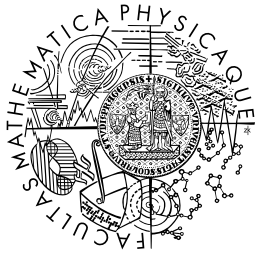

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Jakub Matěna

VMA merging in Linux

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	RNDr. Vlastimil Babka, Ph.D.
Study programme:	Computer science
Study branch:	Software systems

Prague 2022

I declare that I carried this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In Prague date 21. 7. 2022 signature of the author

I am grateful to my supervisor, consultant and authors of the vim command in Linux.

Title: VMA merging in Linux

Author: Bc. Jakub Matěna

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Vlastimil Babka, Ph.D., SUSE

Consultant: prof. Ing. Petr Tůma, Dr., D3S

Abstract: This thesis deals with virtual memory management in the Linux kernel. Files or devices can be mapped into virtual memory using the mmap() system call, which is also capable of creating so-called anonymous mappings. Those do not map anything and serve only as a memory allocation method that can specify memory protection and flags for the created anonymous virtual memory area (VMA). A mapping can be merged or split depending on its flags, memory protection, location in the virtual memory and other parameters. However, in some cases even when two VMAs have practically identical parameters, they may not merge successfully because of implementation limitations. This thesis concentrates on anonymous VMAs and their page offset and anon_vma parameters, which in some cases prevent a merge in the current mainline kernel. It is demonstrated that in most of the cases with reasonable amount of effort the obstacles can be removed and the merges happen.

Keywords: Linux Kernel Memory management VMA C

Název práce: Spojování VMA regionů v operačním systému Linux

Autor: Bc. Jakub Matěna

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: RNDr. Vlastimil Babka, Ph.D., SUSE

Konzultant: prof. Ing. Petr Tůma, Dr., D3S

Abstrakt: Tato práce se zaobírá se správou virtuální paměti v Linuxovém kernelu. Soubory nebo zařízení mohou být mapována do virtuální paměti pomocí systémového volání mmap(), které je také schopné vytvořit takzvané anonymní mapování. Ty nic nemapují a slouží pouze jako metoda alokace paměti, která může specifikovat ochranu paměti a příznaky pro vytvořenou anonymní oblast virtuální paměti (VMA). Mapování lze sloučit nebo rozdělit v závislosti na jeho příznacích, ochraně paměti, umístění ve virtuální paměti a dalších parametrech. Nicméně v některých případech, i když dvě VMA mají prakticky identické parametry, nemůžou být úspěšně sloučeny kvůli omezení implementace. Tato práce se zaměřuje na anonymní VMA a jejich parametry page offset a anon_vma, které v některých případech brání sloučení v současném mainline kernelu. Je demonstrováno, že ve většině případů lze s přiměřeným úsilím odstranit překážky a dojde k sloučení.

Klíčová slova: Linux Kernel Memory management VMA C

Contents

Introduction

The Linux kernel manages virtual address space of each process using Virtual memory area (VMA) structures to represent mapped ranges. Each area can map a part of a file or be anonymous which makes it basically just a block of usable memory.

The memory areas can be created, removed, resized, split or merged as a result of `mmap()`, `munmap()`, `mremap()`, `mprotect()` and other syscalls. Ideally, each virtually contiguous range mapping a single linear file segment, or anonymous memory, with single set of attributes such as read-write-protection, would be always represented with a single VMA. However, for anonymous memory ranges, the current implementation may not merge their VMA structures after non-contiguous ranges become contiguous due to implementation limitations. Thus in processes utilizing `mremap()` heavily this limitation results in extra memory and CPU overhead due to the need to manage larger amounts of VMA structures. An unaware application can even gradually create lots of mappings and eventually exhaust the mapping limit. Also due to another limitation in `mremap()` implementation, it's impossible to `mremap()` areas that should be contiguous, but consist of multiple VMA structures, in a single syscall.

This thesis is about low-level details of the kernel's memory management. The target audience are kernel developers and daring students. The reader should already be familiar with the virtual memory concepts including the process memory structure, the address translation and paging, parallelism including locking, system calls and operating systems in general. Understanding the Linux kernel is no easy task because the documentation is scarce and incomplete. Also, getting the community to thoroughly review the code is sometimes close to impossible. The only way leading to a success are therefore incremental changes, debugging and testing to slowly acquire the knowledge and experience needed to get to the bottom of it, however deep the bottom is.

Document structure

This chapter is a short summary of what you can find in individual chapters. If you already have a good grasp of the kernel, feel free to skip Analysis and continue directly to Design and other chapters.

Analysis

This chapter briefly introduces Linux as an OS and its development process. Then, it dives into the development tools necessary for the analysis of memory management from the memory mapping point of view. This contains description of individual functions and structures, as well as more high level point of view of the whole system.

Design

The design describes changes to the kernel that were necessary to address the current VMA merge shortcomings. This includes the design of individual helper functions, as well as a complex description of the individual patches, including `vma_merge()` refactor, page offset update, different `anon_vma` merge, `mremap()` expansion merge and tracing.

Testing

Testing describes all tests including merge tests, analytic programs and other specialized tests. The tests are added as an attachment to this thesis.

Results

This chapter presents all results of this thesis. Specifically, it provides an overview of the hardware and software used for the measurements, speed and successful merge count results, and of course the final summarization of advantages and limitations.

Problem specification

When merging two VMAs, a number of conditions have to be met. Most importantly, the second area has to follow the first without an empty space between them. This is checked using the sizes of the areas and their start addresses. If the memory is a mapping of a file then also the page offsets in the file have to follow. For example, if the first area contains, in order, pages number 4, 5 and 6, then the second area has to follow with page number 7 and possibly others. And of course both areas must be mapping the same file. Unfortunately, there is no exception for anonymous mappings regarding the page offsets and so this condition is required even when the mapping has nothing to do with files.

The page offsets of anonymous mappings are initially set equal to the virtual address of the mapping with page bits removed. This is done as the `mmap()` call creates the area. The offset is left unchanged when the area is moved using the `mremap()` call after at least one page in the mapping has already been faulted. Later, when a merge is attempted between this and another VMA, the page offset may not follow up, because it was not updated after the move operation. Such a VMA cannot merge with basically any other VMA, which causes unnecessary and long term mapping fragmentation.

Additionally, because VMAs can be shared among processes and can be created and removed pretty easily, there is a structure called Anonymous VMA structure (AV), which is referenced from a VMA and also from each page. When merging two VMAs, their references must point to the same AV. A VMA can map pages belonging to different AVs only in certain cases, for details see What prevents merges (page 35). Therefore, if the VMAs point to different AVs, the merge also fails. Unfortunately, in many cases two VMAs that would otherwise be perfect match for a merge do not share the same AV and cannot be merged.

The last identified opportunity for a merge is when the `mremap()` call expands an already existing mapping and this growth causes the adjacent gap to disappear. In this case, a merge attempt with the next VMA might succeed, but unfortunately, the current kernel omits the necessary call to `vma_merge()` in such a situation.

Implementation brief

The solution is theoretically quite simple. Update the page offset during `mremap()`, modify the `anon_vma` check and call `vma_merge()` after the mapping expansion. Unfortunately nothing in kernel is easy. The page offset is not only saved in the VMA itself, but also in page structs that represent physical pages, specifically its `page→index` field. Updating the page offset therefore implies updating index in all the page structs. This means locating all of them by the means of page tables, which can be walked using the page walk mechanism already implemented in the kernel. Another problem is that these pages might be shared e.g. due to the Copy on write (COW) mechanism, in which case changing the page offset is not possible as it would change for all the VMAs mapping it and that would affect other processes than the one actually calling `mremap()`. Sharing can be detected by looking at the AV relations and physical pages themselves.

Merging two VMAs linked to two different AVs is also problematic and again requires changes to the page structures and so the pages must not be shared as well. The AV pointer is stored in `page→mapping` and can be updated using an already existing function. We again use the page walk to get to the page itself.

The only relatively easy part is the last change, which just means to call `vma_merge()` after a mapping is expanded.

1. Analysis

This chapter presents the Linux kernel, its development process, the coding style and the debugging tools. It also describes the high-level memory management in the kernel and also the individual functions and mechanism related to the mappings that must be understood before the patches can be explained. This whole chapter is based on kernel version `v5.18`. Skilled kernel memory management developers can jump directly to Design (page 35).

1.1 Linux

Today's Linux distributions are open-source Unix-like operating systems consisting of the Linux kernel and supporting software libraries and packaging systems. The resulting product is therefore a combination. Although there are big software differences between individual distributions, all the main distributions use the same Linux kernel. Any change or upgrade to the kernel means an upgrade to all such distributions and can potentially have a huge impact.

1.1.1 Linux kernel

The kernel was initially released on 5th of October 1991 and nowadays is the fastest growing operating system in the world, supporting various devices including embedded, mobile, personal, servers and even supercomputers.

1.1.2 Development process

The most up to date public version of the Linux Kernel can be found in the git repository created by the Linux Foundation[1]. If anyone wants to commit to the kernel, he/she has to make changes against some specific version as is described in the Documentation/process folder and especially in the submitting-patches.rst file located in the git repository itself. As a result, all changes in the kernel can be seen as individual commits, at least for several last years. This is very useful when trying to understand the changes because it is easy to find the corresponding changes in other parts of the kernel and usually a commit message explaining the idea behind it. Git itself was created by Linus basically for kernel development and is therefore embedded deeply in the development process.

When the work is done, the commits are converted into patches and are then send as emails to the Linux mailing list[2] and also directly to the people responsible for the part of the kernel changed. There is a tree system of maintainers and if the patch is to get merged into the mainline, it has to go through one branch from the bottom to top where Linus performs final check. When a given maintainer agrees with the patch, he/she just adds his sign-off signature to the email message/patch. All of the approved patches are then merged with the mainline to create a testing

[1] Linux repository - `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/`

[2] Richman [2021]

version. Apart from several test frameworks, most of the testing is done by just running the test version on as many devices as possible. It is made possible by quite a big community of kernel testers that voluntarily install and use this unstable version. This is mostly regression testing to ensure everything that worked before still works, stability is required much more than new features.

All of the development communication is done by sending emails, which nowadays seems as a really obsolete development tool. At the first glance at thousand of emails daily with patches and other discussions about kernel, it really is overwhelming. If they were not well-formatted git patches that were being send around, most probably everyone would have gotten lost by now. The trick is also to use the right set of filters, spam blockers and if everyone follows the rules it just works. The fact the kernel is a functioning and widely used OS can serve as a proof.

1.1.3 Tools used for development

This is a list of tools that can be used for easier development.

vim

Vim is a modified version of vi, which stands for visual. It is a command line text editor and can edit all kinds of documents including source files in this case. It can be used as an editor to make simple changes, but vim can do much more than that. The editor is packed with commands that can be used to copy or move text, search patterns of text, replace text, traverse multiple files and most importantly create macros that are basically saved series of command that can do several simple tasks at once to achieve a complex change. When using several tabs with several files opened, you might want to consider using view command as a read-only variant of vim. Otherwise navigating to function definition might try to open a file for writing which is already opened for writing, which results in an error.

grep

Grep is a shell command used for finding patterns in its input, which is usually a file. This is probably the most useful tool to find definitions of functions, macros and variables. Additionally it can be used to find the correct function if you just guess part of the name or something that might appear in a comment nearby.

git

Git is a fast and distributed version management system. It is usually composed of upstream and local repositories. Local repositories are located at the development devices of individual contributors, upstream repository is usually located at a server either on premise or supplied by a third party.

A git repository consists of individual commits as small and compact change blocks and these commits together build up a graph with branches representing

individual features. Branches are usually merged back into the mainline branch called master.

Local repository is fully functioning copy and can use wide scheme of git commands. The history can be modified using rebase or cherry pick commands. This enables any user to achieve a clear and obvious step by step development history.

Concurrent modifications can also create conflicts when trying to merge two versions together. This is usually problematic but with git, it can in many cases be fully automatic and even if manual resolving of conflicts is needed, it is often quite straight forward.

Third party providers like GitHub or GitLab provide web frontends to manage the project and also offer many additional features that can enhance the development process like CI/CD.

ctags

This tool can process source files and create tags for individual elements like functions and properties. This allows for a quick search of definitions for function calls, which is used usually by client tools like vim (`ctrl+]` to find definition and `ctrl+t` to return). This upgrades vim to very useful source file viewer and dramatically decreases the time and effort needed to find definitions of helper functions used in the kernel. Unfortunately, it is not always accurate and sometimes fails to find the definition completely. Accuracy is problematic when multiple definitions for the same function are provided for different architectures or configurations. If the user does not pay attention, he might not realize the definition is valid only for the specific architecture. Another problem is with macros, where the localization of definitions sometimes fails completely and the only solution is using grep.

1.2 Running the kernel

This section contains the whole process from getting the source files to building, running and debugging a modified kernel.

1.2.1 Getting source files

The most recent source files can be found in Linus's git repository[3]. However for normal uses the stable releases are ideal. This basically means to not use release candidates versions ending with the `rc` suffix.

1.2.2 Building

Before building the source files, the kernel build has to be configured properly depending on your hardware and preferences, producing the `.config` file. This can

[3]torvalds/linux git repository - `https://github.com/torvalds/linux/commits/master`

be quite hard for an inexperienced user but there are tools to do this automatically like genkernel for Gentoo. The generated configuration is not as optimized as if it was created by a skilled administrator, nonetheless the result is a functional configuration. The compilation itself can take some time, although on faster devices it should not be too long. Additionally, thanks to make, recompilation of partially modified kernel only recompiles the modified files.

