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

1 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

✠

PWit: Komentarz na boku

Cahir: Komentarz w treści

✠

Cahir: Komentarz na boku

Author: Odpowiedź na komentarz w treści

✠

Author: Odpowiedź na komentarz na boku

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.

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 (1), as it allows the user to select any major version of the source tree, while (2) 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, (2) 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_mematttr_t mematttr;   /* 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_mematttr_t mematttr;
```

`mematttr` 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

```
struct pmap {
    struct mtx          pm_mtx;
    pd_entry_t          *pm_pdir;       /* KVA of page directory */
    TAILQ_HEAD(,pv_chunk)   pm_pvchunk; /* list of mappings in pmap */
    cpuset_t            pm_active;      /* active on cpus */
    struct pmap_statistics  pm_stats;   /* pmap statistics */
    LIST_ENTRY(pmap)    pm_list;        /* List of all pmaps */
    struct vm_radix     pm_root;        /* spare page table pages */
    vm_page_t           pm_ptdpg[NPGPTD];
};
```

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