Virtual Memory Initialization and Management in the FreeBSD Operating System

(Inicjalizacja i zarządzanie
pamięcią wirtualną
w systemie operacyjnym FreeBSD)

Jakub Piecuch

Praca licencjacka

Promotor: dr Piotr Witkowski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

17 sierpnia 2018

Abstract

Virtual memory is one of the most important abstractions provided to user space programs by modern operating system kernels. While it greatly simplifies application development, implementing it efficiently is a great challenge. The FreeBSD operating system kernel VM (Virtual Memory) subsystem succeeds at this goal. However, in doing so it employs complex data structures and algorithms, which make it difficult to understand for newcomers to the kernel. This thesis provides an overview of the architecture of the FreeBSD VM subsystem, along with fragments of real-world (albeit simplified) source code that implement its most important functions. Subsequently, it goes through the initialization of the VM subsystem on the x86 architecture. It also acts as a guide to reading the kernel's source code.

Pamięć wirtualna jest jedną z najważniejszych abstrakcji udostępnianych programom użytkownika przez jądra nowoczesnych systemów operacyjnych. Znacznie upraszcza ona budowanie aplikacji, jednak efektywna jej implementacja stanowi duże wyzwanie. Jądro systemu operacyjnego FreeBSD jest przykładem efektywnej implementacji pamięci wirtualnej. Niestety, osiągnięcie dobrej wydajności wymaga zastosowania złożonych struktur danych i algorytmów, co sprawia że osobom niezaznajomionym z jądrem trudno jest zrozumieć działanie podsystemu pamięci wirtualnej.
...

Contents

Przykłady komentarzy do tekstu

PWit: Komentarz w treści

✠

Cahir: Komentarz w treści

✠

Author: Odpowiedź na komentarz w treści

✠

PWit: Dodałem makra umożliwiające komentowanie i odpowiedź na komentarze. Wymagało to dodania wiersza do do iithesis.cls, ale nie powinno mieć to wpływu na układ tekstu. Po napisaniu i zaakceptowaniu pracy można te zmiany usunąć.

PWit: Komentarz na boku

Cahir: Komentarz na boku

Author: Odpowiedź na komentarz na boku

Chapter 1

Introduction

1.1 What is virtual memory?

A system implementing virtual memory provides every process in the system with an address space of its own, its **Virtual Address Space** (VAS). The modifications that a process makes to its own VAS are not visible to other processes, unless they explicitly choose to share fragments of their address spaces. This protects running programs against unwanted interference.

The size of the VAS can (and typically is on 64 bit architectures) much larger than the amount of available **physical memory**, i.e. memory actually installed in the system. For instance, the VAS size on the AMD64 architecture is 2^{48} bytes, that is 256 TiB (tebibytes)[1].

The VAS of any process contains the whole **process image**, that is the instructions, data and the run-time stack. A process in execution accesses memory to fetch its instructions, as well as to read and write data. All instructions executed by the process access memory using **Virtual Addresses** (VAs), which are integers in the range from 0 to $N - 1$, where N is the VAS size.

Since the VAS size can be larger than the amount of available physical memory, there cannot be a 1-to-1 correspondence between VAs and **Physical Addresses** (PAs), which refer directly to locations in physical memory. It is necessary to allow either VAs which have no corresponding PAs, or multiple VAs with the same corresponding PA. To accomplish this, a scheme called **address translation** is used.

Modern architectures provide hardware support for address translation in the form of a **Memory Management Unit** (MMU). The MMU translates VAs generated by the program into PAs used to address the physical memory. As the details are architecture-specific, this thesis focuses on the x86 architecture.

[1] 1 TiB = 1024 GiB (gibibytes). A gibibyte is slightly larger than a gigabyte.
See `https://en.wikipedia.org/wiki/Binary_prefix`.

In the x86 architecture, the mapping between VAs and PAs is determined by an in-memory data structure called a **Page Table** (PT). The VAS is divided into units called **pages**. Their size is usually 4 KiB. Likewise, the physical memory is divided into page-sized units called **page frames**. The Page Table describes the mapping between pages and page frames, instead of between individual virtual addresses and physical addresses. This significantly reduces the size of the Page Table.

In addition to specifying the mapping between pages and page frames, the Page Table also determines the protection attributes of each page. For instance, pages containing program code can be marked read-only, e.g. so that a malicious user can't exploit a bug in the program to overwrite the original instructions with malicious ones.

Not every page needs to be mapped to a page frame. Whenever the program references an address which is in a page that is not mapped, a **page fault** occurs. The MMU detects this and the CPU generates an exception. Execution then jumps to a routine provided by the OS, which handles the exception. The same is done when the programs tries to do something to a page that violates the page's protection attributes, e.g. write to a read-only page.

An important observation is that with virtual memory, processes can be partially **resident** in physical memory (i.e. only parts of their image have to be in physical memory). Unused pages can be unmapped, as an exception will generated only when a process references the page. Furthermore, even pages that are used by a process don't have to be resident at all times, as the OS can map them into the VAS in response to a page fault, and then restart the instruction that caused it. This strategy is called **demand paging** and is widely used. It enables the system to use memory resources efficiently by holding only as many mapped pages in physical memory as are necessary for any one process to run smoothly.

In summary, virtual memory has the following important characteristics:

- The address spaces in which processes live are isolated from one another
- A process may run even if the total amount of memory it requires is larger than the total amount of available physical memory
- A process can have its memory pages mapped into its address space on demand, and unmapped when the OS determines the process is unlikely to reference them in the near future

1.2 The FreeBSD Operating System

FreeBSD is an open-source operating system, first released in 1993. It is the most popular OS in the BSD family of Operating Systems, which includes FreeBSD,

NetBSD, OpenBSD and DragonFlyBSD. Unlike Linux, FreeBSD is a whole Operating System, providing the kernel, drivers and a suite of utility programs.