1.2.3 Running the kernel in a virtual machine

When making changes to the kernel code and especially in the early phases, it is not a good idea to run the kernel on physical HW, as the code is most probably not completely flawless. Therefore it is best to run the kernel first virtually using `qemu` or its wrapper `virtme`. This will quickly reveal badly malfunctioning versions without threatening any real system. It is also quicker and overall more convenient to start a virtualized system. However, after the code seems to work more or less, it can be tested in a normal environment doing to get rid of the remaining bugs.

1.2.4 Debugging the Kernel

From my point of view, the documentation seems mostly incomplete or hidden very deep in the git commit messages. Some parts of code are quite well commented and understandable, but other are rather unintuitive. Most of the compact documentation on the internet is sometimes obsolete and sometimes it is hard to distinguish if it is obsolete or just incomplete. When you finally seem to understand what you need to do and you create a patch that builds and even solves the problem, you are left with debugging the side effects causing bug reports in kernel or other problems. Judging by my experience, it is usually very hard to pinpoint the source of the problem and create an appropriate fix. And when you do, you cannot be sure if it really helped, because replicating the problem is often not possible and you just have to wait if the problem reoccurs again or if it was truly resolved by the fix. One thing that can help is to add warnings or bug calls for any unexpected state of things, an overview can be seen in Table 1.1.

When you want to run the modified kernel on a hardware device, you have to choose an appropriate Linux distribution that supports custom kernel versions. Usually it is possible for almost every distribution, however, some distributions make it very easy as Gentoo did for me.

Name	Description
BUG	Usually calls some special or invalid instruction. This interrupts the execution of the system call and the kernel is not responding properly to future requests, although the OS is not completely frozen. Specific behaviour depends on the architecture and the kernel version.
WARN	Basically prints similar information as BUG but does not interfere with execution. Although the same problem that triggered the warning might cause other problems in kernel.
BUG_ON	Internally calls BUG.
VM_WARN_ON	Internally calls WARN_ON if CONFIG_DEBUG_VM is set.
VM_BUG_ON	Internally calls BUG_ON if CONFIG_DEBUG_VM is set.
VM_BUG_ON_PAGE	Dumps the given page and internally calls BUG if CONFIG_DEBUG_VM is set.
VM_BUG_ON_FOLIO	Dumps the given folio and internally calls BUG if CONFIG_DEBUG_VM is set.
VM_BUG_ON_VMA	Dumps the given VMA and internally calls BUG if CONFIG_DEBUG_VM is set.
VM_BUG_ON_MM	Dumps the given memory descriptor and internally calls BUG if CONFIG_DEBUG_VM is set.

Table 1.1: Overview of bugs and warnings calls.

1.2.5 Kernel coding style

This subsection is abbreviated as all the important and usually up-to-date information is located directly in the Linux kernel's git repository. According to GitLab's statistics, the kernel and related scripts in the repository are mostly written in C (98.5%), Assembler takes up about 1 %, makefiles and other scripts take the rest. Coding style is more or less standard for C and is quite extensively described in the Documentation/process folder in the files coding-style.rst and 4.Coding.rst.

The first and obvious difference against probably any modern code is the use of gotos, which is not recommended in modern coding style. But their use in kernel is quite reasonable and prevents problems when changes are necessary. It is often used for clean up at the end of a function and because of some possible errors there might be several exit points. All of these can be directed to the right code at the end of the function via mentioned gotos. This saves unnecessary code duplication and ensures clean up is done thoroughly in all cases. There might be several labels for gotos to enable for gradual clean up depending on how deep in the function the thread gets before encountering an error. This is of course not the only use for gotos. A more complex application might be the context switch and other tricks in the deepest levels of the kernel.

The rest of the coding style is quite normal and can be easily found in the git repository as mentioned before. However, not all parts follow the standard as

mostly the older code remains in the state as it was written years ago as there is no good reason to modify code just because of coding style.

1.3 Memory management in the Linux kernel

The main role of the memory management is to distribute the available memory among the running processes and collect the unused memory, either indicated by a process or from a finished or a killed process.

Today's devices usually have gigabytes of memory. Although in some cases, like embedded computers, the actual amount can be a lot lower. Memory is split into addressable units which are usually several bytes, e.g. 64-bit processors have natively 64-bit addressable units, although in some cases it might be possible to operate with smaller units like 32-bit.

1.3.1 Virtual memory

All of the memory resides in an address space, but because it would be impractical for all processes to use the same address space and it might cause interference between the processes, virtual address spaces are used. This means that every process has its own address space and individual addresses are translated by the CPU when they are accessed. This address space is called virtual address space and the memory is therefore called virtual memory.

In the address space, the memory is divided into areas, which usually share some parameters like access rights and other flags, which depend on what is actually stored in the memory.

1.3.2 Program memory anatomy

As mentioned before, program's memory is virtualized and addresses are translated into their physical counterparts using page tables.

The address space of each process can be split into several main parts. These parts are the kernel space, the stack, the memory mappings including mapped files and mapped libraries, the heap and the uninitialized and initialized static variables and of course the binary image of the process's code.

The individual memory mappings of a process can be examined by reading the `/proc/$pid/maps` file and also `/proc/$pid/smaps`, which is even more detailed. The `$pid` can also be replaced with `self` for some use cases.

Kernel space

The kernel space is used to store kernel's data and code and is part of the virtual address space, but it maps to the same physical memory in all processes. The kernel space is not directly accessible by a process, such an attempt will result in a page fault. Instead, a system call must be made to ask the kernel to do something for the calling process. This ensures that potentially malfunctioning or even insidious software will not be able to cause serious harm. On the other hand, user space, which contains all other parts of the address space, can be directly accessed by the process and usually changed as needed.

Stack

The stack is used to store local variables and function parameters. When a function call is made, all parameters (except for those passed directly in registers) are pushed to the stack and vice versa when the function returns. Stack is therefore dynamically growing and shrinking as needed. If many function calls are made, the stack might need to grow too large and exceed the maximum stack size. When this happens, the process is killed by the kernel. The limit ensures the stack will fit in the address space and not collide with any neighbouring memory segments. Each thread has its own stack and therefore even with plenty of available memory, multi-threaded applications might take up lots of address space.

Memory mappings

This is the part of process's memory where files can be mapped into virtual memory. This can be done using the `mmap()` system call and is usually used to map extensively used file data. Dynamic libraries are also mapped in this way, and, for this thesis most importantly, anonymous memory mappings. The name itself is quite confusing, because it is not really a mapping but rather just a block of memory available for program's data.

Heap

The heap is used for runtime memory allocation that is not bound to function lifetime and can be used via pointer. Heap can be used to allocate bigger data segments like arrays and data can also be shared with other threads. Allocation is usually done via the malloc call. Heap can be extended using the brk system call if it runs out of space. After several cycles of allocating and freeing memory blocks, there might remain gaps which can be too small to use. This is called heap fragmentation and can be quite problematic in languages without garbage-collection.

Static variables

Static variable are divided into two segments. The first contains the uninitialized static variables, which are stored in an anonymous memory area. The second contains initialized static variables, which are stored in the binary file which is mapped using a private memory mapping, which ensures that potential changes are not reflected in the underlying file and static variables are initialized to values defined in the source code in the next run.

Binary image

The last is the binary image, which stores all the code and also string literals like error messages and so on. It is again private memory mapped binary file, which ensures immutability of the code.

A more detailed explanation can be found in an article at `manybutfinite.com`[4].

[4]Duarte [2009a]

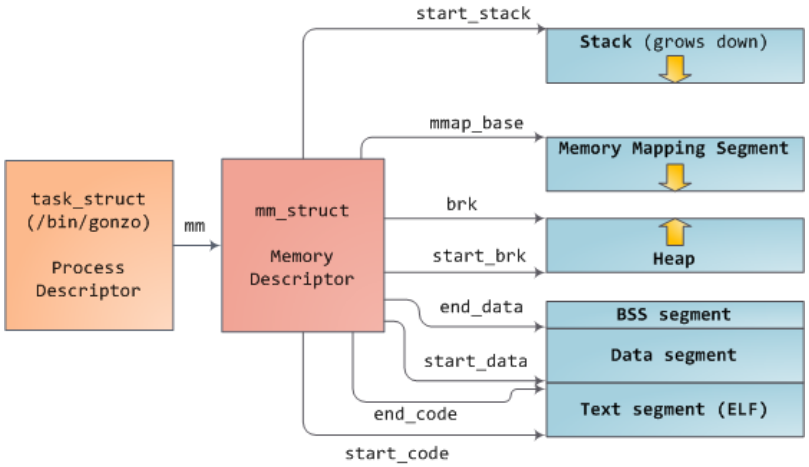

Figure 1.1: Memory descriptor struct. Source: https://manybutfinite.com

1.4 Memory descriptor

In Linux, every process is implemented as an instance of `task_struct` structure. Memory itself is described in `mm_struct`, which is accessible through the `mm` field in the mentioned `task_struct`. `mm_struct` contains the start and end addresses of all the segments mentioned in the previous subsection as can be seen in Figure 1.1, the number of physical memory pages of the process, the size of virtual address space `total_vm` and other data.

Most importantly `mm_struct` contains a pointer to a list of virtual memory areas
`struct vm_area_struct *mmap` and pointer to page tables `pgd_t *pgd`. Also important is the lock struct `rw_semaphore mmap_lock`, which locks the whole structure during `mmap()` or `mremap()` calls.

The stack grows down from `start_stack` and so do memory mapped areas from `mmap_base`. The heap grows up from `start_brk` and has also an end pointer called `brk`. The data segment is restricted by `start_data` and `end_data`, the code block is restricted by `start_code` and `end_code`, these segments do not grow in any direction, because they are immutable. Each of these areas is internally implemented as `vm_area_struct` and can be traversed via `vm_next` as can be seen in Figure 1.2, `vm_area_struct` is in more detail explained in Virtual memory area (page 22).

A more detailed explanation of the memory descriptor and the source of figures is in `How the kernel manages your memory`[5].

[5]Duarte [2009b]

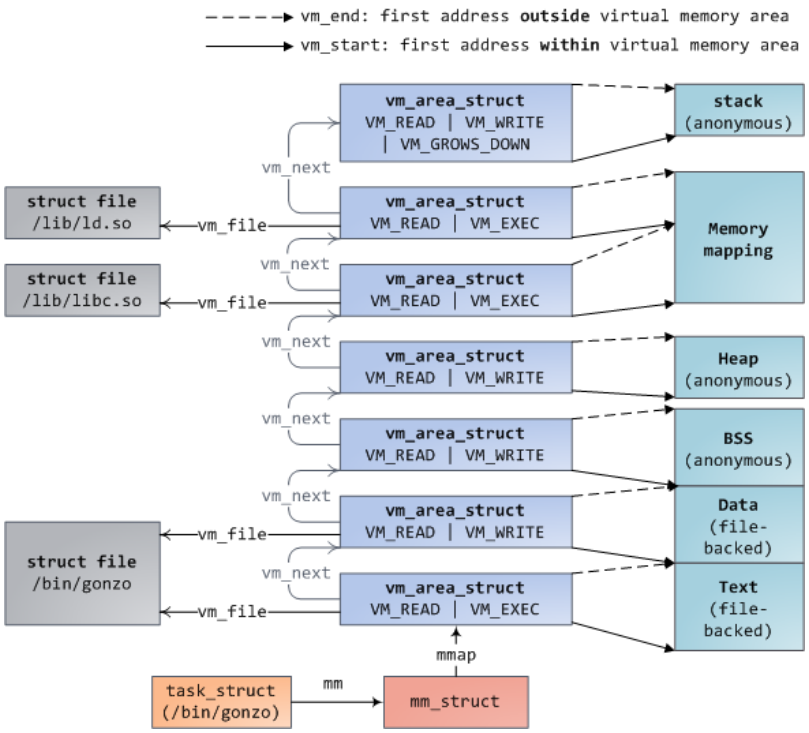

Figure 1.2: Memory descriptor and areas. Source: https://manybutfinite.com

1.5 struct page

This structure is a representation of a physical page, which usually has a size of 4096 bytes as it has been since the very first Linux kernel, and contains a number of various flags and other properties. The following two paragraphs describe only the two most important properties for this thesis, which are index and mapping. Useful functions operating with struct page can be viewed in Table 1.2

The `page→index` property stores an offset of this page within its mapping. Imagine we are mapping a second half of a file containing altogether 10 pages, then the indices of the pages will be 5, 6, 7, 8 and 9. If they all reside in a single VMA then the VMA's `vm_pgoff` (page offset) will be the page offset of the first page that is 5. In the case of the anonymous mappings, that do not map any files, the page offset also exists and is e.g. used during Rmap walk (page 33) to pinpoint the correct VMA from the list of VMAs related to the AV of the page; the AV is accessible via `page→mapping` explained next.

The `page→mapping` property can store a pointer to `anon_vma` and also two flags (`PAGE_MAPPING_ANON` and `PAGE_MAPPING_MOVABLE`) in the lowest two bits. `anon_vma` itself can be access through `page_get_anon_vma()` function if it is assigned to the page. `PAGE_MAPPING_ANON` is set if the page is anonymous and mapped into a user virtual memory area. `PAGE_MAPPING_MOVABLE` is used during migration. A combination of the previous two bits is defined as `PAGE_MAPPING_KSM`, which means the page is a `KSM` shared page and in this case `page→mapping` points to a private structure of the `KSM`. The bits are defined in `include/linux/page-flags.h`. In case of a page shared among several process through the COW mechanism, the mapping property points to the root `anon_vma`.

Function/Constant	Description
`page_mapcount()`	The number of times given page is referenced by a page table.
`page_swapcount()`	The number of references to the given page that are in swapped out state.
`follow_page()`	The function used to locate and return a page based on its VMA and virtual address. It performs an operation similar to a page walk to get the correct `PTE`. When working with the `PTE`, it locks it using `pte_offset_map_lock()` and also waits for possible page migration to end if the `FOLL_MIGRATION` flag is set.
`vm_normal_page()`	Simplified `follow_page` that gets a direct pointer to the `PTE` and therefore does not have to perform a page walk. Also omits some checks compared to `follow_page()`.
`PAGE_SHIFT`	Determines the page size and is used to shift the virtual address to remove page offset bits. It's value for standard 4096 (2^{12}) byte page is 12.

