FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Wiederverwendung des RAM Inhalts nach Neustarts

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Manuel Wiesinger
Matrikelnummer 0825632

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Anton Ertl

Wien, 27. Juni 2016

Manuel Wiesinger

Anton Ertl

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Recycling RAM content after reboots

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Manuel Wiesinger
Registration Number 0825632

to the Faculty of Informatics

at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Anton Ertl

Vienna, 27th June, 2016

Manuel Wiesinger

Anton Ertl

Erklärung zur Verfassung der Arbeit

Manuel Wiesinger

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 27. Juni 2016

Manuel Wiesinger

Acknowledgements

First of all, I would like to thank my supervisor Anton Ertl, for his time and ideas, and especially for patiently waiting for me to start writing. My gratitude extends to the people who contributed to the NetBSD project, and to all other free software projects essential for this thesis. Furthermore, I would like to thank my friend and colleague Benjamin Maurer, who was working on his thesis next to me every single day. Last but not least, I would like to thank my dear parents, grandparents, sister and in particular Luna for their support, patience and care.

Kurzfassung

Moderne Computer verwenden DRAM-Module als Hauptspeicher. Darin gespeicherte Daten überstehen mit hoher Wahrscheinlichkeit einen kurzen Stromausfall. So ist es möglich, diese Speicherinhalte beim Wiederhochfahren des Betriebssystems weiter zu nutzen. Zweckdienlich könnte es sein, den Bootprozess zu beschleunigen indem man benötigte Speicherblöcke nicht neu vom Sekundärspeicher lädt. In der vorliegenden Arbeit wird anhand der aktuellen Literatur die Umsetzbarkeit in dem Betriebssystem NetBSD® behandelt. Abschließend, werden Probleme, die in der Anwendung auftreten analysiert.

NetBSD ist eine eingetragene Marke der NetBSD Foundation, Inc.

Abstract

Modern computers use DRAM modules as main memory. Data stored in them is likely to persist even in case of a short power outage. So it is possible, to reuse these memory contents, when rebooting the operating system. It could be useful to improve the boot time by recycling what what is already available in memory, instead of loading it from secondary storage. The present thesis investigates on the basis of the current literature the feasibility in the NetBSD ® operating system. Finally, the arising problems, which occur in practice are analysed.

NetBSD is a registered trademark of The NetBSD Foundation, Inc.

Contents

CHAPTER 1

Introduction

DRAM (Dynamic Random Access Memory) is an integrated circuit, using capacitors for information storage. Each of these charged or uncharged capacitors represents a single bit. As time passes, the charge of the capacitors leaks. In order to avoid information loss, a capacitor needs *refreshments* in regular intervals, typically after some milliseconds. In other words, this means a recharge. [Hal+09] The maximum time, data is available in DRAM, without refreshments, is between 2.5 and 35 seconds, depending on the hardware. Manufacturers are interested in increasing the retention time, since longer retention time means less frequent refresh rates for the capacitors, resulting in lower power consumption. [Hal+09, p. 4] The effective retention time depends heavily on the manufacturer and type of the RAM module. SRAM (Static Random Access Memory) modules have data retention times up to several seconds [Sko02] too. However, this thesis does not consider them any further, since typical desktop computers or servers use DRAM rather than SRAM, as primary memory.

Due to the retention times of data in memory, a considerable chance exists, [Hal+09, p.7] that the whole memory footprint of the operating system, running before an unexpected reboot (e.g. caused by an operating system crash, by pressing the machine's reset button, or a short power outage), is available.

Modern operating systems cache data from disk in memory, in order to make data operations more efficient. Commonly, the operating system's kernel maintains a table of addresses, that represent the data stored on disk.

1.1 Motivation and problem statement

So far, the effect of data remanence in RAM has only been used in security research and forensic analysis [Hal+09]. This thesis suggests a different use of data artefacts. The idea is to take advantage of data remanence, after an unexpected reboot, by reusing what is left of the operating system's disk cache, to save expensive disk accesses during boot time. Other uses, such as preventing data loss or safely skipping file system checks, by analysing file system operations which were not written to disk before a crash, are also conceivable.

When the operating system requests a block from a file system, it computes a checksum of the block and stores it in a dedicated log. While booting again, the kernel searches for the log and marks it as part of its memory, to protect it from being overwritten. When the operating system requests a block from disk, it checks if the block is already available in memory. To guarantee that a block is undamaged, it calculates the checksum of the block and verifies it against the checksum of a corresponding block in the log.

Since the file system blocks, requested at boot time, are always similar (kernel, daemons, etc., are always loaded) this approach may save a considerable amount of disk accesses and thus speed up the boot process. Even though Solid State Drives (SSDs), with dramatically improved disk access times (and thus also boot times), have become popular and inexpensive recently, there are valid reasons why hard disks will still be employed in the future:

- Recent research showed that SSDs have a significant higher rate of uncorrectable errors, than classical hard disks [SLM16]. Another drawback is that SSDs have a limited amount of write cycles. Hence, system administrators may prefer to use hard disks for systems where reliability is more important than performance.
- They are less expensive and can reach performance, comparable to SSDs, when suitable caching techniques are used [Kou13].