It is released under a permissive BSD license, which makes it an attractive choice for commercial applications. FreeBSD has been used as the basis for Operating Systems such as Apple's MacOS, as well as the OS running on Sony's PlayStation 3 and PlayStation 4 consoles.

1.2.1 Browsing the source code

This thesis includes many source code listings from the FreeBSD repository. Because of its scope, the level of implementation detail covered is limited. Should the reader want to explore the source tree themselves (which is strongly encouraged), there are several websites providing the code of the whole FreeBSD kernel with support for identifier search:

(1) `http://fxr.watson.org/`

(2) `http://bxr.su/`

The author recommends **??**, as it allows the user to select any major version of the source tree, while **??** only presents the latest revision. This is especially relevant since all source code discussion and listings apply specifically to the `11-STABLE` version of the kernel. On the other hand, **??** has a more aesthetically pleasing interface, which includes syntax highlighting.

The kernel source code can be found in the `/sys` directory of the tree. Since other parts of the tree are not going to be discussed, all subsequent paths are relative to this directory.

The `kern` directory holds most of the kernel's portable code, like the scheduler, subsystem initialization and shutdown, and the Virtual File System.

The `vm` directory contains the machine-independent part of the Virtual Memory subsystem. This is where most of the code listings will come from. Machine-dependent parts of the subsystem can be found in the directories named after a particular architecture, e.g. the `i386` directory for the x86 architecture. This is quite confusing, since the kernel source tree contains a directory named `x86` – this directory contains architecture-specific code that is shared between the AMD64 and x86 architectures.

Chapter 2

An overview of the FreeBSD VM subsystem

The goal of this chapter is to describe the data structures used for managing virtual memory in the FreeBSD Operating System. For each one, a brief high-level introduction is given, followed by a source code listing that shows the C language structure definition. Important member fields are then individually described below the listing.

2.1 vmspace

The `vmspace` structure is the highest-level structure describing the virtual address space of a process. It contains both the machine-independent (`vm_map`) and machine-dependent (`pmap`) structures used for describing a mapping. The other fields hold various statistics and parameters and are not relevant to the discussion.

```
struct vmspace {
    struct vm_map vm_map;           /* VM address map */
    /* Certain fields omitted */
    caddr_t vm_maxsaddr;            /* User VA at max stack growth */
    volatile int vm_refcnt;         /* Number of references */
    struct pmap vm_pmap;            /* Physical map */
};
```

Listing 1: `vm/vm_map.h`: Definition of `struct vmspace`

2.2 vm_map

The vm_map structure represents the machine-independent part of a virtual address space. It is structured as a tree of vm_map_entry structures, each of which describes a continous fragment of the address space.

```
struct vm_map {
    struct vm_map_entry header; /* List of entries */
#define min_offset  header.end
#define max_offset  header.start
    struct sx lock;             /* Lock for map data */
    struct mtx system_mtx;
    int nentries;               /* Number of entries */
    vm_size_t size;             /* Virtual size */
    u_int timestamp;            /* Version number */
    u_char needs_wakeup;
    u_char system_map;          /* Am I a kernel map? */
    vm_flags_t flags;           /* Flags for this vm_map */
    vm_map_entry_t root;        /* Root of a binary search tree */
    pmap_t pmap;                /* Pointer to physical map */
    int busy;
};
```

Listing 2: vm/vm_map.h: Definition of struct vm_map

```
struct vm_map_entry header;
```

This vm_map_entry is used for holding the minimum and maximum virtual address for use by the user. It also serves as the header of a linked list of all vm_map_entry structures in the vm_map, sorted by start address. It is used for quickly retrieving the immediate left and right neighbours of a vm_map_entry, as well as iterating over all entries in a vm_map.

```
struct sx lock;
struct mtx system_mtx;
```

These are synchronization tools used to manage concurrent access to the map. If the value of system_map is TRUE, system_mtx is used exclusively, otherwise only lock is used.

```
u_int timestamp;
```

The value of this field is incremented each time exclusive access is acquired to read or modify the map. This is so that algorithms that require relinquishing and reacquiring the lock after some time can detect if someone has possibly tampered with the data structure in the meantime.

```
u_char needs_wakeup;
```

This is a flag indicating that there is a thread (or threads) waiting for a large enough chunk of free address space to satisfy its allocation request. Whenever space is freed from the map and the flag is set, waiting threads are woken up to retry the allocation.

```
vm_map_entry_t root;
```

This is a pointer to the root of the binary tree used to look up entries in the map. The tree uses the self-balancing splay algorithm by Tarjan and Sleator. The most recently looked up entry is at the root of the tree, which speeds up page fault handling by taking advantage of the spatial locality of page faults (i.e. when a page fault happens, the next one is likely to happen at an address close to the previous one).

```
int busy;
```

The `busy` field is an integer counter that is incremented whenever a thread is in the middle of performing an operation on the `vm_map` and has to release the lock for some reason, e.g. to wait for some event to occur. This signalizes to other threads that some other thread is relying on the map being in the same state as it was before it relased the lock. Thus, effectively, when the value of the counter is greater than 0, modifications to the `vm_map` are forbidden.

