, int prot,
          int flags, int fd, off_t offset)
```

addr	virtual base address for the segment
len	segment length
prot	read, write or exec
flags	*map_shared*, *map_private* or *map_anon*
fd	file descriptor for the file to map
offset	the mapping begins at address offset in file

Mmap

- Two processes can create a shared memory area by requesting a shared mapping of the same file.
  - → Changes in 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 and are invisible to other processes.
- If the *anon* flag is set, an anonymous area for temporary process communication is created.
- If *map_anon* is set, filedescriptor should be -1
- Another way to get a temporary area for process communication is to map a file on a memory-resident file system.

Shared Memory - More System Calls

A mapped memory region can be removed with the system call:

```
munmap(caddr_t addr, size_t len)
```

A process can change the protection on a memory region with:

```
mprotect(caddr_t addr, int len, int prot)
```

A process can prevent a memory region from being paged out with:

```
mlock(caddr_t addr, size_t len)
```

- Can be used by processes with soft real-time requirements.
- To prevent a single process from acquiring all physical memory, there is a limit on the amount of memory that may be locked.

Shared Memory - More System Calls

A locked memory region can be unlocked with *unlock*.

A process can force all modified pages in a specific memory region to be rewritten to disk with:

```
msync(caddr_t addr, int len)
```

Only modified pages in within the specified region are rewritten. Has no effect on anonymous regions.

Shared Mapping - implementation

- Each mapping that a process has to a file is described by a *vm_map_entry* structure.
- A shared region is described by only one *object* structure and all *vm_map_entry* structures that reference the shared region point to the same object structure. (see fig. 5.7)

Private Mapping - Implementation

- If a process has requested a private mapping, changes to the memory mapping are not visible to other processes and are not written back to the file.
- A private mapping is set up as COW (Copy On Write) when it is created.
- When a process writes to the region, the kernel makes a copy of the page and creates a *shadow object* to describe the modified page (see fig. 5.8).
- In fig. 5.8 process A has modified page 0 of the privately mapped file object. The kernel has copied the page to a *shadow object*.
- When a page fault for a private mapping occurs, the kernel traverses the list of objects headed by the vm_map_entry. The first object in the list that has the desired page is used.
- If the search reaches the last object in the list the page is requested from the file object.

Private Mapping - fork

- When a process forks, its list of vm_map_entry structures is copied to the child process. All privately mapped regions are marked COW both in the parent and child processes.
- When any of the processes writes to a private region, the page is copied and a new shadow object is created (see fig.5.9).
- When a private mapping is removed (due to *munmap* or *exit*) all pages in the shadow object are moved to the free list. The content is not rewritten to the file.
- When a child process terminates, the parent is often left with a chain of shadow objects that are not needed any more.
- These shadow objects can be collapsed to a single shadow object.
- Unfortunately the collapse of shadow object chains is complicated and may be time consuming. FreeBSD uses a more efficient method for this than earlier versions of BSD.