Another interesting reason to analyse memory relics, is the search for file system commands which have not been written back to backing storage. This can prevent data loss by retrieving data, that cannot be restored by file system checking tools, such as `fsck(8)`. In an ideal case, file system checking can even be skipped safely.

The basic idea is taken from Chen et al. [Che+96]. Their approach is to safe the RAM contents by using battery-backed RAM, dump it to a swap partition during reboot and analyse the content with a user space tool. The aim of this work is to evaluate, if their idea can be implemented on modern hardware, without requiring battery-backed RAM modules.

Research question:
Can reuse of buffer cache be implemented in an existing operating system, with reasonable effort?

1.1.1 Notes to the reader

In this thesis it is often useful to refer to NetBSD manual pages. Manual pages are referred, as usual in the UNIX community, with its corresponding section in parenthesis.

For Example, `mmap(2)`, denotes the manual page of the `mmap` system call, which is part of section two of the NetBSD manual pages.

Where it is useful to mention C-structures, they are denoted like this: `struct inode`, for the inode structure.

Manual pages used in this thesis are available online at
`http://netbsd.gw.com/cgi-bin/man-cgi?++NetBSD-7.0`.

CHAPTER 2

Theoretical background

This chapter gives some background information on the concepts which are used in this thesis. Unless otherwise stated it is based on Stallings' *Operating Systems: Internals and Design Principles* [Sta09]. It briefly recapitulates basic concepts, implemented in practically every modern general purpose operating system, which are fundamental for the understanding of memory recycling.

2.1 Disk cache

In general computer memory is the more expensive, the faster it is. Greater storage capacity means lower cost and slower access time. To meet reasonable cost and performance requirements a memory hierarchy is employed. Typically a memory layer is used as cache for the layer below, to minimize more expensive data transfers from the level below. Commonly both, data and instructions used one after the other, tend to cluster. This is known as the *locality of reference* [as cited in Sta09]. Various caching strategies are applied (e.g. least-recently-used or last-recently-used), to improve performance while saving costs. Caching mechanisms are also employed in other components of computers, such as the CPU (L1, L2 cache) or a network interface controller.

Following this principle the primary memory is used as *cache memory* of the secondary memory. In a typical general purpose desktop computer that means that the RAM is used as cache for hard disk access.

2.2 Virtual Memory

The term virtual memory refers to techniques which make it possible to run processes independently of the machine's physical memory. This helps to conserve memory usage. Processes do not operate on the *physical memory*, but use a *virtual address space*.

Typically, the process perceives much more memory that is physically available on the system. Furthermore, processes do not need to be fully resident in memory, only the parts which are about to be executed need to be kept in memory. This makes it also possible to execute programs whose size is larger than the actual configuration of physical memory, or to run a huge number of (possibly very large) processes at the same time. The latter can also increase the overall performance, since a process can be quicker ready for execution (in ready state), because only the next context needs to be in memory.

The total amount of memory that is kept in memory, is referred to as *memory footprint*.

The both following examples, illustrate the advantages of virtual memory.

Example 1: On most hardware platforms code that is executed needs to be in memory, because it is loaded from memory to the CPU. Consider a system, which has to run many large processes at once. Some of them are not busy and in sleeping state. While they are waiting the other processes on the system can use the memory, which would be allocated for the sleeping processes.

Example 2: A program operates on a very large file (like a video file or disk image), which is much larger than the physical memory. Let's assume the program wants to modify only meta-information at the beginning of the file. With virtual memory only the beginning blocks, which contain the meta data have to be loaded to main memory.

2.3 Paging

In order to make use of virtual memory, the memory has to be divided into, fixed-sized parts. These parts are commonly referred to as *pages*. Each process is assigned a *page table*, which keeps track of pages in use. A page table typically contains a virtual address (relative to the program's starting address) and an offset, to calculate the physical address. An entry of a page table is usually referred to as *PTE* (Page Table Entry), in this thesis the term *map entry* is used, in accordance with the terminology used in the NetBSD documentation. The translation of virtual to physical address is done by the hardware. The CPU accesses the page table and does the translation. Almost all modern hardware architectures provide integrated co-processsors for this purpose, the *Memory Management Unit* (MMU).

An important feature of paging is that pages can be *paged out* That means list of pages, or single pages can be stored on disk, when they are currently not used. A page that has been modified and not yet been written to its backing storage, is called *dirty* page.

Paging also helps to prevent wasting memory. If associated pages (like a process) are not required any more, they are removed, leaving a "hole". This hole can be easily be filled with parts of another process. This distribution of pages, into non-continuous areas is called *fragmentation*. Modern operating systems try to reduce fragmentation, to ease handling of non-virtual, kernel internal pages as well as huge pages. If a system is unable to carry out its actual tasks, because it is only busy with loading pages, one speaks of *thrashing*.

CHAPTER 3

Implementation in NetBSD

This thesis uses NetBSD, [1] because it has a clearly structured kernel architecture, with excellent documentation and high code quality. The NetBSD development puts great emphasis on portability, currently 58 hardware platforms are supported. Complexity of the kernel is reduced in comparison to other modern operating systems. Hence, it is relatively simple to read and understand how things work. NetBSD took over huge parts of code from BSD4.4, which is described in *The design and implementation of the 4.4BSD operating system* [McK96].