2.3 vm_map_entry

The `vm_map_entry` structure describes a contiguous, page-aligned segment of an address space. All virtual addresses within this segment have the same protection attributes. The `vm_map_entry` is not responsible for providing the contents of the segment, that's the job of the `vm_object` that is **backing** the entry.

```
struct vm_map_entry {
   struct vm_map_entry *prev;  /* Previous entry */
   struct vm_map_entry *next;  /* Next entry */
   struct vm_map_entry *left;  /* Left child in binary search tree */
   struct vm_map_entry *right; /* Right child in binary search tree */
   vm_offset_t start;          /* Start address */
   vm_offset_t end;            /* End address */
   vm_offset_t next_read;      /* Vaddr of the next sequential read */
   vm_size_t adj_free;         /* Amount of adjacent free space */
   vm_size_t max_free;         /* Max free space in subtree */
   union vm_map_object object; /* Object I point to */
   vm_ooffset_t offset;        /* Offset into object */
   vm_eflags_t eflags;         /* Map entry flags */
   vm_prot_t protection;       /* Protection code */
   vm_prot_t max_protection;   /* Maximum protection */
   vm_inherit_t inheritance;   /* Inheritance */
   uint8_t read_ahead;         /* Pages in the read-ahead window */
   int wired_count;            /* can be paged if = 0 */
   struct ucred *cred;         /* tmp storage for creator ref */
   struct thread *wiring_thread;
};
```

Listing 3: vm/vm_map.h: Definition of struct vm_map

```
struct vm_map_entry prev;
struct vm_map_entry next;
```

These pointers link the entry into the doubly linked list of all entries within a map, sorted by start address.

```
vm_offset_t next_read;
```

This field is used by the kernel to detect when a process accesses pages in the segment sequentially. When the kernel knows the accesses are sequential, it will bring into physical memory pages beyond the faulting one, since it is reasonable to assume that the pattern of accesses will continue, and therefore reduce the number of page faults generated by the process.

```
vm_size_t adj_free;
vm_size_t max_free;
```

`adj_free` is the amount of free space (in bytes) between this entry and the adjacent entry to the right. It is used when searching the `vm_map` for a free segment of a certain size. `max_free` is simply the maximum of the values of `adj_free` of all entries inside the subtree rooted at this entry.

```
union vm_map_object object;
```

This field is either a pointer to a `struct vm_object`, or a pointer to another `struct vm_map` called a submap. Almost every time it is the former, as submaps are only used inside the kernel map to allocate space in advance for certain data structures.

```
vm_ooffset_t offset;
```

The offset determines which part of the `vm_object` is accessible through the `vm_map_entry`. Let `ent` be a `vm_map_entry` backed by some object. Virtual addresses from `ent.start` to `ent.end - 1` (inclusive) map to offsets `ent.offset` to `ent.offset + (ent.end - end.start) - 1` within the object.

```
vm_eflags_t eflags;
```

`eflags` contains various flags, some notable of which are:

- `MAP_ENTRY_COW`: indicates that the entry is a **copy-on-write** entry, which means that initially all pages are marked read-only. As soon as the process attempts to write to a page in the enty, the kernel will copy the faulting page to a new page and map the new page into the address space of the process, this time with the write bit set. This avoids unnecessary copying of pages when a process forks or requests a private mapping of a file (i.e. a mapping such that changes to it aren't reflected in the file).
- `MAP_BEHAV_{NORMAL, SEQUENTIAL, RANDOM}`: these flags specify the access pattern that is to be expected from the process. The default is `MAP_BEHAV_NORMAL`, which makes the kernel detect sequential access patterns and act accordingly.

```
vm_prot_t protection;
vm_prot_t max_protection;
```

These two fields specify the current protection attributes of the entry and the maximum allowable access rights, respectively. Both are bitmasks composed of 3 fields: `VM_PROT_READ`, `VM_PROT_WRITE`, and `VM_PROT_EXECUTE`.

```
vm_inheritance_t inherit;
```

The `inherit` field determines what happens to the entry when the process forks. The possible behaviours are:

- `VM_INHERIT_SHARE`: The child gets an entry that shares the underlying object with the parent. Changes made by one process are visible to the other.

- `VM_INHERIT_COPY`: The child gets a copy-on-write entry with contents identical to the parent entry's contents. Changes made by one process are not visible to the other.

- `VM_INHERIT_ZERO`: The child gets an anonymous zero-filled entry with the same size and protection as the parent entry.

- `VM_INHERIT_NONE`: The entry won't appear in the child.

2.4 vm_object

A `vm_object` contains a `vm_map_entry`'s resident pages. Multiple entries belonging to different processes can have the same backing object, allowing for fast interprocess communication through shared memory.

Copy-on-write is implemented in BSD using special **shadow objects**. These objects have backing objects themselves, and hence can form chains ending in a non-shadow object. Shadow objects hold pages copied as a result of a copy-on-write fault.