Table 1.2: Overview of some page functions and a constant.

Compound page

Compound page is a group of two or more physically contiguous pages that can in many aspects be treated as one large page. This is used to create transparent huge pages, but can be used for other purposes as well.

The page flags are used to mark a compound page and distinguish between head and tail pages. This differs for 32-bit and 64-bit systems, but the functions `PageCompound()`, `PageHead()` and `PageTail()` create a common interface.

The page offset, which is normally saved in the `page→index`, is saved in the `page→index` of the head page. The tail pages can reuse the `page→index` for other metadata like page order in case of `page[1]` (second compound page). Functions `thp_nr_pages()` or `compound_nr()` for general compound pages can be used to get number of regular pages in the transparent huge/compound page. Compound pages are covered by a LWN article[6].

Folio

Folios are an abstraction level to cover pages and compound pages under a common interface. Otherwise when a function works with a page that is a part of a compound page, it might expect a head or a tail page. Page folios create a common interface for all functions that operate with normal or compound pages. The functions using folios are guaranteed that they will receive a normal page or a head page in the case of compound page and therefore do not have to check

[6]Corbet [2014]

each time. This is quite a new concept[7] and therefore there is still old code not fully using folios.

1.5.1 Page tables, page translation

Entries PGD, P4D, PUD, PMD and PTE are all parts of the virtual address and are levels of indirect addressing. Full names of the shortcuts follow.

- Page global directory (PGD)
- Page level 4 directory (P4D)
- Page upper directory (PUD)
- Page middle directory (PMD)
- Page table entry (PTE)

Page middle directory (pmd)

PMD level is important because migration waiting (see Migration (page 21)) is done at this level and also transparent huge pages take up a whole PMD range and can be accessed using the PMD entry.

Page table entry (pte)

PTE is the lowest level and each PTE entry represents a single page or a subpage of a compound page (see Compound page (page 17)). Page struct is obtainable using the `vm_normal_page()` function. Not every PTE necessarily points to a page struct, as it is possible that the page is swapped out, as can be determined using `is_swap_pte()` in which case the PTE can be converted to a Swap entry (page 20) using `pte_to_swp_entry()` function. Another case when PTE does not point to a page is during migration. If a page is present or not can be determined by checking the PTE entry bits, which is done using the `pte_present()` function. What bits precisely are checked depends on the architecture. Converting PTE to a page struct inside `vm_normal_page()` function is done by shifting the PTE value and treating it as an index into the `mem_map` array, where the page structures are stored[8].

Everything is pretty nicely described in Mel Gorman's book[9] and a more up to date LWN article[10].

[7] Corbet [2021]
[8] see Documentation/admin-guide/kdump/vmcoreinfo.rst → mem_map
[9] Gorman [2004a]
[10] Corbet [2017]

1.5.2 Pagewalk

The pagewalk mechanism walks page tables and applies user defined callbacks. There are multiple entry functions, with the main two being `walk_page_range()` and `walk_page_vma()`. The pagewalk goes recursively through all page table levels (from `pgd` to `pmd` and `pte`). The walk is directed by the `mm_walk` structure, which contains `page_walk_action`, `vm_area_struct`, `mm_walk_ops` and other parameters. The `mm_walk_ops` structure contains, among other things, entries for all page table levels and entries that are called before and after entering a VMA.

walk→action

An action is a mechanism that allows user-specific callbacks to instruct the page-walk core on further actions. `ACTION_CONTINUE` can be used to skip PTE action when PMD action is successful. This is useful when `PMD` represents a transparent huge page and the desired operation is performed at this level and further action on `PTE` does not make sense or would break something. `ACTION_AGAIN` can be used to repeat certain action if previous attempt at the given page table level fails. The default setting is `ACTION_SUBTREE`, which continues with actions on lower page table levels.

walk_pte_range_inner()

This function calls a predefined `pte_entry` operation on each PTE in a for-cycle.

walk_pte_range()

This function calls `pte_offset_map_lock()` or `pte_offset_map()` to protect the whole PMD. Walks through PTEs by calling `walk_pte_range_inner()`.

walk_pmd_range()

This function walks through the PUD range, goes through an array of PMDs and calls `walk_pte_range()` for each. If the PMD holds a `hugepd`, then `walk_hugepd_range()` is called.

1.5.3 Forking and Copy-on-write

When forking a process, fork will not create copies of all the necessary data immediately, but the two processes (parent and child) will share some pages as long as they do not write to them. When the page is to be written for the first time after the fork, then the page is first copied and only afterwards modified. This saves memory that would otherwise be wasted for duplicate copies and also saves some operation time that would be consumed for the copying itself at that time. This is especially important when the first thing the new process does is calling exec to load a new program, which would make all the copying completely useless. More details can be found in an article at Halolinux.us[11]

[11] Frazier [2022]

During `mremap()` or `vma_merge()` `mmap_lock` is locked and therefore forking cannot interfere with either of the mechanisms.

1.5.4 Kernel samepage merging

KSM is a de-duplication mechanism that can merge private anonymous pages with identical content. The KSM daemon periodically scans areas of memory specified by a `madvise` call and if duplicates are found, they are replaced by a single write-protected page. If the page is modified later then it is automatically copied before the modification itself. This mechanism is mainly used in virtualization to prevent duplication between individual instances but can be used anywhere. It can be quite demanding though in terms of processing power, so it should be used with care. To activate the daemon, value `1` has to be saved into the `/sys/kernel/mm/ksm/run` file. Also other important parameters must be set in order for the daemon to work properly, these are `pages_to_scan` and `sleep_millisecs` located in the same folder. The `pages_to_scan` parameter specifies how many pages should be scanned, low value can dramatically increase the time needed for the daemon to find duplicates, the same goes for `sleep_millisecs`, which prolongs waiting time.

More details about KSM can be found in the kernel documentation[12] in the git repository.

1.5.5 Swap

Swapping is the process of removing pages from memory and storing them to a long-term storage device. This is usually done to acquire free memory that is needed for other purposes. A more precise term for swap is page out, which is also sometimes used in the kernel. For the purposes of this work, it is necessary to be aware of swapping as it means some pages might not be present in memory when merge is attempted and such a situation is reflected in the state of the concerned kernel structures, most importantly the swap entry.

Swap entry

Normally, when a page is mapped into virtual memory, there is a PTE pointing to this page. However, when the page is not mapped, the PTE internally holds an instance of a swap entry. The swap entry can be converted into the `swap_info_struct` to get the number of processes referencing it either as mapped or as unmapped (swapped out).

Swap map

The `swap_info_struct` structure has a variable called `swap_map`, which is an array of reference counts of the swapped page. Each cell can hold the maximum value of `SWAP_MAP_MAX`, which is `0x3e`. If the reference count gets higher, a special mechanism using continuation pages is used to extend the maximum possible value. Continuation pages can be allocated several times to support very high reference counts that can be caused by e.g. swapped out KSM pages.

[12] see "Documentation/admin-guide/mm/ksm.rst"

Swap cache mechanism

This mechanism ensures consistency between memory and swap space when a shared mapping is created among several processes and one of the pages is being swapped out. This can happen even for a private mapping as it can be shared using the COW mechanism after a fork call.

When attempting to swap out a page, the page is first added to the swap cache and then an attempt is made to unmap it for all the processes using it. The Rmap walk (page 33) is used for this purpose and it is done by converting the appropriate PTEs to instances of a swap entry.

Until this is done, the swap cache is being used to ensure consistency. This is necessary as the whole process can take some time and in the meantime one of the other processes might try to write to the page. The swap cache ensures that such a write is successfully written to the backing storage.

If the unmap fails for even one process, the whole swapping attempt is cancelled as it would not make any sense because the page has to remain in memory as long as even a single process has it mapped.

Another use for the swap cache is in the reversed situation, when a shared page is swapped out and one process accesses it. In this case the page is swapped back in and the PTE of the faulting process is marked as present. However, the other processes do not get notified and still see this page as swapped out and the swap cache again ensures that possible changes in memory from one process are not overwritten by an outdated version in the backing storage.

More details are available in Mel Gorman's book[13]

kswapd()

The `kswapd()` function is responsible for swapping when running out of memory or already out of memory. `kswapd()` is started as a kernel thread at the boot and continuously monitors memory. The `pageout()` function actually performs the action but kernel first jumps through a series of other functions doing the necessary preparations.

1.5.6 Migration

The migration mechanism can be used to move physical pages between memory of separate NUMA nodes or during memory compaction[14]. If the page is being moved then all PTEs mapping the page are set to a migration entry that signals the operation is in progress. The underlying page struct cannot be accessed at such time.

[13] Gorman [2004b]
[14] Zhang [2021]

1.6 Virtual memory area

The virtual memory area or shortly VMA is at the center of memory management in kernel and also this thesis. As the name states it describes a memory area as a part of the virtual memory of a process. Sometimes VMA is referred to as a mapping. Each VMA has a start and an end address, a file pointer and also a page offset, referring to an offset in a file in number of memory pages. Other attributes include protection rights, a pointer to an address space, the embedded node of a red black tree containing other VMAs, a number of flags and most importantly a pointer to its `anon_vma` and a linked list of `anon_vma_chain` structures as described in the following section Anonymous virtual memory area aka AV (page 22).

1.6.1 Flags

When creating a virtual memory area a number of `mmap()` call flags can be used. The individual flags are described in the `mmap()` manual page, but I will mention some of them here as well. The most important one for us is `MAP_ANONYMOUS`, which creates a mapping that is not backed by a file and basically provides a chunk of memory to the caller. An anonymous mapping can be used in combination with `MAP_PRIVATE` to create a private mapping not shared with other processes. On the contrary `MAP_SHARED` creates a mapping shared among several processes and can be used to share data, communicate etc. Another useful flag is `MAP_FIXED`, which enforces the use of the specified address parameter, that is normally only taken as a hint, any existing mapping standing in the way is discarded. More details about various flags are described in the `mmap()` syscall manual page. Each mapping also has an attribute called `vm_flags` which holds similar flags, but not quite identical. All `vm_flags` flags are defined and described in `include/linux/mm.h`. Conversion between the two sets of flags is defined by the `mmap()` implementation.

1.7 Anonymous virtual memory area aka AV

The `anon_vma` heads a list of private "related" VMAs. These VMAs are related because they usually originated from a single VMA either by forking or splitting. When `mmap()` is called on a range adjacent to an already existing VMA then the `anon_vma` of the already existing VMA can be reused when the new mapping is faulted to enable future merge when both VMAs share flags and other parameters.

The AV structure is important because VMAs come and go as they are split and merged in `mprotect()`, `mremap()` and other syscall calls. The `anon_vma` serves as a relatively stable structure that anonymous pages can point to and `anon_vma` then points to a list of the VMAs. For example when we want to swap out a page, we need to unmap it from all the VMAs mapping it. Which is easily and efficiently done using the pointer to the AV and then by going through its VMAs.

The list (internally a red black tree) of VMAs can be traversed through the `rb_root` variable. More details follow.

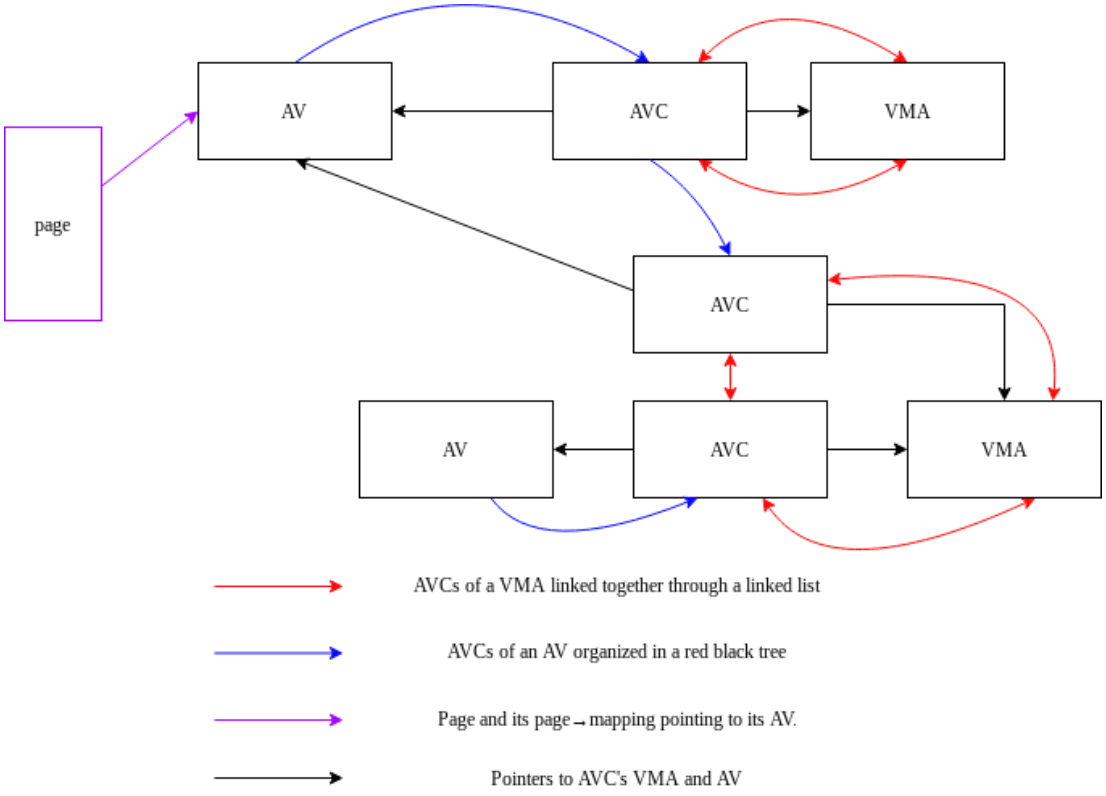

Figure 1.3: AVC overview.

