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

15 lipca 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

1 Introduction **9**

1.1 What is virtual memory? 9

1.2 The FreeBSD Operating System 10

1.2.1 Browsing the source code 11

2 An overview of the FreeBSD VM subsystem **13**

2.1 `vmspace` 13

2.2 `vm_map` 14

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.