Since `vm_objects` are only containers for resident pages, and not all pages are always resident, there must be some other component responsible for providing the contents of the pages. This is the job of a **pager** structure contained within the object. It provides the abstraction of a backing store from which pages can be filled with contents, and to which pages can be written back in case of memory shortage.

```
struct vm_object {
    struct rwlock lock;
    /* List entry used to link into the list of all vm_objects */
    TAILQ_ENTRY(vm_object) object_list;
    /* List of objects shadowing this object */
    LIST_HEAD(, vm_object) shadow_head;
    /* List entry used to link into shadow_head of shadowed object */
    LIST_ENTRY(vm_object) shadow_list;
    struct pglist memq;     /* List of resident pages */
    struct vm_radix rtree;  /* Root of resident page radix trie */
    vm_pindex_t size;       /* Object size */
    /* Certain fields omitted */
    /* How many vm_map_entries/vm_objects reference this object*/
    int ref_count;
    int shadow_count;       /* Length of linked list at shadow_head */
    vm_memattr_t memattr;   /* Default memory attribute for pages */
    objtype_t type;         /* Type of pager */
    /* Certain fields omitted */
    int resident_page_count;    /* Number of resident pages */
    struct vm_object *backing_object;   /* Object that I'm a shadow of */
    vm_ooffset_t backing_object_offset; /* Offset in backing object */
    /* List of all objects of this pager type */
    TAILQ_ENTRY(vm_object) pager_object_list;
    LIST_HEAD(, vm_reserv) rvq; /* List of reservations */
    void *handle;   /* Opaque pointer used by the pager */
    union {
        /* Various pager structures omitted */
    } un_pager;
    /* Certain fields omitted */
};
```

Listing 4: `vm/vm_object.h`: Definition of `struct vm_object`

```
vm_memattr_t memattr;
```

`memattr` specifies the default cache behaviour of pages belonging to the object. For example, contents of pages with the `VM_MEMATTR_UNCACHEABLE` attribute cannot be cached and all read and write requests have to go directly to the physical memory. This attribute is used primarily for pages representing memory mapped devices, e.g. frame buffers.

```
objtype_t type;
```

There are several types of pagers that can provide the contents of memory pages to a `vm_object`. The `type` field determines the type of pager used by the object. The most widely used types of pagers are:

- `OBJT_SWAP`: The swap pager is used to provide contents of anonymous memory segments (i.e. segments that are not backed by a file and are initially filled with zeros). When asked to fill a page with contents, it first checks if the page had been swapped out before. If so, the contents are fetched from secondary storage – either a swap file or a dedicated swap partition. If not, the page is simply filled with zeros. In case of memory shortage, the swap pager may be asked to store the contents of a page in backing store.

- `OBJT_DEFAULT`: This is the pager type used in all newly created anonymous mappings (i.e. zero-filled mappings not backed by any file). It fills all new pages with zeros. This pager type is an optimization of the swap pager for the common case where the pages never need to be swapped out to backing store, as memory resources in today's systems are abundant. Not needing to keep track of swap space speeds up the pager's initialization procedures. When there is a need for an object with a default pager to write some of its pages to backing store, its pager is changed to the swap pager.

- `OBJT_VNODE`: The contents of pages backed by a file are supplied and written back by the vnode pager. An object with a vnode pager acts as a general-purpose cache of the backing file's pages, used not only by the virtual memory subsystem (i.e. when a process maps a file), but also when a process reads files via file descriptors.

- `OBJT_DEVICE`: The device pager manages pages belonging to objects which represent memory-mapped physical devices. The pages used by the device pager are different from ordinary pages in that they don't represent a frame of physical memory, or, in FreeBSD terminology, they are **fictitious**.

2.5 `vm_page`

The `vm_page` structure represents a unit of physical address space that can be mapped into a virtual address space. Every `vm_page` has a physical address, although it does not need to be a physical address that can be used to address physical memory. Most pages, though, are pages representing frames of physical memory. Pages with physical addresses beyond the physical memory range are called **fictitious** pages and are primarily used for accessing memory-mapped devices.

```
struct vm_page {
    union {
        TAILQ_ENTRY(vm_page) q; /* Page queue or free list */
        /* Some union fields omitted */
    } plinks;
    TAILQ_ENTRY(vm_page) listq; /* Pages in same object */
    vm_object_t object;         /* Which object am I in */
    vm_pindex_t pindex;         /* Offset into object */
    vm_paddr_t phys_addr;       /* Physical address of page */
    struct md_page md;          /* Machine dependent stuff */
    u_int wire_count;           /* Wired down maps refs */
    /* Some fields omitted */
    uint16_t flags;             /* page PG_* flags */
    uint8_t aflags;             /* Flags with atomic access */
    uint8_t oflags;             /* page VPO_* flags */
    /* Omitted */
    u_char  act_count;          /* page usage count */
    /* Omitted */
};
```

Listing 5: `vm/vm_object.h`: Definition of `struct vm_object`

```
struct md_page md;
```

This field contains per-page information maintained by the `pmap` module, i.e. the module responsible for the machine-dependent part of the mapping. Here is the definition of `struct md_page` for the x86 architecture:

```
struct md_page {
   TAILQ_HEAD(,pv_entry) pv_list;
   int                   pat_mode;
};
```

Listing 6: `i386/include/pmap.h`: Definition of `struct md_page`

The `pv_list` field is a list of structures identifying virtual addresses pointing to this page (there can be many of them, since a single `vm_object` can be shared by different `vm_map_entry` structures). The `pat_mode` field specifies caching attributes for the page.

```
uint16_t flags;
uint8_t aflags;
uint8_t oflags;
```

The flags for an object are split into several fields due to different characteristics (e.g. locks needed to read/modify the flag, whether operations need to be atomic). Flags contained in these fields include:

- `PGA_REFERENCED`: An access has recently been made to the page.
- `PG_FICTITIOUS`: The page doesn't represent a real physical frame.

2.6 pmap

The `pmap` structure encapsulates the architecture-dependent aspects of the mapping. Most architectures provide some kind of hardware support for paging, for example the MMU on the x86 architecture. the MMU contains a hardware **page table walker**, which traverses the in-memory page table each time a memory access is made and the needed mapping isn't cached in the TLB. The TLB (Translation Look-aside Buffer) is simply a cache that contains mappings of recently accessed pages). The hardware page table walker expects the page table and page table entries to have a specific layout. This layout is exactly what is implemented in the x86 version of the `pmap` structure.