The intention of this chapter is to make the reader understand what is happening, when a file is accessed on NetBSD. The virtual memory implementation is explained in great detail, so the suggestions in chapter 4 can be retraced at source code level.

3.1 File System

On UNIX file systems administer files with index nodes (*inodes*). An inode stores meta information (such as access control information or the creation time), as well as the blocks that contain the managed file.

NetBSD's default file system is *FFS2* (short for Berkley Fast File System). On BSD4.4 the file systems were hierarchically divided into two layers. The upper layer called UFS (Unix File System) defines *what* (especially meta-data) to store. The lower layer FFS2, defines *how* to store the data on disk. [2] This hierarchically approach is still used in BSD systems. UFS is also the default file system of other BSD operating systems, such as FreeBSD and OpenBSD.

[1] `https://www.netbsd.org`

[2] Often the term "UFS2" is used for UFS with FFS2. Instead of FFS2, another back-end, the Log-structured File System (LFS) can be used. Similarly, the term LFS is used for UFS with LFS as backend. In this thesis the term FFS2 is used, because it is used in most NetBSD documentation.

To manage files on disk, the UFS layer defines the `struct inode`. Inodes are described in `inode(5)`. They contain information about the file system entry it describes. Each file or directory is represented by an inode. Inodes store either the whole file (in case of very small files), or pointers to *indirect blocks*. UFS supports up to tree levels of indirect blocks (*triple indirect blocks*), which make it possible to store (with respect to the used block size) approximately two petabyte in a single file. [Fre]

All files sytems have in common, that they are implemented using the `buffercache(9)`, which is described in section 3.3 of this chapter.

3.2 The virtual file system

The virtual file system (`vfs(9)`) is an interface layer to abstract operations on file systems in the kernel. It defines what a file system diver must provide to the kernel (unless the file system implementation uses `puffs(3)`, which is not discussed here). In the kernel an on-disk file is represented by a `struct vnode` which is completely independent of the backing file system. Operations performed on a vnode are defined in `vnodeops(9)`. Most of these *vnode operations* are closely tied to a specific system call, e.g. the vnode operation `VOP_OPEN` is, as the name suggests, used by the `open(2)` system call.

The vnode operations should usually never be called directly. There are high level convenience functions, which abstract some complexity and make development easier, they are explained in `vnsubr(9)`. For instance, when opening a file using `open(2)`, it is necessary to lock the vnode for each element of the path (each directory), or when a file does not exist yet, `VOP_CREATE(9)` must be used instead, of `VOP_OPEN(9)`. `vn_open(9)` does this job for the developer.

3.3 File system buffer

The file system buffer is the core of interaction between primary and secondary memory (normally RAM and disk). All transactions of the virtual file system layer are done via them. The file system buffer in NetBSD is called `buffercache(9)`. Figure 3.1 gives an idea how the NetBSD file system buffer interacts with the virtual memory system and the device driver.

When a file system reads a block, it calls the `bread(9)` function of the `buffercache(9)` interface. This function allocates memory via the kernel's memory pool (and thus finally via UVM) and reads it from the disk, by invoking the disk driver. It gets the logical block address of the file system block as argument. The logical block number is calculated from the physical block number, stored in the file's inode. The inode contains a *shift value* (`fs_bshift`), which can be used to calculate the logical block address (*LBA*) from the physical addresses. The LBA must be unique per file system.

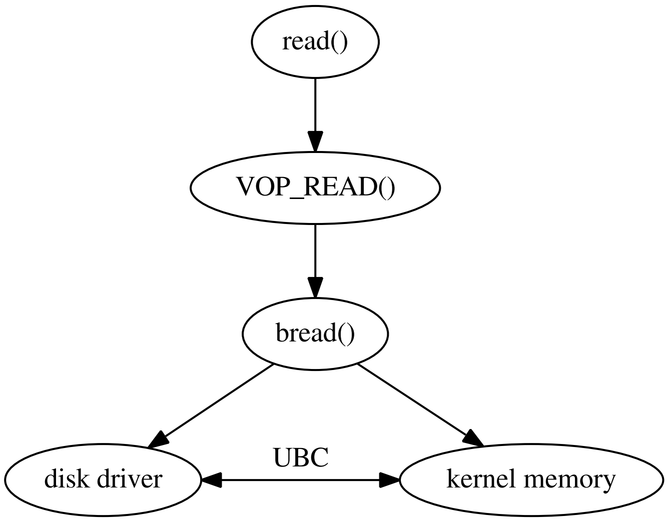

Figure 3.1: Simplified depiction of what happens, when the `read(2)` system call is invoked.

The buffer type that is used by buffercache is `buf_t`, which is implemented in `struct buf`. This structure is the general purpose structure for buffered I/O in NetBSD.

3.4 Virtual memory

This section is a short summary of [Cra98] and [CP99]. It gives a brief overview of virtual memory system and explains the parts which are necessary to understand this thesis.

The virtual memory implementation employed in NetBSD is called UVM [3]. It is divided into two layers, a machine dependant layer (called the *pmap* layer) and a machine independent one (called the *amap* layer).