1.7.1 Structure anon_vma_chain

Copy on write can cause an `anon_vma` to become associated with several processes and therefore even several VMAs. Furthermore each child VMA must have its own `anon_vma`. When child process writes to a page then the page is copied (by COW) and the copy is assigned to child's own `anon_vma`. This means that the relation goes both ways, each VMA can be related to several `anon_vma` and each `anon_vma` can be related to several VMAs. This is where AVC (`anon_vma_chain`) comes in.

In figure Figure 1.3 we can see a simple example consisting of two VMAs sharing a common AV, e.g. as a result of a fork. The top line belongs to the parent process and the other items belong to the child. The child VMA also has its own AV. Generally speaking each AVC points to its VMA and to its `anon_vma` (black links). Each VMA can have several AVCs that are linked together through a linked list (red links) starting in `anon_vma_chain` variable in a VMA struct. On the other hand each AV can also have several AVCs that are organized in a red black tree (blue links) with root stored in `rb_root` variable in the AV structure.

Quite good, although slightly outdated, article about anonymous VMAs and their relations called `The case of the overly anonymous anon_vma`[15] can be found at lwn.net.

[15] Corbet [2010]

1.8 The mmap() function

```
void *mmap(void *addr,
           size_t length,
           int prot,
           int flags,
           int fd,
           off_t offset)
```

This is the central syscall of the whole work. As the manual page says the `mmap()` call can be used to map files or devices into memory, but apart from that it can also created mapping which do not map any files and are basically just an allocation of memory or rather the first step in the process. These special mappings are called anonymous and are created using the appropriate flag argument called `MAP_ANONYMOUS`.

For the purposes of this thesis we will not deal with the file or device mappings directly, but sometimes we will come across them, so it is important to keep their existence in mind.

`mmap()` only creates the VMA but does not allocate anything. This a lazy approach betting on the possibility that the memory might not be used or maybe not fully at this time. Also allocating big amounts of memory instantly might seem slow and it can be better to allocate the memory over a period of time as needed. This is closely connected to `anon_vma` allocation, which is not created during the `mmap()` call, but later after the first page fault. A fault can either allocate a new `anon_vma` or it can reuse one already being used by a neighbouring VMA, if it has identical vma policies and other flags. This is checked by `find_mergeable_anon_vma()` function, which is called through series of functions from `handle_pte_fault()`. Reuse is important, because areas with different `anon_vma`s can never merge as merging of two VMAs with different `anon_vma`s is not allowed.

The actual properties of the created mapping can be specified by the call parameters, which include the desired address and length, the protection setting, the file descriptor and the offset in the file. For this thesis, file mappings will be just marginal topic as it mostly concentrates on anonymous mappings.

Even though page offset originally meant offset inside of the mapped file, the anonymous mappings use it as well. However the value is set as a page offset from zero address and is therefore calculated as `address >> PAGE_SHIFT`. This is very important when it comes to merging later as page offset is one of the parameters compared before a successful merge.

1.9 The vma_adjust() function

```
int __vma_adjust(struct vm_area_struct *vma,
                 unsigned long start,
                 unsigned long end, pgoff_t pgoff,
                 struct vm_area_struct *insert,
                 struct vm_area_struct *expand)
```

This function rearranges VMAs locally, effectively performing a merge, or reassigns part of one VMA to another. The `vma_adjust()` function is just a wrapper for `__vma_adjust()`, which contains the implementation. All the parameters are more or less self-explanatory except for `insert`, which holds a new VMA that is to be inserted into the VMA interval tree and other structures when the `vma_adjust()` call is a result of a split operation. Other less understandable parameter is `expand`, which marks which VMA should be expanded when merging two or three VMAs together.

For the purposes of this section I will use the variable names from the code. First, the parameters `start` and `end` are compared with the `next` VMA to determine the intersection and set the appropriate `remove_next` and `adjust_next` variables, which are later applied. There is also the `importer-exporter` mechanism, which ensures that when moving pages between two VMAs, where the receiving VMA does not have an `anon_vma` assigned to it, then the providing VMA's `anon_vma` is used.

When preparations are ready, the actual adjusting is to be done. This consists of basically 2 parts. The first part is locking and unlocking of appropriate locks and adjusting boundaries and other parameters. The second part is removing the obsolete merged structures and cleaning up, the more sensitive part of the clean up is done before releasing the locks. When the `vma` is to be merged with not one but two following VMAs, which is indicated by `remove_next` value, then the adjusting is repeated twice, once for each of the merged VMAs. This is necessary because the presented code is capable of merging only two VMAs at once and therefore the merge is basically done twice.

1.10 The vma_merge() function

```
struct vm_area_struct *vma_merge(
                  struct mm_struct *mm,
                  struct vm_area_struct *prev,
                  unsigned long addr,
                  unsigned long end,
                  unsigned long vm_flags,
                  struct anon_vma *anon_vma,
                  struct file *file,
                  pgoff_t pgoff,
                  struct mempolicy *policy,
                  struct vm_userfaultfd_ctx vm_userfaultfd_ctx,
                  struct anon_vma_name *anon_name)
```

Function `vma_merge()` is used for attempted merging of VMAs. This function is called when a new VMA is created or when an already existing VMA is moved, resized or otherwise modified and there is a chance it might be merged with another VMA. Possible candidates are the previous and the following VMA. If the VMA in question neatly fills the hole between the two VMAs, all three of them might be merged together.

A number of conditions have to be fulfilled before the merge itself can be executed. First, the VMAs have to be adjacent, this in case of the previous VMA means that `prev→vm_end == addr` and for next VMA it means `next→vm_start == end`. Apart from this, both VMAs also have to be compatible in regard of their policies and also other flags and parameters including page offsets. This is checked in two special functions (Function can_vma_merge_before() (page 27) and Function can_vma_merge_after() (page 28)), which depend on whether the VMA being merged is before or after the VMA we are merging to.

A big issue preventing merging of VMAs is impossibility to merge two VMAs linked to two different AVs. This is because it is problematic to change to which AV a particular VMA is linked to. Every physical page, respectively every struct page representing it, has a pointer to its AV and so each physical page may be linked to only one AV. But the page might be used by several processes and therefore several VMAs. Each VMA has only one main AV, but can have several other AVs connected via AVCs. These other AVs are the result of a COW operation where VMA inherits AVs of its parent. One of the AVs has to be the same as is stored in the physical pages of the VMA in question. Assigning the physical pages to a different VMA (as a result of the merge) would imply adding new AV to the merged VMA. This would enlarge the AVCs and slow down their traversal, but most importantly it would break the invariant that additional AVs are inherited from ancestors.
Another option is to instead change the AV of the physical pages. This would be impossible if another VMA with a different AV were using it too. When the other VMA is linked to the same AV, we might change the AV for both of them, but finding all concerned VMAs might take some time and gets complicated.

On the other hand there are simple cases in which none of the above mentioned problems occur and they might be merged, however, in the current kernel two VMAs linked to two different AVs cannot be merged no matter what.

1.10.1 The is_mergeable_vma() function

```
static inline int is_mergeable_vma(
                struct vm_area_struct *vma,
                struct file *file,
                unsigned long vm_flags
                struct vm_userfaultfd_ctx vm_userfaultfd_ctx,
                struct anon_vma_name *anon_name)
```

Compares a given VMA A and given VMA flags and other parameters that describe VMA B. The check determines if these two VMAs can be merged together. VMA B is represented only by its parameters because in some cases it does not yet exist.

1.10.2 The is_mergeable_anon_vma() function

```
static inline int is_mergeable_anon_vma(
        struct anon_vma *anon_vma1,
        struct anon_vma *anon_vma2,
        struct vm_area_struct *vma)
```

When trying to merge an already existing VMA A and a possibly not-yet-existing VMA B, this function compares their AVs, which have to be the same. An exception is when at least one of the AVs does not exist (meaning that its VMA is not yet faulted) and at the same time VMA A does not exist or VMA A references only a single AV.

Function can_vma_merge_before()

```
static int can_vma_merge_before(
        struct vm_area_struct *vma,
        unsigned long vm_flags,
        struct anon_vma *anon_vma,
        struct file *file,
        pgoff_t vm_pgoff,
        struct vm_userfaultfd_ctx vm_userfaultfd_ctx,
        struct anon_vma_name *anon_name)
```

This function receives a list of parameters representing VMA B that might not yet exist and also an already existing VMA A that B should be merged to. In this `before` case B is or will be lying before A. This function internally calls functions `is_mergeable_vma()` and `is_mergeable_anon_vma()`, described above, and additionally checks if the page offsets are compatible.

Function can_vma_merge_after()

```
static int can_vma_merge_after(
        struct vm_area_struct *vma,
        unsigned long vm_flags,
        struct anon_vma *anon_vma,
        struct file *file,
        pgoff_t vm_pgoff,
        struct vm_userfaultfd_ctx vm_userfaultfd_ctx,
        struct anon_vma_name *anon_name)
```

Same as `can_vma_merge_before()` with the difference of VMA B being `after` VMA A.

1.11 The mremap() function

```
void *mremap(void *old_address,
             size_t old_size,
             size_t new_size,
             int flags,
              ... /* void *new_address */)
```

The `mremap()` syscall is used to remap a memory mapping, which means to shrink or expand it and potentially also move it at the same time. The `mremap()` call takes up to five arguments, which correspond to what task `mremap()` is supposed to perform. They are the old size and the old address, the new size and potentially also the new address. The last argument is an integer representing flags, they are only three compared to dozens of flags in the case of the `mmap()` syscall.

The new address can be either specified in combination with the `MREMAP_FIXED` flag, or, if not, the kernel will find a suitable new address on its own when moving is necessary and `MREMAP_MAYMOVE` is set to allow moving. When moving is not allowed but is necessary, because expansion in place is not possible, `mremap()` will fail and return an error value.

`mremap()` can be an opportunity for a merge, because in many cases the new location might be adjacent to a compatible mapping. However, in the current kernel, the merge never happens for anonymous mappings, because the page offset is not updated in `mremap()` and prevents the merge. On the other hand, for file mappings, this works when the mappings map the same file and are adjacent.

Code structure

As it was mentioned above, `mremap()` can do several things defined by the flags and other parameters and also by the circumstances of the chosen location.

The first thing done is evaluation of flags in combination with other parameters to die quickly in case of incompatible combinations. In such cases, `-EINVAL` is returned. Next, the VMA at the given address is found and if no such VMA exists, `EFAULT` is returned.

Now there are several cases of what needs to be done. In case of flags `MREMAP_FIXED` or `MREMAP_DONTUNMAP`, VMA will be moved either because the caller wanted it or because we have to preserve an old mapping. This is done in the `mremap_to()` function, described in the next subsection. If that is not the case it might be possible to shrink or expand the mapping in place. The shrinking is easy and it just means to unmap the range between the old and the new end address. The expansion is more complicated because it might interfere with the next VMA. If old length is exactly to the end of the area then it is possible to just expand the mapping when it will not intersect with the next area. On the other hand if the old length is not to the end or expansion would intersect with another area, it is necessary to find a suitable unmapped area using

get_unmapped_area() and move the VMA to the new location, which is done using the move_vma() function.

1.11.1 The mremap_to() function

```
static unsigned long mremap_to(
              unsigned long addr,
              unsigned long old_len,
              unsigned long new_addr,
              unsigned long new_len,
              bool *locked,
              unsigned long flags,
              struct vm_userfaultfd_ctx *uf,
              struct list_head *uf_unmap_early,
              struct list_head *uf_unmap)
```

This function moves the VMA mapping to a new location and optionally shrinks or expands it. This happens in mremap(), if the new location is either enforced by MREMAP_FIXED flag or, if the old mapping should not be unmapped, in case of MREMAP_DONTUNMAP flag. Basically, it just calls move_vma(), but it does a number of checks and preparations first. It unmaps the already-existing mappings in the target location and also exceeding part of the source mapping if the VMA is being not only moved but also shrinked, the rest of the source mapping is unmapped in the move_vma() as part of the move process (this is of course omitted if MREMAP_DONTUNMAP is set). If the new location is not fixed, the new location is chosen as the first suitable location found by get_unmapped_area() call.

1.11.2 The get_unmapped_area() function

```
get_unmapped_area(struct file *filp,
              unsigned long addr,
              unsigned long len,
              unsigned long pgoff,
              unsigned long flags)
```

Finds an unmapped area of the required length either at the specified location, if it is unmapped or flag MAP_FIXED is set, or anywhere where there is enough space.

1.11.3 The move_vma() function

```
static unsigned long move_vma(
              struct vm_area_struct *vma,
              unsigned long old_addr,
              unsigned long old_len,
              unsigned long new_len,
              unsigned long new_addr,
              bool *locked,
```

```
                unsigned long flags,
                struct vm_userfaultfd_ctx *uf,
                struct list_head *uf_unmap)
```

This function takes a VMA pointer, the old and the new addresses and the length, and moves the given VMA to the new specified location. It assumes the new location is already prepared and there is no mapping there. This is solved by the caller either by unmapping the old mapping or by finding a suitable unmapped area. Internally, the move is done by calling `copy_vma()`, which creates a copy in the new location (it might possibly be merged to its neighbours). Either way, the next step is to move the page table entries from the old to the new VMA, which is handled by `move_page_tables()`, it might happen that this fails and in that case `move_page_tables()` is called again in the opposite direction.