2.6.1 x86 page table layout

Logically, the page table is an array of 4-byte **Page Table Entries** (PTEs). The virtual address used to access memory is split into a **Virtual Page Number** (VPN) and an offset into the page. On x86, the VPN consists of the 20 most significant bits of the address, while the 12 least significant bits are the offset. The VPN is used to find the appropriate PTE.

Each entry contains a **valid bit**, which tells whether or not the corresponding page is mapped to a frame of physical memory. The bit is set if and only if the page is mapped. Every entry also contains the physical address of the frame to which the page is mapped, provided the valid bit is set. The entries also describe the page's protection attributes (whether write access is permitted), cacheability attributes (whether its contents can be cached at all, as well as whether the write-through or write-back policy should be applied), and minimum privilege level required to access the page (user or supervisor).

To help the OS to determine which resident pages are actively used and which are not, the PTEs contain a **referenced bit** and a **dirty bit** which are set by the hardware and can be read by the OS e.g. during a page fault.

The last field contained in PTEs is the **global bit**. When it is set, the entry containing it won't get flushed from the TLB when the current page table is changed. Since the kernel is usually mapped into the same region of address space for every process, this bit can be used to avoid unnecessary page faults after performing a context switch by setting it in all entries mapping the kernel.

The size of the address space is 2^{32} bytes and each page is $2^{12} = 4096$ bytes large. Therefore, to describe the entire address space of a single process as a flat array of PTEs, we would need $4 * \frac{2^{32}}{2^{12}} = 2^{22}$ bytes, or 4 MiB. With hundreds of concurrently running processes on a single system not being uncommon, the amount of memory needed to store the page tables of resident processes would quickly become a bottleneck preventing the system from increasing its degree of multiprogramming (i.e. how many processes are resident in memory at the same time).

The solution to this problem is to give the page table a hierarchical structure. The virtual address, instead of being split into 2 parts, is split into 3 parts. The first and second parts (bits 22-31 and 12-21 respectively, where 0 is the least significant bit and 31 is the most significant bit).

Bits 22-31 are used as an index into the **Page Directory**, an array of 4-byte **Page Directory Entries** (PDEs). Each PDE contains a pointer to a single page-sized chunk of the page table. However, if that chunk contains no valid entries, no physical memory is allocated for it and the PDE that points to it has its valid bit cleared. This allows for significant memory savings, since usually most of a process's address space is unmapped.

Once the physical address of the page table chunk is extracted from the PDE, bits 12-21 are used to index the chunk and retrieve the corresponding PTE, whose structure has been described above.

```
/* PGPTD = size of page directory in pages (1 in this case) */
struct pmap {
    struct mtx          pm_mtx;
    pd_entry_t          *pm_pdir;       /* VA of page directory */
    TAILQ_HEAD(,pv_chunk)   pm_pvchunk; /* List of mappings in pmap */
    cpuset_t            pm_active;      /* Set of CPUs using this page table */
    struct pmap_statistics  pm_stats;   /* pmap statistics */
    LIST_ENTRY(pmap)    pm_list;        /* List of all pmaps */
    struct vm_radix     pm_root;        /* Tree of spare page table pages */
    /* Table of pointers to page directory pages */
    vm_page_t           pm_ptdpg[NPGPTD];
};
```

Listing 7: `i386/include/pmap.h`: Definition of `struct pmap`

Chapter 3

Virtual Memory management

In this chapter we take a look at the inner workings of the most important functions carried out by the operating system when it comes to managing a process's virtual address space. The order in which the functions will be discussed roughly corresponds to the order in which these functions are invoked during the lifetime of a process.

The first function we will discuss is the creation of a new address space. A new virtual address space is created whenever a new process enters the system. All processes in FreeBSD are created using a mechanism called **forking**. When a process forks, an almost identical copy of it is created by the kernel. The copy is called the **child**, and the forking process is called the **parent**. The layout of the child's virtual address space is identical to the one of its parent.

Usually, new processes are created in order to run some executable program stored in a file. To do that, a process invokes the `execve` system call. The operating system then completely replaces the contents of the process's address space with the process image contained in the file.

Once the program begins to execute, it accesses instructions and data using virtual addresses. Some of these accesses will generate **page faults**, and the kernel has to handle them appropiately, fetching data from secondary storage or terminating the process if it tried to access an unmapped region.

A process can also modify the layout of its own address space. The `mmap` system call allows a process to create a new segment of virtual memory that it can access, or overlay an existing one. The initial contents of these segments can be provided by a file from the file system or simply filled with zeros.

Each section of this chapter will describe the steps taken by the kernel when carrying out each of these functions.

3.1 Address space creation – the `fork` system call

The `fork` system call creates an almost identical clone of the calling process, which includes an address space cloned from the address space of the calling process.

The `vmspace_fork` is used by the code implementing the system call to create a copy of an address space. The function takes a pointer to the `vmspace` structure of the calling process as an argument and returns a pointer to the copy which it has created.