One of UVM's major job is to manage lists of continuous pages (referred to as *UVM objects*). These UVM objects are represented by `struct uvm_object` and represent portions of memory belonging together (e.g. a file, or the memory footprint of a process). Unlike the old BSD VM system, UVM has no cache itself. The caches of other layers, such as the virtual file system (see section 3.2) are used, to avoid unnecessary copies. Were necessary copies are performed using the copy-on-write principle.

Like in traditional BSD implementations, a *pager* is responsible for transferring data from backing storage (commonly the file system, or swap space) to memory. Each UVM object has a pointer to a set of pager functions.

UVM is used for every process that runs on the system. The kernel has its own private pool, called `uvm_km(9)`, which is reserved for it. Figure 3.3 depicts the subdivisions of

[3]UVM is a name, not an acronym.

UVM.

3.4.1 Initialization

Physical memory configuration

Before UVM can be initialized, physical memory needs to be configured. UVM provides a single interface to configure systems with contiguous and non-contiguous memory, called `MACHINE_NEW_NONCONTIG` (MNN for short). Non-contiguous memory means that physical memory cannot be addressed by contiguous address numbers. That also means that more than one physical memory module is supported. Modern architectures are typically non-contiguous memory architectures.

The MNN interface creates a statical array, with an entry for each physical memory module. When it is necessary to find the corresponding *virtual* page for a *physical* page, it has to be searched. This is can by the `vm_physseg_find()` function. Note that this is reverse of the usual case, but common in certain situations.

Each element of this array contains a pointer to a list of `struct vm_pages` (the structure describing physical pages in the virtual memory implementation).

UVM Initialization

The UVM system is started by calling the `uvm_init(9)` function, this call is part of the kernel's main function. As of NetBSD 7.0, it is called after the console has been set up. Until that point, all addresses used by the kernel are direct physical addresses. The system's MMU is still disabled at that point.

The initialization process has the following steps, which are executed in the following order:

- initialize global UVM data structures
- initialize the page subsystem. That includes allocation of `struct vm_page` structures for each physical page`struct vm_physseg`.
- allocation of the kernel's private memory pool.
- the kernel's virtual memory data structures (memory map, submaps and the memory object of the kernel (`uvm_km`) are allocated and initialized.
- The machine-dependant `pmap_init()` function is called. It gives the used module the chance to do machine-specific initializations, before UVM is operational.
- The kernel's memory allocator is initialized
- The pager daemons are started. An initial pool of free anonymous memory is allocated.

After this steps UVM is fully operational.

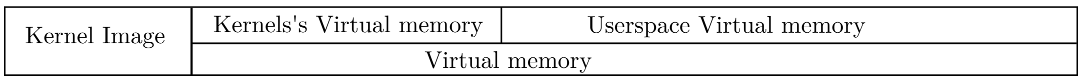

Figure 3.2: Memory partitioning after the UVM has been initialized.

3.4.2 The machine-independent layer

One of the design goals of UVM was the strict separation of machine-dependent and machine-independent layer. The amap layer is the part of UVM and responsible for the interaction with other kernel subsystems. Following the design goals of NetBSD, it contains the main logic, so that the machine-dependent code can be kept at a minimum. Figure 3.3 gives an coarse overview of the abstraction layers.

vitual memory space Each process, maintains a `struct vmspace`, containing a pointer to the virtual memory address map (`struct vm_map`) and some statistical data for that process (or the kernel). [4]

memory map Virtual memory maps are management structures for areas of virtual memory and the link to the machine-independent part of the subsystem (i.e. the pmap layer). Technically memory maps are implemented as `struct vm_map` and contain a pointer to a linked list of `vm_map_enty` structures, which represent the actual memory mappings. `vm_map_enty` structures contain the starting and ending point of UVM objects and some attributes, like access control information.

UVM objects are represented by `struct uvm_object` and are the virtual memory system's representation of a file, an available (zero filled) memory area or a mapped device. In a nutshell an UVM object (or memory object) is a list of pages (including meta data), which are accessed via the *pager operations* (see below). The design goal of memory objects is that they are used by other kernel subsystems, (especially the I/O subsystem). A good example, with regard to the purpose of this thesis it the fact that vnodes contain a pointer to the respective memory object. When using UBC (section 3.4.4) a `struct vnode` is interchangeable with a `struct uvm_object`.

page A page is the representation of a physical page. There is a `struct vm_page` for each physical page.

pager A pager is a set of operations to access the backing storage of a memory page. For instance, when a page, that is backed by a vnode (and thus by a file), the vnode specific paging operations, and hence finally the file system operations are invoked. The UVM pagers write *clusters* of pages back to backing store. That means, if

[4]Note that [CP99] says that it contains a pointer to the machine-dependent mapping (i.e the process' pmap). When UVM was first released in NetBSD 1.4., this pointer is one layer below, in `struct vm_map`.