1.11.4 The copy_vma() function

```
struct vm_area_struct *copy_vma(
                struct vm_area_struct **vmap,
                unsigned long addr,
                unsigned long len,
                pgoff_t pgoff,
                bool *need_rmap_locks,
                bool *update_pgoff)
```

Creates a copy of a VMA in the specified location defined by the address, the length and the page offset. Other arguments needed are taken from the old VMA. If the VMA is anonymous and not yet faulted, the page offset is updated to correspond to the new address, this means recalculated as `new_addr >> PAGE_SHIFT`. After that, the `vma_merge()` is called, which tries to merge the not-yet-created VMA with its soon-to-be neighbours. If the merge succeeds, it might happen that the copy in the new location is actually merged with the old VMA, this might happen when the new location is precisely next to the old one. In this case, the pointer to old VMA must be updated. When the merge does not succeed, a duplicate VMA is created using parameters from the old VMA.

1.11.5 The move_page_tables() function

```
unsigned long move_page_tables(
                struct vm_area_struct *vma,
                unsigned long old_addr,
                struct vm_area_struct *new_vma,
                unsigned long new_addr,
                unsigned long len,
                bool need_rmap_locks,
                bool update_pgoff)
```

The `move_page_tables()` function is charged with replicating the page tables. This is done preferably at the PUD level if possible, but e.g. smaller VMAs do

not take up whole PUD, in which case the move is done at PMD or even PTE level. The page table move is done regardless whether we merge the VMA with its neighbour or not.

1.12 Other kernel memory mechanisms

1.12.1 Rmap walk

Rmap walk is a mechanism that goes through all mappings of a page and performs an arbitrary action. Because VMAs can easily be merged or split, pages do not reference their mappings directly. Instead, each anonymous page has a pointer to an AV, which serves as a bridge between pages and their mappings as described in more detail in Anonymous virtual memory area aka AV (page 22). The rmap walk basically goes through all the VMAs via the AV and compares the page offset values as can be seen in Figure 1.4. It is important to realise that VMA's position in virtual memory and therefore also virtual address can change during `mremap()` call, but VMA's page offset is immutable the same way as the page offset values stored in individual pages. For KSM pages and file mapping pages the implementation is a bit different.

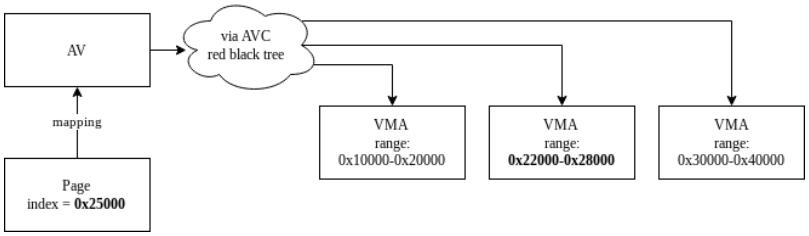

Figure 1.4: Rmap walk identifying the correct VMA using AV and the page offset.

Rmap walk is used during migration to replace the page table entries with migration entries for the time of migration. Another usage is during unmapping of a page to find and replace the page table entries with swap entries.

1.12.2 The do_munmap() function

```
int do_munmap(struct mm_struct *mm,
              unsigned long start,
              size_t len,
              struct list_head *uf)
```

Unmaps a range at the given address with the given length. This unmaps all the VMAs residing there and might also split up to two VMAs, one at the beginning and one at the end of the range, if the range just intersects them.

1.12.3 Brk syscall

Enlarges or shrinks program break, which is the end of program's data segment. This is used to allocate more or less memory for the running program with regard to maximum data size limitation.

Function do_brk_flags()

This subfunction internally allocates a new mapping, or, if possible, enlarges the old one, creating an opportunity for a merge.

1.12.4 The __split_vma() function

```
int __split_vma(struct mm_struct *mm,
                struct vm_area_struct *vma,
                unsigned long addr, int new_below)
```

Splits a VMA into two based on the `addr` address using `vma_adjust()`.

2. Design

The goal of this project was to discover why VMAs sometimes do not merge even though it might be possible and to resolve such cases in as many situations as possible. In this part, we are going to summarize what decisions were made and what obstacles had to be overcome, including a detailed explanation of the final solution. This means everything from choosing the appropriate points of entry, verifying the conditions to implementing the actual changes including using already available solutions for partial problems.

2.1 Analysis of the problem

This section concentrates on where the problem is located and outlines the process necessary to solve it.

2.1.1 What prevents merges

When analysing the kernel code and specific cases in which merge does not happen for anonymous VMAs, even though it might, it comes down to these two main reasons.

The first is the page offset, which is one of the parameters that has to be compatible in order to merge two VMAs. This makes perfect sense for memory mapped files, in which case it ensures that two parts of a single file represented by two individual VMAs can merge together only if the parts follow up and the resulting merged VMA would be one continuous file part. On the other hand, for anonymous mappings, dealing with the page offset is mostly just a side effect of using identical kernel mechanism for both anonymous and file mappings. When anonymous mapping is created, its page offset is identical to its virtual address with the page bits removed. This is fine until the mapping is moved to a new virtual address. In such a case in the current kernel, the page offset is updated only if the mapping has not been accessed by a process and therefore none of its pages have been faulted. In other cases, the page offset remains at the original value that does not correspond to its new virtual address. If two mappings are moved next to each other then this inconsistency prevents any possible merge even if all other conditions are met. This means that the first main change has to enable the update of the page offset when moving a mapping, which it turns out is not easy and for some cases sharing pages between processes not even possible.

The second main reason is the AV to which the VMA belongs. In current kernel it is possible to merge only those VMAs that belong to identical AV. The most obvious reason why this is not allowed is that if such a merge would happen, it would mean updating all involved pages. This is necessary because every normal anonymous page stores a pointer to its AV in the `mapping` property and this property would need to be updated during such a merge. This is complicated because traversing and locking pages during the merge process is not supported and most importantly the changes at the VMA, AV and page level should be done atomically at least from the Rmap walk (page 33) point of view. Allowing

different AV merging does not only help when moving a mapping but also when the mapping is only modified in terms of flags or access protection in place e.g. in `mprotect()` call.

Moreover, these two reasons can apply simultaneously and so removing only one of them would yield fewer additional successful merges. And unfortunately both cases require update at the level of physical pages, which turns out to be problematic and probably the reason it has not been implemented yet. A detailed explanation is in dedicated sections for each of the problems.

One last small identified problem preventing more merges is a missing `vma_merge()` call in `mremap()` when expanding an already existing mapping, in which case a merge might be possible with the following mapping. An overview of system calls potentially benefiting when the problems are resolved are summarized in Table 2.1.

System call	Description
`mremap()[PAE]`	Expanding and/or moving an already existing mapping may relocate it next to another mapping and enable a merge.
`mmap()[A]`	New mapping being created next to a compatible neighbouring mapping may result in a merge involving both its neighbours with potentially different AVs.
`brk()[AE]` `sbrk()[AE]`	Modifies the program break which can enlarge a mapping and enable a merge with a new neighbour.
`mprotect()[A]` `pkey_mprotect()[A]`	Modifies memory protection and therefore can unify the protection with a neighbouring mapping to enable a merge.
`madvise()[A]` `process_madvise()[A]`	Can in some cases modify flags of a mapping and this way unify the flags with a neighbouring mapping.
`mlock()[A]`	Locks pages in the address range, which can unify flags with a neighbouring mapping.
`mbind()[A]`	Sets memory policy for a memory range, which can unify flags with a neighbouring mapping.

Table 2.1: Overview of system calls using vma_merge() and their beneficial from resolving problems, page offset[P], different AV[A], expansion merge[E].

2.1.2 Solution outline

As mentioned in What prevents merges (page 35), we have to update `page→index` for individual pages involved in a `mremap()` call and also `vma→vm_pgoff` for the mapping itself to enable merges for mappings with faulted pages. Moreover, this update is necessary even when the `mremap()` call does not result in a merge - this way, the page offset of the mapping is updated to enable a merge with another mapping that might become adjacent in the future. Details in Page offset update (page 49).

To allow merging of VMAs with different AVs, it is necessary to perform an update of individual pages involved in the merge so that their `page→mapping` points to the correct AV after the merge is done. The resulting AV is always the AV that belonged to one of the mappings at the beginning and so only pages of one of the mappings need to be updated. Details in Different AV merging (page 51).

Both approaches outlined above require an update of individual pages, which is problematic because if the pages are shared with another process, the update would interfere with the other process as well and could require a cascade of updates. This could get quite complicated and slow down the whole merge process. Therefore, when deciding to perform the update itself, we must first check if the pages are not shared, which is explained in Checking if VMA is shared (page 38), and if the pages are indeed shared, then we cannot proceed with the merge.

An addition of a `vma_merge()` call during expanding `mremap()` is explained in Allowing merge during mremap() in-place expansion (page 55).

2.2 Checking if VMA is shared

For every VMA that we want to modify either to update page offset or to assign it to a different `anon_vma`, we have to check if its physical pages are not shared with other processes. If two VMAs usually in two different processes share the same physical pages, we say, these VMAs are shared. There are two approaches to effectively recognise this. The first one looks at the their AV structures that head a list of private related VMAs and searches for a parent-child relationship. The second approach goes through all the pages belonging to the VMA and checks their mapcount, which indicates in how many mappings the page is involved, however, this approach is more complicated and slower.

AV checks

In order to check if a VMA shares some pages with a different VMA from another process, we perform the parent check and the child check. Both are described in detail in Detecting parent-child relationship in AV trees (page 40) and work with an AV, where the parent check checks for a parent and the child check checks for children.

Parent check is a simple comparison of AV and its root: `vma→anon_vma→root == vma→anon_vma`. If it is the same object then there is no parent. But if it is two different objects, then the parent exists and we are looking at a shared scenario.

Child check is complicated and involves traversing a red black tree of private related VMAs and their AVs from the original AV point of view. The check can be simplified by checking AV's degree, which is always higher than one in the shared scenario, but it is only an implication, not equivalence. Therefore if it is higher than one, normal checks must follow.

Individual page checks

Checking if an individual page is shared is a bit complicated, because the page might not be present in memory, can be swapped out or even worse, it can be swapped out for one process and not for the other ones.

First, we check if the page is not swapped out in this process by looking at the PTE's swapped bit, which indicates if the PTE is actually a PTE or rather a swap entry. For the case of swap entry, we check the swap map, which holds a number of references that basically indicates how many processes reference this page in a swapped out state. It is not so simple though as there are also some flag bits stored in the same variable, but fortunately we only want to check if the page is shared or not and therefore basically anything bigger than one means the page is shared. This is true because both flags imply that the page is shared somehow. The first flag, `SWAP_HAS_CACHE` with value `0x40`, indicates that the swap cache is active. This means that for some processes the page is mapped and for some it is not, which inherently implies the page is involved with more than one process and therefore shared. The second flag is `COUNT_CONTINUED` and indicates that the reference count is too high to fit in only one instance of the variable. This again

implies the page is necessarily shared. See Swap cache mechanism (page 21) for more details.

In the other case, when the PTE is not swapped out and we can access the page structure, we call functions `page_mapcount()` and `page_swapcount()`, which give us the number of processes that map this page and the number of processes that have this page swapped out. The sum of these two is the total number of processes that use this page and therefore if it is higher than one then the page is shared.

In both cases, we use the page walk with flags `WALK_MIGRATION` and `WALK_LOCK_RMAP` to go through all the necessary pages. These two flags ensure that the page is not in the migration entry state where we would not be able to access the above-mentioned parameters. If the page is in the migration entry state, then the page walk first waits for the migration to end and afterwards takes locks preventing new migration from starting until all checks are finished.

2.3 Detecting parent-child relationship in AV trees

Shared physical pages in case of anonymous mappings can always belong to only one AV through their `page→mapping` parameter, which in this case holds a pointer to the AV in question. On the other hand, each VMA can have several AVs, but one of them is always the main one and is accessible through `vma→anon_vma`. Sharing usually happens because of forking and the underlying COW mechanism. Only the original VMA has its main AV identical to the AV of all of its pages. Copies of the VMA in other processes have several AVs and the one used by the shared pages is one of the other AVs, the main one is used only for new pages that come into being when the VMA copy is enlarged or when the shared pages are to be written and have to be copied. These new pages are then exclusively owned by this VMA copy and their AV matches the main AV of the VMA copy.

The reason why the child VMAs have additional AVs is because the shared pages are gradually replaced with new ones that can belong to a different AV rather than the original AV, which would otherwise keep growing and unrelated pages would unnecessarily belong to a single huge AV.

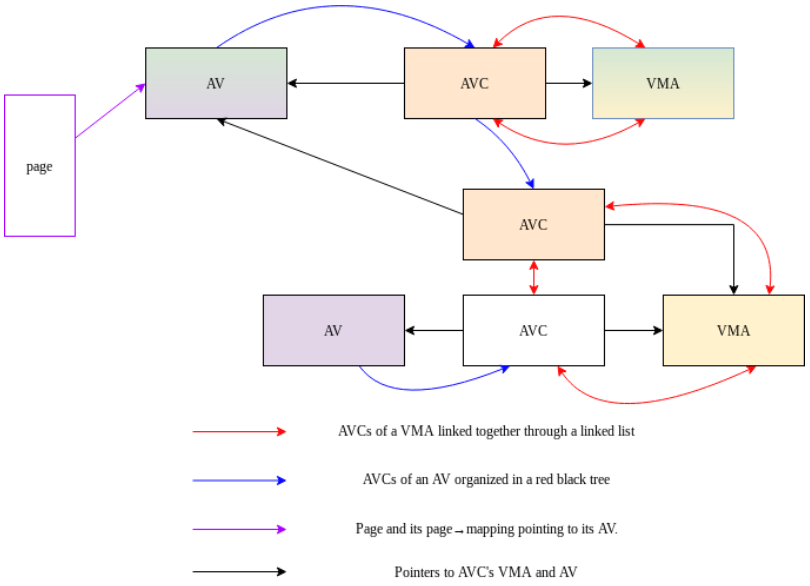

Figure 2.1: Finding two different main AVs.