The code listing below is a simplified version of the actual implementation of the `vmspace_fork` function. Fragments of code which deal with edge cases, synchronization and accounting have been removed for clarity.

```
struct vmspace *
vmspace_fork(struct vmspace *vm1, vm_ooffset_t *fork_charge)
{
    struct vmspace *vm2;
    vm_map_t new_map, old_map;
    vm_map_entry_t new_entry, old_entry;
    vm_object_t object;

    old_map = &vm1->vm_map;
    /* Allocate a new vmspace structure with the given address range */
    vm2 = vmspace_alloc(old_map->min_offset, old_map->max_offset, NULL);
    /* Copy immutable fields of vm1 to vm2. */
    vm2->vm_taddr = vm1->vm_taddr;
    vm2->vm_daddr = vm1->vm_daddr;
    vm2->vm_maxsaddr = vm1->vm_maxsaddr;
    new_map = &vm2->vm_map;
    old_entry = old_map->header.next;

    /* Iterate over all entries in old_map */
    while (old_entry != &old_map->header) {

        switch (old_entry->inheritance) {
        case VM_INHERIT_NONE:
            /* Don't clone the entry to new_map */
            break;

        case VM_INHERIT_SHARE:
            /*
             * Clone the entry, but share the underlying vm_object between
             * both entries.
             */

            /*
             * The object may not have been created.
```

```
 * Create it if that's the case.
 */
object = old_entry->object.vm_object;
if (object == NULL) {
    object = vm_object_allocate(OBJT_DEFAULT,
        atop(old_entry->end - old_entry->start));
    old_entry->object.vm_object = object;
    old_entry->offset = 0;
}

/*
 * Add the reference before calling vm_object_shadow
 * to insure that a shadow object is created.
 */
vm_object_reference(object);
if (old_entry->eflags & MAP_ENTRY_NEEDS_COPY) {
    /*
     * old_entry is a copy-on-write entry and its shadow
     * object hasn't been created yet (which is possible
     * because shadow objects are created only when necessary).
     * We need to create the shadow object here.
     * If we didn't, the entries in the child and parent
     * would get separate shadow objects if they attempted
     * to write to the region of memory represented by this
     * entry, which is not what we want.
     * We want the changes made by the parent to be visible
     * to the child and vice versa.
     */
    vm_object_shadow(&old_entry->object.vm_object,
        &old_entry->offset,
        old_entry->end - old_entry->start);
    old_entry->eflags &= ~MAP_ENTRY_NEEDS_COPY;

    /* Transfer the second reference too. */
    vm_object_reference(
        old_entry->object.vm_object);

    /*
     * Now, instead of both the parent and child entries
     * referencing object, only the shadow object references
     * it, therefore we decrement the reference counter.
     */
    vm_object_deallocate(object);
    object = old_entry->object.vm_object;
}

/*
 * Clone the entry, referencing the shared object.
 */
```

```
        new_entry = vm_map_entry_create(new_map);
        *new_entry = *old_entry;

        /*
         * Insert the entry into the new map -- we know we're
         * inserting at the end of the new map.
         */
        vm_map_entry_link(new_map, new_map->header.prev,
            new_entry);
        /*
         * Do some bookkeeping associated with the sizes of
         * the text, data and stack segments.
         */
        vmspace_map_entry_forked(vm1, vm2, new_entry);

        /*
         * Update the physical map
         */
        pmap_copy(new_map->pmap, old_map->pmap,
            new_entry->start,
            (old_entry->end - old_entry->start),
            old_entry->start);
        break;

    case VM_INHERIT_COPY:
        /*
         * Clone the entry and link into the map.
         */
        new_entry = vm_map_entry_create(new_map);
        *new_entry = *old_entry;

        new_entry->object.vm_object = NULL;
        vm_map_entry_link(new_map, new_map->header.prev,
            new_entry);
        vmspace_map_entry_forked(vm1, vm2, new_entry);
        /*
         * Make old_entry and new_entry point to the same vm_object.
         * Both entries are marked copy-on-write so that the initial
         * contents of the memory segment are identical to the
         * parent and the child processes, but changes made by one
         * are not visible to the other.
         */
        vm_map_copy_entry(old_map, new_map, old_entry,
            new_entry, fork_charge);
        break;

    case VM_INHERIT_ZERO:
        /*
         * Create a new anonymous mapping entry modelled from
```

```
                 * the old one.
                 */
                new_entry = vm_map_entry_create(new_map);
                memset(new_entry, 0, sizeof(*new_entry));

                new_entry->start = old_entry->start;
                new_entry->end = old_entry->end;
                new_entry->eflags = old_entry->eflags &
                    ~(MAP_ENTRY_USER_WIRED | MAP_ENTRY_IN_TRANSITION |
                    MAP_ENTRY_VN_WRITECNT);
                new_entry->protection = old_entry->protection;
                new_entry->max_protection = old_entry->max_protection;
                new_entry->inheritance = VM_INHERIT_ZERO;

                vm_map_entry_link(new_map, new_map->header.prev,
                    new_entry);
                vmspace_map_entry_forked(vm1, vm2, new_entry);


                break;
            }
            old_entry = old_entry->next;
        }

        return (vm2);
}
```

Listing 8: `vm/vm_map.c`: Simplified implementation of the `vmspace_fork` function

3.2 Executing a new program – the `execve` system call

A process may request the operating system to replace the program that is currently executing in the process with another program stored as a file in the file system. To do this, the process invokes the `execve` system call.

Among other things, the `execve` system call is responsible for wiping the address space of the calling process and populating it with the image contained in the file supplied by the caller.

The bulk of `execve` logic is contained in the `do_execve` function, defined in `kern/kern_exec.c`. However, most of the address space management logic is implemented elsewhere. Hence, we will not the code for `do_execve` here.

To understand the code responsible for populating an empty address space with a program contained in a file, it is helpful to understand the concept of **image activators**.