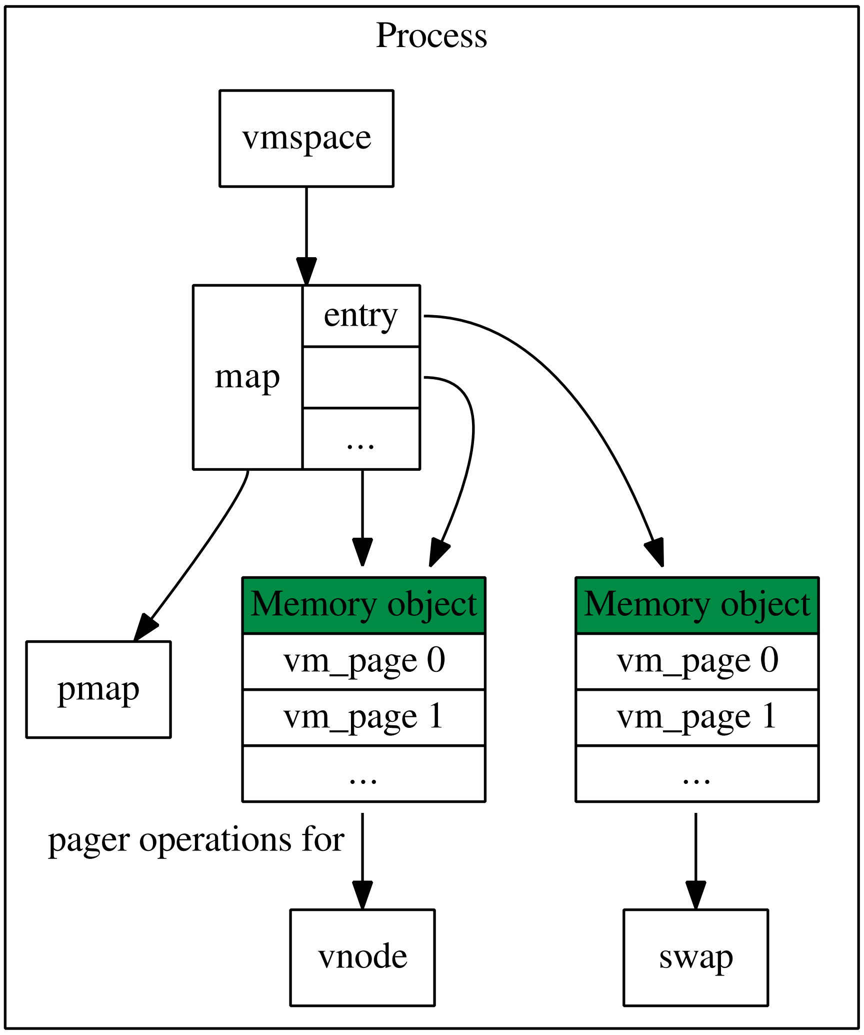

Figure 3.3: The abstractions of UVM. Each map entry points to a memory object, which contains a pointer to the specific pager operations.

several dirty pages of one object are detected, they are grouped and flushed at once. This saves some overhead.

The pager is also responsible for one of the core functionalities of a virtual memory implementation. It is responsible for allocating memory (e.g. when a page fault occurs), this hides memory allocation from processes.

Currently there are tree pagers in NetBSD. responsible

Swap (or aobj) pager
Maps anonymous memory from swap space, to memory.

Vnode pager
: Maps vnodes to user memory. Backed by the filesystem drivers.

Device pager
: Maps device memory to user space processes. A typical examle are graphical framebuffers.

pager daemon The pager daemon is a kernel level daemon, that does housekeeping work for in memory pages. It writes dirty pages back to backing store and pages out pages when memory becomes scarce.

3.4.3 The machine-dependent layer

The physical mapping layer (*pmap*) layer is the interface to the hardware memory management unit (MMU) (See section 2.3). It's job is to manage physical address maps, which the virtual addresses map to. That means it creates, removes and modifies physical pages.

The pmap layer is also responsible for maintaining hardware specific information, about the system's memory, which is necessary to run UVM. It can be seen as a 'driver' for the systems MMU and memory.

UVM's upper layer (i.e. the machine independent) has no knowledge about the hardware-specific part, so the interface is the same on every hardware platform for the machine-independent layer.

UVM was designed to be compatible with the pmap module of the old BSD VM system, as well as other pmap modules such as FreeBSD's performance optimized pmap module. Due this modular approach UVM is very portable. As of version 7.0, NetBSD provides pmap modules for 66 architectures.

Information (like page size, backing file, etc.) about physical pages can be gained by the `pmap(1)` tool.

Implementation

The pmap module keeps track of all `struct pmap` structures that are referenced. In most implementations this is done by simply keeping a list. Each `strcut vm_page` has the physical address stored in it (`phys_addr`).