The task of detecting the parent-child relationship means that we get a VMA and must determine if it is sharing its pages with other processes. This is done by taking its main AV and then going through all the VMAs related to this AV. If any of these related VMAs have a different main AV from the given one, then we have a parent-child relationship and therefore an instance of page sharing. This process is demonstrated in Figure 2.1, the individual steps are highlighted by different colors and their order can be seen in Figure 2.2. More details about AVCs can be found in Structure anon_vma_chain (page 23).

Figure 2.2: Phases of detecting parent-child relationship.

If all these AVs of the related VMAs are identical to the original one, then we have a group of VMAs all associated with the same AV, but not sharing any pages. This is the case e.g. when we split an existing mapping into two or create two adjacent mappings that have different flags (if the flags were the same, they would merge). An example can be seen in Figure 2.3.

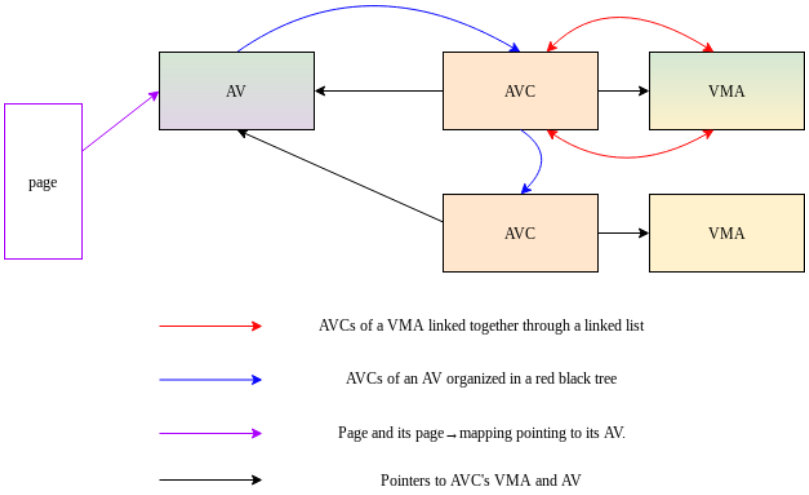

Figure 2.3: Sibling VMAs associated with the same AV.

The entry point through the given VMA is `vma→anon_vma` and then through `anon_vma→rb_root`. The red-black tree is a tree of AVCs, which is a structure connecting an AV and a VMA. Nonetheless, we can easily get VMA through AVC by `avc→vma` and then its main AV by `vma→anon_vma`.

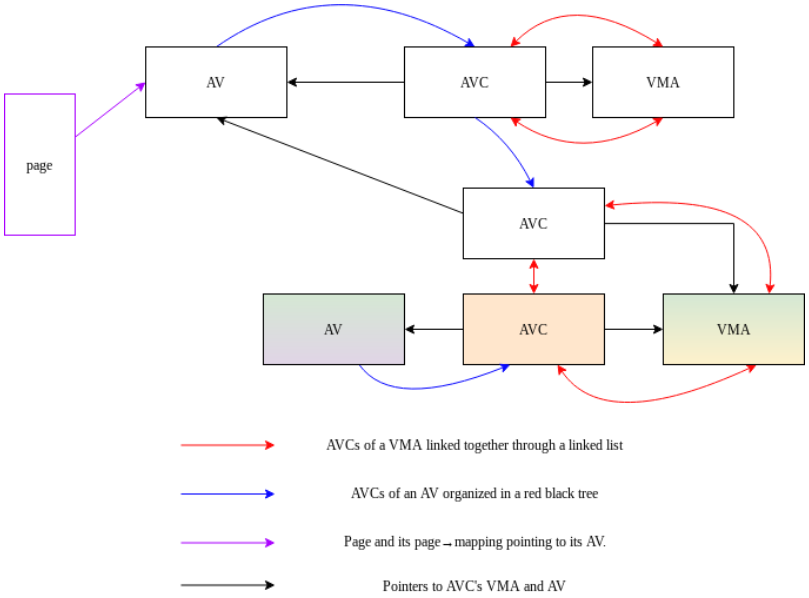

Figure 2.4: Finding only one main AV.

This whole mechanism works only in the case where the given VMA is the parent VMA, as can be seen in Figure 2.4. If the given VMA is a child VMA in a parent-child relationship then we determine this by accessing its main AV and then by checking if this AV has a root AV different from itself, as can be seen in Figure 2.5, Figure 2.6 and Figure 2.7. This is actually much easier and it is done simply by `vma→anon_vma→root`. Unfortunately, we do not know if we are looking at a child or a parent and therefore we do both checks.

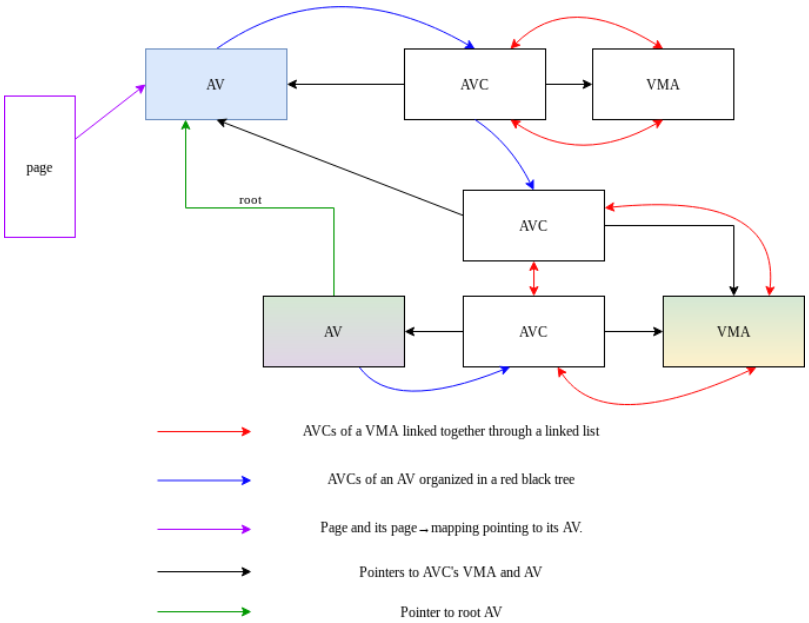

Figure 2.5: Detecting parent-child relationship from child.

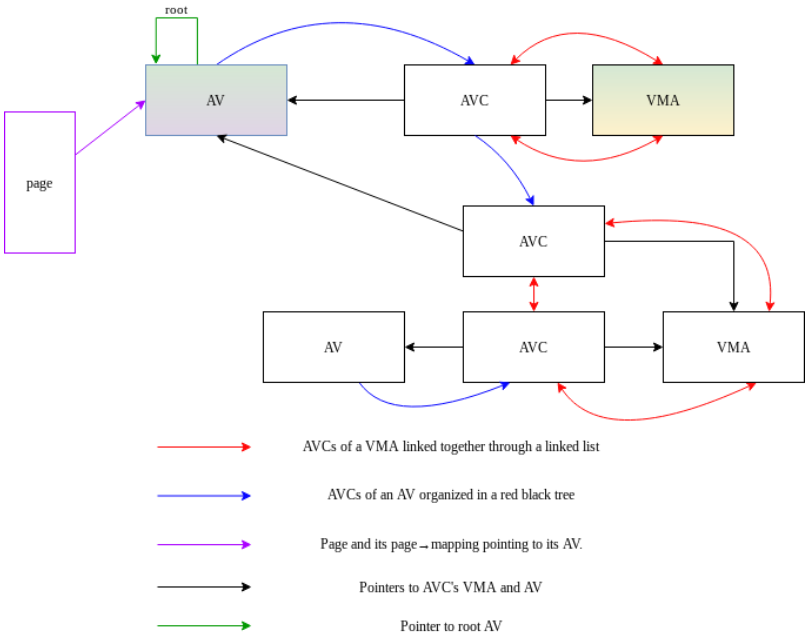

Figure 2.6: Trying to detect parent-child relationship from parent using the root method.

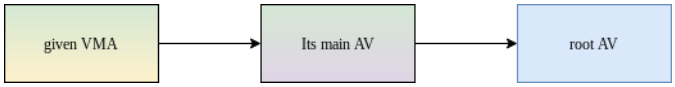

Figure 2.7: Phases of detecting parent-child relationship via root method.

More detailed schemas showing even support structures are demonstrated on a child with a parent and a grandparent. Figure 2.8 gives the grandparent point of view and Figure 2.9 gives the the grandchild point of view.

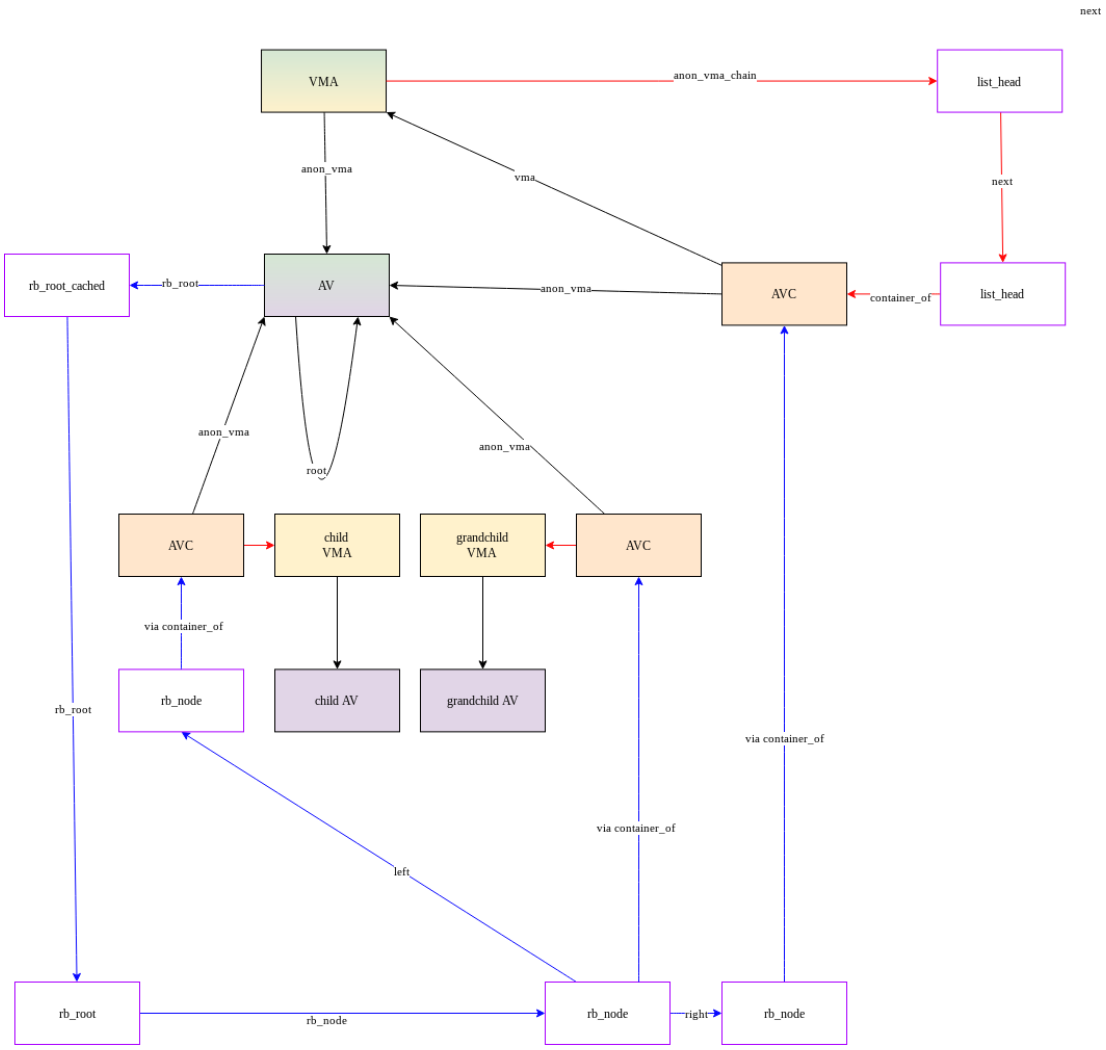

Figure 2.8: AVC tree traversal from grandparent.