The kernel supports multiple different file formats that contain an executable program. For instance, the file pointed to by the path passed to `execve` can be a plain-text script, with the first line containing the path to the interpreter program preceded by the characters `#!`. For example, if a script begins with the line `#!/bin/bash`, this is information to the kernel that the actual program to be executed is the interpreter, and the path to the script should be passed as an argument to the interpreter. Alternatively, an executable file can be an object file in the ELF (Executable and Linkable File) format. In that case, the kernel populates the address space according to the layout specified in the file.

Each executable file format has its own activator. At a certain point in the `do_execve` function, the kernel invokes every activator, passing the file to execute and other parameters to it. Each activator first checks the initial bytes of the file to see if they match a special constant, called the magic number, which is assigned to a specific file format. If the activator detects a match for its file format, it proceeds to load the program from the file. Otherwise, it returns an error code and the kernel tries the next activator.

In this thesis we will take a closer look at the activator for the ELF64 file format. The initial layout of the address space is described using **sections** and **segments**.

A section contains data that serves the same or similar purpose and has the same protection attributes. For instance, the `.data` section contains statically allocated, initialized, read-write data for use by the program, and the `.text` section contains read-only program code.

A segment comprises a number of sections and has a specified virtual address at which it is to be loaded. The sections comprising the segment are loaded into memory starting at the segment's initial address and continuing, one section after another.

The image activator for the ELF64 file format is a function called `exec_elf64_imgact`. We are only interested in the part which scans the table of program headers (which describe individual segments) and loads appropriate segments into memory, which is shown below.

```
/*
 * hdr - ELF header, contains general information about the file
 * phdr - array of program headers, which describe segments
 */
for (i = 0; i < hdr->e_phnum; i++) { /* For each segment... */
    switch (phdr[i].p_type) {
    case PT_LOAD:   /* Loadable segment */
        if (phdr[i].p_memsz == 0)
            /* Nothing to load */
            break;

        /* Read the segment's protection attributes */
        prot = elf64_trans_prot(phdr[i].p_flags);
        /* Load the segment into the address space */
        elf64_load_section(imgp, phdr[i].p_offset,
            (caddr_t)(uintptr_t)phdr[i].p_vaddr,
            phdr[i].p_memsz, phdr[i].p_filesz, prot);

        break;
    /*
     * Other cases omitted
     */
    default:
        break;
    }
}
```

Listing 9: `kern/imgact_elf.c`: fragment of the `exec_elf64_imgact` function mapping program segments into the address space of the calling process

The confusingly named `elf64_load_section` function loads individual segments into the virtual address space. Most of its code handles the case when the size of a segment in memory is larger than the size of the segment in the executable file and the end of the mapping backed by the file is not on a page boundary. The mapping of the segment is done inside the `elf64_map_insert` function.

The `elf64_map_insert` function is responsible for mapping a range of offsets inside a `vm_object` to a range of virtual addresses in a `vm_map`. It also contains code to handle edge cases, but the essence has been distilled in the code listing below.

```
static int
elf64_map_insert(struct image_params *imgp, vm_map_t map, vm_object_t object,
    vm_ooffset_t offset, vm_offset_t start, vm_offset_t end, vm_prot_t prot,
    int cow)
{
    struct sf_buf *sf;
    vm_offset_t off;
    vm_size_t sz;
    int error, locked, rv;

    /*
     * Handle edge cases where the start/end virtual addresses
     * are not page-aligned.
     */
    if (start != trunc_page(start)) {
        /* ... */
    }
    if (end != round_page(end)) {
        /* ... */
    }
    if ((offset & PAGE_MASK) != 0) {
        /*
         * Another edge case: (offset % PAGE_SIZE) != (start % PAGE_SIZE)
         * This means that we have to copy the data.
         */
         /* ... */
    } else {
        /*
         * This is the most common code path: no edge cases,
         * simply map a range of pages in the vm_object to a range
         * of virtual pages in the address space
         */
        vm_object_reference(object);
        rv = vm_map_fixed(map, object, offset, start, end - start,
            prot, VM_PROT_ALL, cow | MAP_CHECK_EXCL);
    }
    return (KERN_SUCCESS);
}
```

Listing 10: `kern/imgact_elf.c`: fragment of the `exec_elf64_imgact` function mapping program segments into the address space of the calling process

3.3 Handling page faults

Once a process begins execution, it is almost inevitable that at some point it will attempt to access an page of virtual memory that isn't currently resident. When

it happens, control is passed to the kernel, which attempts to bring the page into physical memory and resume the process which caused the page fault.

The main function responsible for handling page faults is `vm_fault_hold`. Its arguments include the `vm_map` of the faulting process, the virtual address that the process attempted to access, and what type of access was attempted (read or write).