3.4.4 Unified Buffer Cache

UBC (Unified Buffer Cache) is a subsystem of the NetBSD kernel, which was written to improve the collaboration between the I/O and the virtual memory subsystem. The improvement are summarized in [Sil00]. It can be used by system calls and file systems to implement direct transfer from device to the page cache. System calls and file systems (such as NTFS) that do not support UBC, use a special area of the kernel's memory

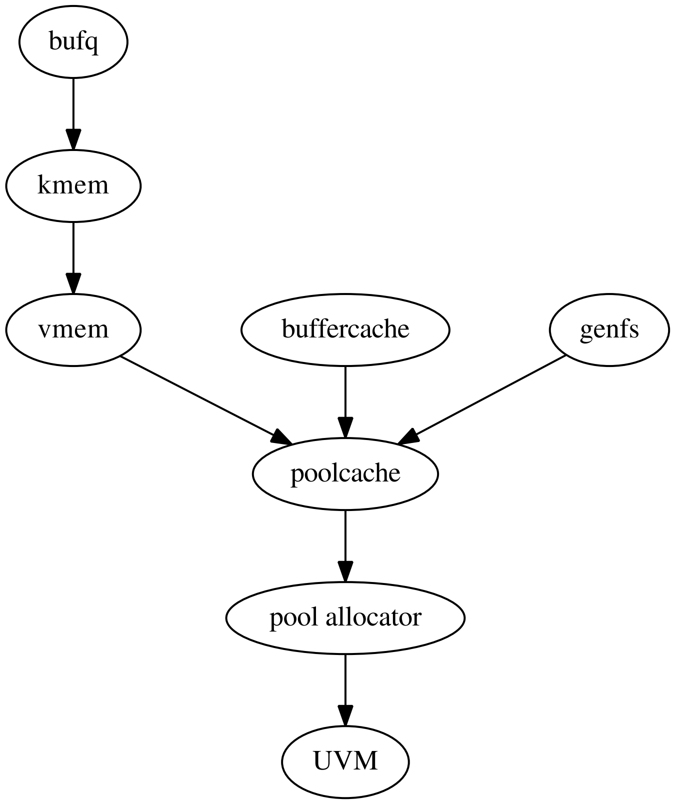

Figure 3.4: The memory allocators

as file system cache (i.e. the buffercache, see section 3.3). Without UBC the pages are cached twice. Once in the buffercache and once in the UVM object (technically the backing vnode). The communication between those layers, without using UBC, is done by a copy. Removing this unnecessary copy is the main feature of UBC. The system calls that support UBC are mainly `read(2)`, `write(2)` and `mmap(2)`.

UBC created two new vnode operations `VOP_GETPAGES()` and `VOP_PUTPAGES()`, which have to be provided by the file system drivers. They get and put pages from disk to page, as their names suggests. UBC brought a dramatic improvement of performance, when accessing the disk.

3.5 Interaction of I/O and Virtual Memory

After the VM system is initialized, all memory used by the kernel and the user space is done via the UVM interface and thus by the virtual memory subsystem. Of course direct memory access is still possible, but useful, in edge cases, if at all. Figure 3.4 gives an idea of how the different memory allocators of the NetBSD kernel depend on each other.

CHAPTER 4

Suggested Implementation

This part presents the concept of a possible implementation. First in general, then in NetBSD.

The concept of reusing memory can be divided into three parts.

- First, the blocks need to be prepared, so that they can be found, and verified when booting the system again.
- Second, they must be searched and indexed, when the system is starting again.
- Then, they must be reused.

Obviously, least recyclable blocks are overwritten by new ones, when they are searched as soon as feasible during booting.

To be able to add portions of memory, that were found and identified as reusable, the virtual memory system has to be initialized first. The idea is to *sequentially* search the memory, during the initialization of the virtual memory system. That makes implementation less complex, because at that stage, all necessary functions are already available. Implementing this feature outside of the virtual memory subsystem (such as in form of a driver), would be much more effort (and consequently, probably not worth it), because the structures and helping functions are neither available nor intended to be accessed outside of the VM subsystem. In particular the locking mechanisms cannot cleanly be accessed by other subsystems.

Checking whether found pages are reusable or not can be done, by maintaining a checksum for each page or range of pages. It can be either stored in the block itself or in a special log, subsequently referred to as *page log*.

When a file system block is requested, the available, reusable blocks are searched. If the block is already in memory, it can either be copied (to reduce memory fragmentation) or referred to in the freshly initialized virtual memory structures.

After the system has been fully initialized, i.e all daemons are started and the whole system is in a, state similar to before the reboot, the probability that something can be reused decreases. Hence, it is sensible to disable the lookup in the page log at some point. However, if a single lookup, in the table of reusable blocks does not harm performance, there is nothing wrong in doing it.

4.1 Integration into NetBSD

This section proposes a possible implementation in NetBSD. Everything, needed for the understanding of this section was introduced in chapter 3.

First, it is suggested what steps could be taken in order to make pages recyclable. Then it is explained how they can be reused and found during booting after a reboot.

4.1.1 Prepare pages for reuse

When a dirty (i.e. modified) block is written back to backing store a checksum is generated and added to the page log. To make it possible, to find the log, when sequentially searching for it, a hard coded *magic number* is inserted as the first variable of the structure. This way the memory portion can simply be casted to the structure of the page log.

The structure for a page log entry could look as following.

```
struct log_entry {
    dev_t device;
    daddr_t lba;
    struct *pages;
    hash_t checksum;
}
```

where `hash_t` represents the type of the employed hash function. The checksum has to be calculated from the device identifier, the LBA and the referred pages.

Since the logical block numbers (section 3.3) are unique per file system and `struc buf` stores an id of the associated device `dev_t b_dev`, a block can be identified uniquely.