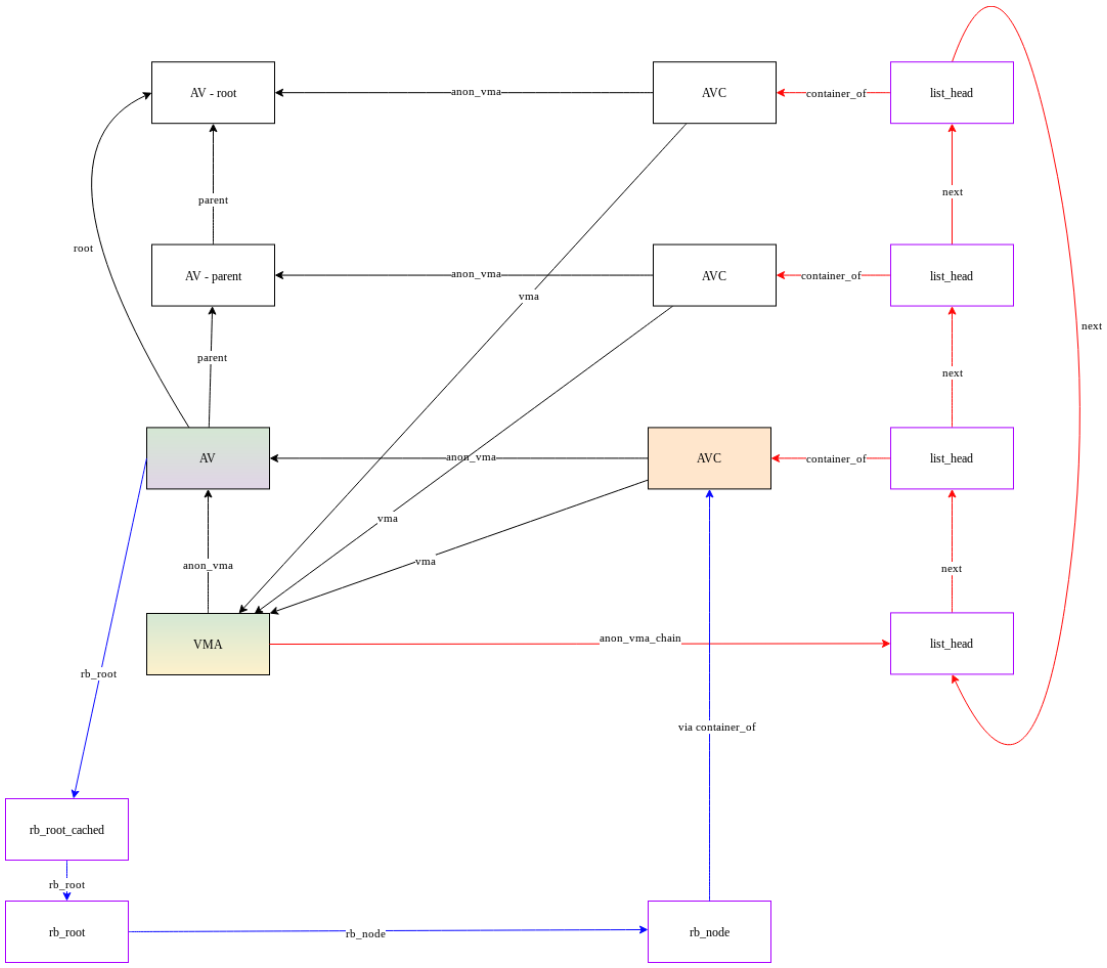

Figure 2.9: AVC tree traversal from grandchild.

2.4 Common helper functions

KSM impact

As mentioned in Analysis, KSM is a page de-duplication mechanism. As a result, mapcount of a de-duplicated page can be higher than mapcount of a normal page, which would interfere with the page checks trying to determine if the page is shared or not. Fortunately, in the case of the page offset update, KSM is always disabled in `move_vma()` for the given area by a `madvise()` call that sets the area to `MADV_UNMERGEABLE`. This temporarily duplicates all the pages in question, but allows for KSM to merge the pages again after `mremap()` is done.

On the other hand, in the case of merging two VMAs with different AVs their pages might be in the de-duplicated KSM state. This will prevent the change of `page→mapping` (holding the AV) that has to be updated during the merge. But fortunately when they return to normal state, their `page→mapping` will be reconstructed from their VMA, which will already hold the possibly updated pointer to its AV.

2.4.1 Pagewalk flags

The pagewalk mechanism did not have a flags mechanism that would be able to perform predefined special operations like locking if a flag is specified. Although the `action` feature is somewhat similar, it does not perform any action on its own. It only instructs the pagewalk core to repeat or skip user-defined table entry operations.

Two flags were added. The first one for locking rmap locks to prevent rmap walk from interfering with the page update action, which could cause failure to find the correct VMA for the page. The second one for enabling migration waiting. Each PTE can be in the state of migration, which is determined by checking if PTE contains a migration entry instead of a normal PTE. Waiting for the migration means the page walk must unlock locks needed for migrations and lock them again after the migration has ended.

2.5 Development history

During the analysis and development, there were several solutions that turned out not to work or at least not to work well enough and had to be redone. This was mostly because some problems were quite specific and hard to discover and of course because of inadequate documentation. This prolonged the whole development process.

There are three more or less working historical versions of the patches located in the attachment's `sources/history` directory that can serve as checkpoints along the development history described in this section.

Only pgoff update

The original idea for a solution was to only add the updating of the page offset when the mapping is moved and this way enable new merges in `mremap()`. This was promising until tracing was added to measure the number of successful merges, then it turned out that most of the merges were still not happening because of different AVs.

Not updating pages

The next problem was that updating the page offset only in the VMA was not enough, although it worked in most of the cases and the system seemed to work more or less properly. Unfortunately, the inconsistency between the offset in VMA and in the page itself causes problems when swapping out the page in question. Therefore updating of page offset stored in `page→index` was added.

Adding AV update

As mentioned above, the difference between AVs of the two VMAs we are trying to merge prevents the merge. This can be solved by allowing such merges and then reassigning pages from one AV to another. This allows additional merges not only in the case of the `mremap()` calls, but also and even in greater number in the case of the `mprotect()` calls and other calls that do not move the mapping but only change its flags.

Using follow page instead of page walk

Originally, when accessing pages that should be updated, `follow_page()` was used. The disadvantage of this approach is that `follow_page()` internally performs a full page table walk before it returns the page mapped at the given address. This is especially a problem when going through higher numbers of pages at the same time. `follow_page()` was therefore replaced using the page walk framework[1], which allows going through several PTEs (which point to a page) at a time and saves some unnecessary page walking.

Pgoff update moved to move_page_tables()

Using the page walk is much better than the `follow_page()` call, but in the case of page offset update, it is even better to do the page update in `move_page_tables()`, which moves all the page tables anyway. This again saves some extra traversing time.

Transparent huge pages

A problematic case is when the VMA does not map normal physical pages but instead transparent huge pages that are basically bigger blocks of pages. Eventually, to avoid additional complexity, THPs are split and then processed as normal pages.

[1] as implemented in mm/pagewalk.c

Added waiting for migration if we have migration entry

When working with the PTEs, occasionally a PTE might be either in the swap entry or the migration entry state. In the case of the swap entry, we can access reference counts or swap cache to acquire the number of processes mapping the page. Furthermore, during the page update itself, we do not need to do anything as the mapping and the index are reconstructed from the VMA when the page is swapped back in. More problematic is the case when PTE holds a migration entry, which does not provide the needed information and cannot be updated. In this case, it is necessary to wait for migration to finish and unlock locks preventing the migration from finishing.

Added rmap walk locks to not race with rmap walk

When updating a page, the rmap locks have to be taken to prevent rmap walk running at the same time. Otherwise the rmap walk might partially work with the old page data and partially with the new page data.

Redone AV update

The `anon_vma` update in `vma_adjust()` had to be redone as it was inherently wrong and was updating only some of the necessary cases. Now the update is done in `vma_adjust()` only for the `mprotect()` cases and in `move_page_tables()` for moving `mremap()` cases (together with page offset update). Although this was quite a big issue, the related bugs and failures occurred only sporadically and were hard to reproduce or pinpoint.

Pgoff update at the lowest level possible

Pgoff update in `move_page_tables()` has to be done at the lowest level to ensure correct locking order of page lock, rmap locks and PTE lock. A similar thing has to be done for all the other page walks that have a wrong locking order. This is achieved by try-locking instead of locking pages and if needed again unlocking all previous locks and retrying the step via the `ACTION_AGAIN` directive in page walk mechanism.

2.6 Refactoring vma_merge()

The first reason for this refactor is to make the function suitable for tracing of successful merges that are made possible by Page offset update (page 49), Different AV merging (page 51) and Allowing merge during mremap() in-place expansion (page 55). The second reason is to make it shorter and more understandable by eliminating code duplicity and unnecessary indentation mainly in the case of merge next check. This is done by first doing checks and caching the results before executing the specific merge case itself. Exit paths are also unified.

2.7 Page offset update

This first change adjusts page offset of a VMA when it's moved to a new location by `mremap()`, which is only possible for VMAs that do not share their anonymous pages with other processes. It is checked by going through the AV tree and looking for parent-child relationship as described in Checking if VMA is shared (page 38). Also if the VMA contains any transparent huge pages, they are split to avoid dealing with them. This is all done in functions `can_update_faulted_pgoff()` and `is_shared_pte()`. At the start, the situation may look like in Figure 2.10.

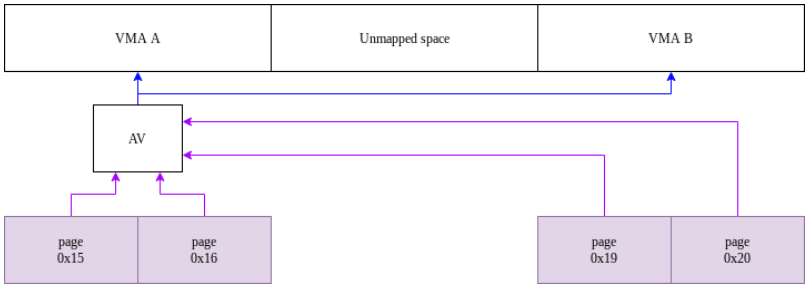

Figure 2.10: At the start we have two VMAs with an unmapped space between them. The right VMA is to be moved next to the left one and merged.

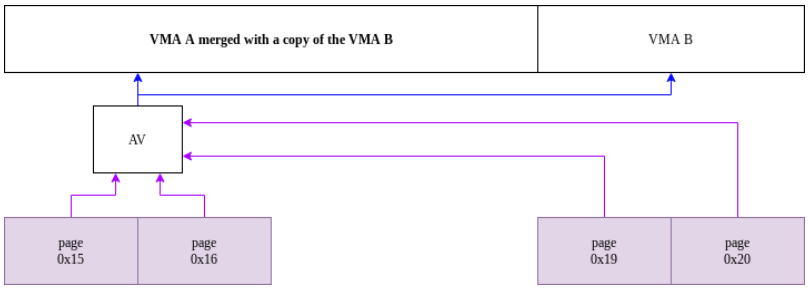

Figure 2.11: VMA copy merged with its new neighbour. Old VMA B is still present.

If none of the pages is shared then we proceed with the page offset update. This means updating page offset in `copy_vma()`, which is then used when creating a copy of the VMA or possibly when deciding whether to merge with a neighbouring VMA as seen in Figure 2.11, where the copy is already merged

with its new neighbour. We also set `update_pgoff` to true to later update page offsets of individual pages that are moved as seen in Figure 2.12. This is done in `move_page_tables()` when moving individual PTE entries to the target VMA. The page offset update actually forces the move to happen at the PTE level by using function `move_ptes()`. It is necessary to perform the move at the lowest possible level, because the page update must happen atomically with the move and that is not possible when moving bigger entries like PMD or PUD.

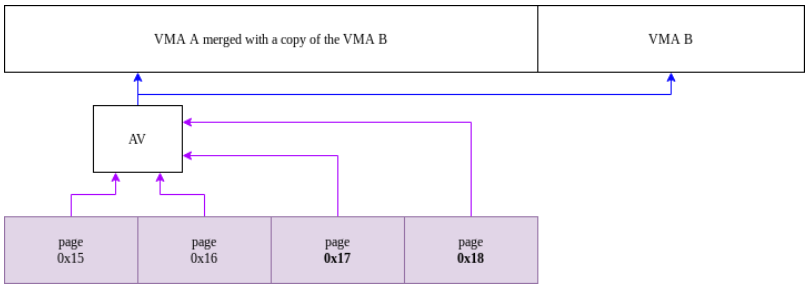

Figure 2.12: Update of page offset of individual pages. Pages in figure are moved only symbolically.

We do not need to update swapped out pages, because in that case the page offset is reconstructed automatically from the VMA after the page is swapped back in. As mentioned above, there is a small amount of time between checking and actually updating the page offset of the pages, as well as between merging VMAs and again updating the pages. This could potentially interfere with rmap walk, however, in that case rmap walk will still use the existing old VMA, as it would before the `mremap()` started. Any other potential changes to the VMA or pages apart from rmap walk are prevented by `mmap_lock`, which prevents forking and therefore also COW and hence creating any copies and therefore sharing during the merge process. Keeping in mind that pages are not shared and belong to only one process, there is no other process which might fork and in that way create a copy of the pages in question. However, if a page is shared, we can't update page offset of that page, because that would interfere with the page offset as seen from the other processes using the same page. Page offset is basically immutable as long as the page is used by more than one process. Previously, adjusting page offset was possible only for not yet faulted VMAs, even though page offset matching the virtual address of the anonymous VMA is necessary to successfully merge with another VMA.

In the end, when the page offset update results in a successful merge, then the old VMA structure can be freed as is seen in Figure 2.13.

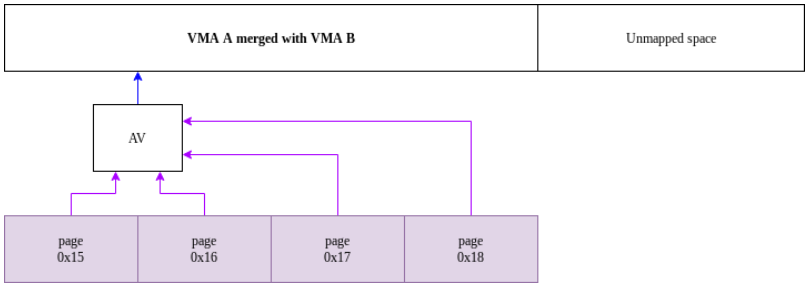

Figure 2.13: Removing old VMA after successful merge.

2.8 Different AV merging

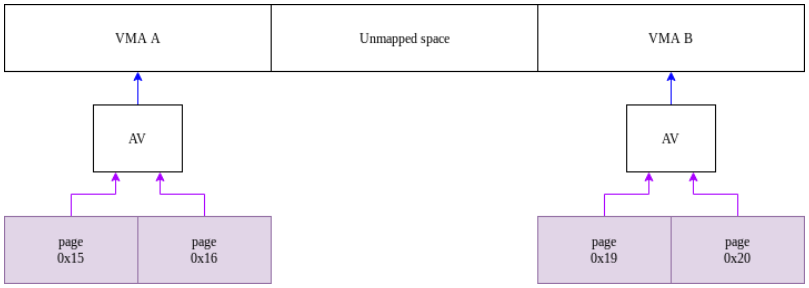

Figure 2.14: Before pgoff and AV update merge.