It begins by looking up the `vm_map_entry` corresponding to the address. It then follows the chain of shadow objects starting at the entry's backing object, looking for an object that contains the needed page. If an object with the page is found and the fault is a write fault (the process attempted to write to the page), the copied is copied to the first object in the chain. If it is just a read fault, the page can be simply mapped into the address space, without the need to copy or move it. The following code listing shows the algorithm in more detail.

```
int
vm_fault_hold(vm_map_t map, vm_offset_t vaddr, vm_prot_t fault_type,
    int fault_flags, vm_page_t *m_hold)
{
    /*
     * The faultstate structure records the parts of the
     * current state of the handler which are most commonly
     * passed to helper functions. Its fields are:
     * map: vm_map of the faulting process
     * first_object, first_index, first_m and
     * object, index, m:
     *   object, offset into the object and page corresponding
     *   to faulting address withing the first and current
     *   vm_object in the chain
     * entry: vm_map_entry containing the faulting page
     */
    struct faultstate fs;
    vm_object_t next_object;
    vm_prot_t prot;
    boolean_t wired;
    int behind, ahead;

RetryFault:;

    /*
     * Find the backing store object and offset into it to begin the
     * search.
     */
    fs.map = map;
    vm_map_lookup(&fs.map, vaddr, fault_type |
        VM_PROT_FAULT_LOOKUP, &fs.entry, &fs.first_object,
        &fs.first_pindex, &prot, &wired);
```

```
fs.first_m = NULL;

/*
 * Set the initial (object, offset) pair that will be used
 * to query objects for the page we're looking for
 */
fs.object = fs.first_object;
fs.pindex = fs.first_pindex;
while (TRUE) {

    /*
     * See if page is resident
     */
    fs.m = vm_page_lookup(fs.object, fs.pindex);
    if (fs.m != NULL) {
        /*
         * Block and retry if the page is busy.
         */
        if (vm_page_busied(fs.m)) {
            if (fs.m == vm_page_lookup(fs.object,
                fs.pindex)) {
                vm_page_sleep_if_busy(fs.m, "vmpfw");
            }
            goto RetryFault;
        }

        /*
         * Mark page busy for other processes.
         */
        vm_page_xbusy(fs.m);
        /* We found the page to be copied to fs.m */
        break; /* Break to PAGE HAS BEEN FOUND. */
    }

    /*
     * Page is not resident. If the pager might contain the page
     * or this is the beginning of the search, allocate a new
     * page. (Default objects are zero-fill, so there is no real
     * pager for them.)
     */
    if (fs.object->type != OBJT_DEFAULT ||
        fs.object == fs.first_object) {

        /*
         * Allocate a new page for this object/offset pair.
         */
        fs.m = vm_page_alloc(fs.object, fs.pindex,
            alloc_req);
        if (fs.m == NULL) {
```

```
            /* No pages available. Block and retry. */
            vm_waitpfault();
            goto RetryFault;
        }
    }

    /*
     * Call the pager to retrieve the page if there is a chance
     * that the pager has it, and potentially retrieve additional
     * pages at the same time.
     */
    if (fs.object->type != OBJT_DEFAULT) {

        /*
         * Calculate how many surrounding pages we should ask the
         * pager for, determining the values of ahead and behind.
         */


        /* Get the pages */
        rv = vm_pager_get_pages(fs.object, &fs.m, 1,
            &behind, &ahead);
        if (rv == VM_PAGER_OK) {
            /* Now we have the page to move to first_page */
            break; /* Break to PAGE HAS BEEN FOUND. */
        }

        /*
         * If an I/O error occurred or the requested page was
         * outside the range of the pager, clean up and return
         * an error.
         */
        if (rv == VM_PAGER_ERROR || rv == VM_PAGER_BAD) {
            vm_page_free(fs.m);
            fs.m = NULL;
            unlock_and_deallocate(&fs);
            return (rv == VM_PAGER_ERROR ? KERN_FAILURE :
                KERN_PROTECTION_FAILURE);
        }

        /*
         * The requested page does not exist at this object/
         * offset. Remove the invalid page from the object,
         * waking up anyone waiting for it, and continue on to
         * the next object.  However, if this is the top-level
         * object, we must leave the busy page in place to
         * prevent another process from rushing past us, and
         * inserting the page in that object at the same time
         * that we are.
```

```
             */
            if (fs.object != fs.first_object) {
                vm_page_free(fs.m);
                fs.m = NULL;
            }
        }

        /*
         * We get here if the object has default pager
         * or the pager doesn't have the page.
         */
        if (fs.object == fs.first_object)
            fs.first_m = fs.m;

        /* Move on to the next object. */
        next_object = fs.object->backing_object;
        if (next_object == NULL) {
            /*
             * If there's no object left, fill the page in the top
             * object with zeros.
             */
            if (fs.object != fs.first_object) {
                fs.object = fs.first_object;
                fs.pindex = fs.first_pindex;
                fs.m = fs.first_m;
            }
            fs.first_m = NULL;

            /* Zero the page and mark it valid. */
            pmap_zero_page(fs.m);
            fs.m->valid = VM_PAGE_BITS_ALL;

            break;  /* Break to PAGE HAS BEEN FOUND. */
        }

        /* Adjust object offset. */
        fs.pindex += OFF_TO_IDX(fs.object->backing_object_offset);
        fs.object = next_object;
    }

    /*
     * PAGE HAS BEEN FOUND. [Loop invariant still holds -- the object lock
     * is held.]
     */

    /*
     * If the page is being written, but isn't already owned by the
     * top-level object, we have to copy it into a new page owned by the
     * top-level object.
```

```
     */
    if (fs.object != fs.first_object) {
        /*
         * We only really need to copy if we want to write it.
         */
        if ((fault_type & (VM_PROT_COPY | VM_PROT_WRITE)) != 0) {
            pmap_copy_page(fs.m, fs.first_m);
            fs.first_m->valid = VM_PAGE_BITS_ALL;
            /* We no longer need the old page or object. */
            release_page(&fs);
            /*
             * Only use the new page below...
             */
            fs.object = fs.first_object;
            fs.pindex = fs.first_pindex;
            fs.m = fs.first_m;
        } else {
            prot &= ~VM_PROT_WRITE;
        }
    }

    /* Put this page into the physical map. */
    pmap_enter(fs.map->pmap, vaddr, fs.m, prot,
        fault_type | (wired ? PMAP_ENTER_WIRED : 0), 0);

    return (KERN_SUCCESS);
}
```

Listing 11: `vm/vm_fault.c`: Simplified implementation of the `vm_fault_hold` function