4.1.2 Reuse of blocks

Assuming the system is booting, after a short power outage and the page log has been found, so the kernel knows about possibly recyclable pages. The system is about to start user space daemons.

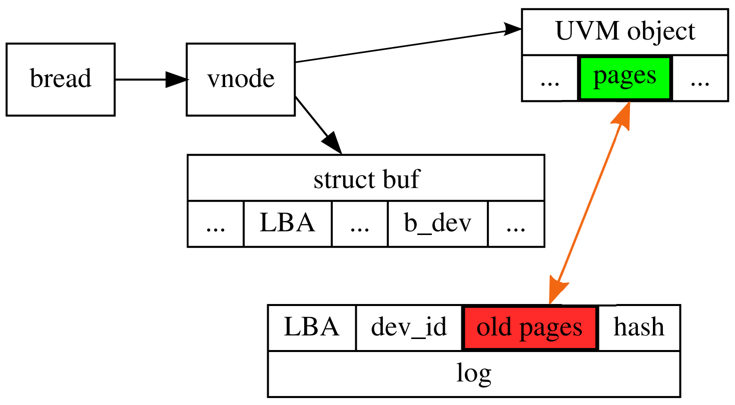

Figure 4.1: The recycling of pages. The orange arrow represents the transfer operation from old, to new pages. The black ones references. The LBA and the device id in the buffer are compared with the respective value in the log.

Figure 4.1 depicts the suggested process, when a block is requested by calling `bread(9)`. A `vnode` is requested for the requested file. `struct vnode` contains a `struct uvm_object`, which contains the list of virtual pages. From the `struct buf` of the vnode, we know the LBA and the device identifier, which is requested, so the block can be identified uniquely. Now a lookup is done in the log, for the block and see if the checksum is still valid. If so, the block is copied (or referenced) to the new UVM object.

Great care has to be taken, since the pages referred in the log are virtual pages. However, usually the addresses should be identical after a reboot, since the initialization process it static. However, currently there is no mechanism that guarantees that. For a practical, well performing implementation, this has to be ensured by UVM.

4.1.3 Searching for blocks

It is best to search the blocks as early as possible in the system, but impractical until the initialization of the virtual memory subsystem, has reached a certain point. A reasonable approach is to implement the search in the `uvm_int(9)` function (described in section 3.4.1). The best time for the search seems to be, right after the kernel's memory allocator has been initialized. At that point it is already possible to use the buffercache interface. So a sequential search for the page log in raw memory can be started.

CHAPTER 5

Practical feasibility

To investigate practical feasibility of memory recycling, two small tools were developed. They show that it is difficult on today's hardware to get memory artefacts, after an unexpected reboot.

5.1 Prototype

To determine how many blocks can be recycled after an unexpected reboot, two helper programs were written. `fillmem`, to fill the memory with random data and `readmem` to search for it after performing a reboot. The necessary meta data is stored in a file, so it can be read again after rebooting. To verify the blocks a SHA256 checksum is calculated. The tools have been tested on NetBSD and should work on other BSD operating systems as well, but not on Linux.

In order to carry out tests, memory protection mechanisms, which shuffle the order of variables, have to be disabled. On NetBSD Address Space Layout Randomization (ASLR) is enabled by default (`security(7)`). It can be disabled by setting the `sysctl(8)` variable `security.pax.aslr.enabled` to zero.

fillmem

This tool fills a given amount of memory with random patterns and meta data. It takes two options `-f` sets the file to safe meta data and `-s`, sets the amount of memory that is to be filled. Common size modifiers such as k for kilo, `M` for Megabytes are accepted.

Then it calculates how many pages fit into the given memory and fills it with the following structure. The structure is exactly as big as a page (including the metadata).

The magic number is at the top of the structure, to make it possible to search it in memory. The checksum variable stores the SHA256 checksum of the data value. This

```
struct block {
    uint64_t magicno;
    unsigned char data[BLOCKSIZE];
    unsigned char hash[SHA256_DIGEST_LENGTH];
};
```

Figure 5.1: The C structure with with the memory is filled.

ensures, that the block really is unchanged. In a real implementation more suitable hashing functions, such as CRC or Adler-32, which take less time, since they are fault tolerant and faster, are to be considered.

After filling the given size of memory with random patterns, the meta data consisting of the number of written blocks and the used magic number is written to a file, to be read by `readmem`. The user is then requested to press the reboot button of the machine.

readmem

`readmem` is called with two options. The `-f` flag specifies the used meta data file and the `-i` flag the image file to be searched. This can either be a memory dump, or the `mem(9)` device (`/dev/mem`). [1] After searching the given image file, statistics are printed.

5.2 Test Results

To investigate the amount of recyclable data, `fillmem` and `readmem` where run on several physical machines, typical desktop machines (with DDR1 and DDR2 RAM), as well as a Raspberry Pi, revision 1[2]. Unfortunately, no memory artefacts occurred on physical machines and very little on virtual ones.

Physical Machines

Even when POST memory checks are disabled and fast boot features are enabled in BIOS. That is in line with observations described by Gruhn and Müller [GM]

The reason for that depends heavily on the used hardware and RAM type. The following circumstances prevent data recycling on many systems.

- [Hal+09] suspect, that systems supporting Error-correcting code (ECC) bring RAM to a known state.

[1]Note: As of NetBSD 7.0, invoking `readmem` with `/dev/mem` as target file, may crash the kernel or freeze the system.

[2]`https://www.raspberrypi.org`

- Some systems implement the TCG Platform Reset Attack Mitigation Specification [Tcg], which obliges to reset the memory, in order to make physical attacks more difficult.
- [GM] observed, that different systems with DDR 3 RAM do not preserve memory. They argue that the reason for this are noise effects which occur because of the high density of the capacitors. However they also observed that systems with DDR1 and DDR2 memory preserve data at room temperature.

Virtual Machines

When performing tests with KVM[3] guests on a Linux host, there were no memory relics, when forcing a reset of the running operating system. However, an interesting exception occurred: One test found 1% of the data was found and reusable. The reason for this is that KVM allocates the memory, only when a guest needs it. It can be easily shown that a forced shutdown frees the memory. Hence, it is up to the memory management of the host operating system which memory is allocated for the guest. It can be assumed, that other virtualization environments handle memory in a similar manner. For this reason data recycling on virtual machines is unlikely to be useful.

[3] `www.linux-kvm.org`

CHAPTER 6

Conclusions and Future Work

This thesis intended to give a proof of concept for the reuse of memory artefacts after a reboot. Investigating the source code of the NetBSD operating system indicated that it is feasible to adapt operating systems, so they can reuse memory contents. However, experiments showed that it is impossible without extra effort (such as modifying the firmware) to run the prototype on common consumer hardware, because memory is cleared during the booting process.

That said, artefacts remaining in memory are not only interesting for security research. Recycling them can dramatically improve loading times, especially in large scale infrastructure, where boot times can be considerably long. Of course, operating systems and hardware need to be designed to make use of it. Many operating systems are able to dump memory to a swap partition, when they hit a kernel panic. It is possible to extract file system operations and to rescue critical data, that may be lost or damaged otherwise. Still, one has to carefully weight up the advantages and the disadvantages. Security features, such as Address Space Layout Randomization (ASLR) do not play together easily with memory recycling.

List of Figures

Bibliography

[Che+96] Peter M. Chen et al. „The Rio File Cache: Surviving Operating System Crashes“. In: *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS VII. Cambridge, Massachusetts, USA: ACM, 1996, pp. 74–83. ISBN: 0-89791-767-7. DOI: 10.1145/237090.237154. URL: http://doi.acm.org/10.1145/237090.237154.

[CP99] Charles D. Cranor and Gurudatta M. Parulkar. „The UVM Virtual Memory System“. In: *Proceedings of the USENIX Annual Technical Conference*. Monterey, California, USA: USENIX Association, 1999.

[Cra98] Charles D. Cranor. „Design and Implementation of the UVM Virtual Memory System“. PhD thesis. Sever Institute of Washington University, 1998.

[Fre] *Frequently Asked Questions for FreeBSD 9.X and 10.X*. URL: https://www.freebsd.org/doc/faq/book.html#idp58525648 (visited on Mar. 17, 2016).

[GM] Michael Gruhn and Tilo Müller. „On the Practicability of Cold Boot Attacks“. In:

[Hal+09] J. Alex Halderman et al. „Lest We Remember: Cold-boot Attacks on Encryption Keys“. In: vol. 52. 5. New York, New York, USA: ACM, May 2009, pp. 91–98. DOI: 10.1145/1506409.1506429. URL: http://doi.acm.org/10.1145/1506409.1506429.

[Kou13] Petros Koutoupis. *Advanced Hard Drive Caching Techniques*. 2013. URL: http://www.linuxjournal.com/content/advanced-hard-drive-caching-techniques.

[McK96] Marshall McKusick. *The design and implementation of the 4.4BSD operating system*. Reading, Mass: Addison-Wesley, 1996. ISBN: 0-201-54979-4.

[Sil00] Chuck Silvers. „UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD“. In: *USENIX Annual Technical Conference, FREENIX Track*. San Diego, California, USA: USENIX, 2000.

[Sko02] Sergei Skorobogatov. *Low temperature data remanence in static RAM.* Tech. rep. UCAM-CL-TR-536. University of Cambridge, Computer Laboratory, June 2002. URL: `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-536.pdf`.

[SLM16] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. „Flash Reliability in Production: The Expected and the Unexpected“. In: *14th USENIX Conference on File and Storage Technologies (FAST 16).* Santa Clara, California, USA: USENIX Association, Feb. 2016, pp. 67–80. ISBN: 978-1-931971-28-7. URL: `https://www.usenix.org/conference/fast16/technical-sessions/presentation/schroeder`.

[Sta09] William Stallings. *Operating Systems: Internals and Design Principles, Global Edition* -. 6. edition. Pearson Education Limited, 2009.

[Tcg] *Trusted Computing Group, Incorporated, TCG Platform Reset Attack Mitigation Specification, Specification Version 1.00, Revision 1.00.* 2008. URL: `https://www.trustedcomputinggroup.org/pc-client-work-group-platform-reset-attack-mitigation-specification-version-1-0/` (visited on May 30, 2016).