The goal of this change is to enable merging of a VMA even when it is linked to a different AV than the VMA it is being merged to, however, only if the VMA in question does not share any pages with a parent or child process. This enables merges that would otherwise not be possible and therefore decreases the number of VMAs of a process. The initial state can look like in Figure 2.14 or in Figure 2.15 depending on the relative position of the VMAs.

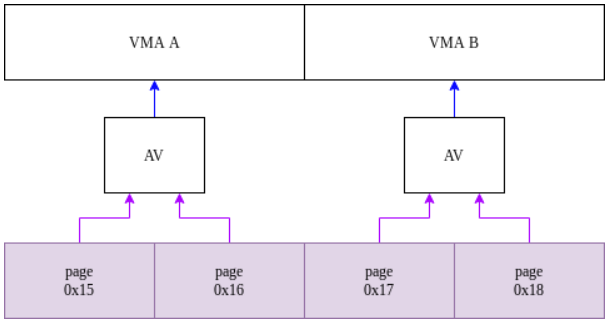

Figure 2.15: Before AV update merge.

The VMA in question is first checked at the level of AV to find out if it shares any pages with a parent or child process, see Detecting parent-child relationship in AV trees (page 40). This check is performed in `is_mergeable_anon_vma()`, which is a part of `vma_merge()`. In the following two paragraphs we are using cases 1 through 8 as described in comment before `vma_merge()`, which can be seen in Figure 2.16.

```
 * The following mprotect cases have to be considered, where AAAA is
 * the area passed down from mprotect_fixup, never extending beyond one
 * vma, PPPPPP is the prev vma specified, and NNNNNN the next vma after:
 *
 *     AAAA              AAAA                   AAAA
 *    PPPPPPNNNNNN    PPPPPPNNNNNN       PPPPPPNNNNNN
 *    cannot merge    might become       might become
 *                    PPNNNNNNNNNN       PPPPPPPPPPNN
 *    mmap, brk or    case 4 below       case 5 below
 *    mremap move:
 *                        AAAA               AAAA
 *                    PPPP    NNNN       PPPPNNNNXXXX
 *                    might become       might become
 *                    PPPPPPPPPPPP 1 or  PPPPPPPPPPPP 6 or
 *                    PPPPPPPPNNNN 2 or  PPPPPPPPXXXX 7 or
 *                    PPPPNNNNNNNN 3     PPPPXXXXXXXX 8
 *
 * It is important for case 8 that the vma NNNN overlapping the
 * region AAAA is never going to extended over XXXX. Instead XXXX must
 * be extended in region AAAA and NNNN must be removed.
```

Figure 2.16: vma_merge() comment.

For cases 4 through 8 and partially for case 1, the update itself is done in `__vma_adjust()`. Other cases must be solved elsewhere, because `__vma_adjust()` can only work with pages that already reside in the location of the merge, in other words if VMA already exists in the location where merge is happening. This points to the cases 2, 3 (and partially case 1 where the next VMA is already present but the middle one is not), which happen when we are either expanding or moving a VMA to the location of the merge. However, at the time of the merge, the VMA is not there yet and therefore the page update has to be done later elsewhere as there is no way how to access the pages in `__vma_adjust()`.

An easy subcase is if the pages do not exist yet and therefore there is nothing to update. This happens e.g. when expanding a mapping in `mmap_region()` or in `do_brk_flags()`, where the pages themselves are created later.

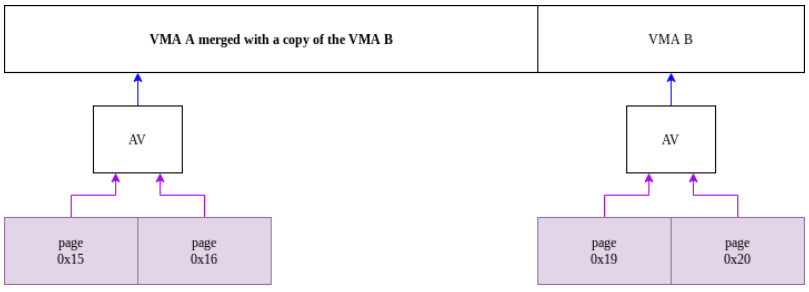

Figure 2.17: Merging copy of the VMA with its new neighbour.

On the other hand, during a `mremap()` call that moves an already existing and possibly faulted mapping, the pages do exist and have to be updated. The VMA is first copied in `copy_vma()` to the target location and possibly merged with a new neighbour in `__vma_adjust()` as shown in Figure 2.17. In this case, the page update is done in `move_page_tables()`. It is actually quite simple because we already introduced page update in Page offset update (page 49) and therefore

the only change is updating one more parameter as seen in Figure 2.18 compared to Figure 2.12. If rmap walk happens between `__vma_adjust()` and page update in `move_page_tables()`, then the old VMA and the old AV are used as it would happen before starting the whole merge, afterwards the new (merged) VMA and AV is used.

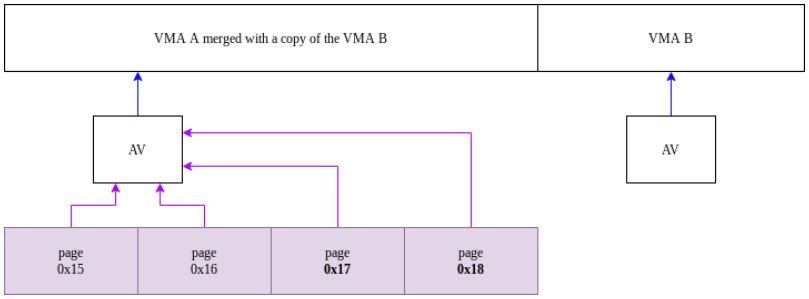

Figure 2.18: Page offset and AV update. Pages in figure are moved only symbolically.

Let's get back to the cases 4 through 8. These cases correspond to merges which are a result of a `mprotect()` call or any other flag update that does not move or resize the mapping. The partial result of such a merge on the level of VMAs can be seen in Figure 2.19, where the future VMA is already expanded, but the old one is also still existing. Together with a part of the case 1, the update of physical pages is handled directly here in `__vma_adjust()` as mentioned before. First, it is determined which address range should be updated depending on the specific case 1, 4, 5, 6, 7 or 8. Second, the AV value to be set to the `page->mapping` must be determined. However, it is always the AV belonging to the `expand` parameter of the `__vma_adjust()` call.

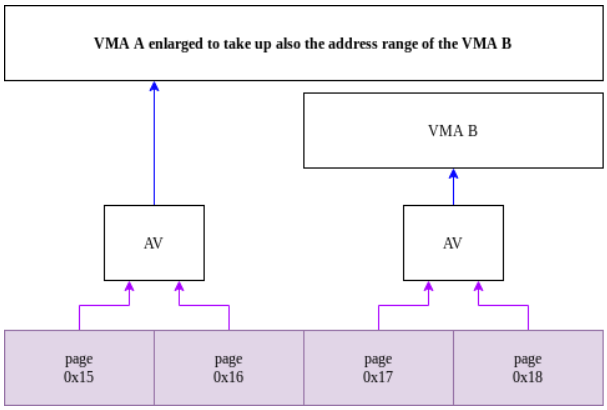

Figure 2.19: Future VMA enlarged but the old is still existing.

The reason we have to update the pages is that in `__vma_adjust()`, the ranges `vm_start` and `vm_end` are updated for involved VMAs and so pages can belong to different VMA and AV from that point on.

The problem is that these two updates (VMAs and pages) should happen atomically from the rmap walk point of view. This would normally be solved

by using rmap locks, but at the same time, we must keep in mind that the page migration uses rmap walk at the start and at the end and so rmap locks might trap the migration in the middle. This would cause a PTE to not point to any actual page and remain in the migration entry state, which would block the page update. The solution is to page walk all the relevant pages, if necessary drop rmap locks for time needed to allow page migration to end and update the `page->mapping` (i.e. the AV) attribute of each page.

This whole page update must be done after the `expand` VMA is already enlarged, but the source VMA still has its original range as seen in Figure 2.20. This way if rmap walk starts while we are updating the pages, it will work either with the old or the new AV and therefore also the old or new VMA.

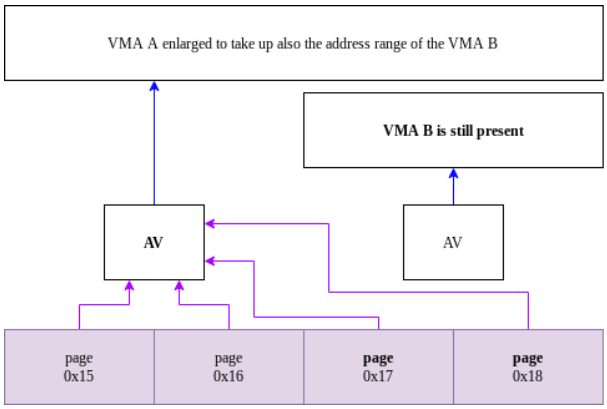

Figure 2.20: Reassigning pages to the new AV.

If the PTE is a swap entry or points to the zero page or a KSM page, then the page is not updated and the correct mapping is reconstructed from the VMA itself when the page returns to normal state. Again as mentioned and explained in Page offset update (page 49), the pages may not become shared between `vma_merge()` checks and actually merging in `__vma_adjust()` as potential fork is prevented by `mmap_lock`. Additionally, in the case where one of the VMAs is not yet faulted and therefore does not have an AV assigned, this change is not needed and merge happens even without it.

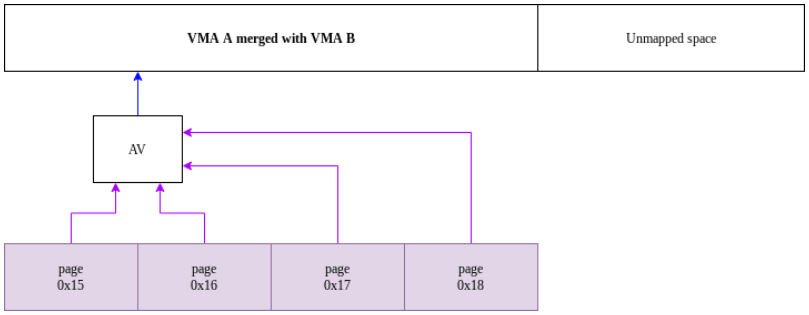

Figure 2.21: Removing old VMA and AV (when it is unused).

In the end, the old AV and possibly VMA is freed as seen in Figure 2.21 and Figure 2.22 for the relevant cases.

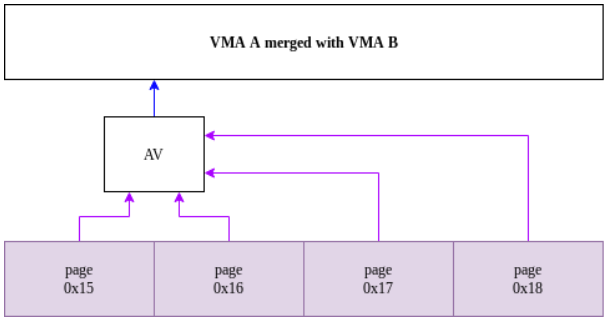

Figure 2.22: Removing old VMA and AV (when it is unused).

2.9 Allowing merge during mremap() in-place expansion

When `mremap()` call results in an in place expansion, it might be possible to merge the VMA with the next VMA if it becomes adjacent. The only missing part is a `vma_merge()` call after the expansion is done to try and merge.

2.10 Tracing

A new tracepoint was added to measure how many successful merges were made because of these changes and which change or changes in particular made the merge possible. There are three different changes and therefore also three different tracing categories.

2.10.1 AV merging

A trace support was added to `vma_merge()` to measure successful and unsuccessful merges of two VMAs with distinct AVs as well as the reason why the merge was successful or unsuccessful using the `vma_merge_res` enum as can be seen in the Table 2.2.

Name	Description
`MERGE_FAILED`	Merge failed for other reasons than the AV.
`AV_MERGE_FAILED`	Merge failed because one of the VMA is shared and the AVs are different.
`AV_MERGE_NULL`	Merge is allowed because one of the AV is null.
`MERGE_OK`	Equal to `AV_MERGE_NULL`. Implementation detail.
`AV_MERGE_SAME`	Merge is allowed because the AVs are identical.
`AV_MERGE_DIFFERENT`	Merge is allowed even though the AVs are different.

Table 2.2: Overview of vma_merge_res enum values.

2.10.2 Page offset merging

Trace support was added to `copy_vma()` to measure successful merges made possible by an update of the page offset.

2.10.3 In place expand merge

Trace support was added to `mremap()` to measure successful merges made possible by the added `vma_merge()` call.

3. Testing

This chapter contains a summary of tests created to ensure the patches do enable merges under the specified conditions and at the same time do not cause any warnings, bugs or other problems. All the source files and directories referenced from this chapter are included in the attachment.

3.1 Merge tests

These tests check if all patches described in this thesis actually work and merges happen when they should happen. And vice versa if the merge should not happen because of some limitations, these tests check that the merge does not happen or cause any other problems. They are all included directly in the patches in the form of standard kernel self-tests. Self-tests are located in `tools/testing/selftests` and can be build using `make`. The source file `vm/mremap_test.c` contains Expand merge test (page 57) among other tests that are not part of this thesis, the file `vm/merge_tests.c` contains all the other tests described below. They can both be executed by using the executables of the same name, `vm/mremap_test` and `vm/merge_tests` respectively.

3.1.1 Expand merge test

This test validates that the merge is called when expanding a mapping in place. A mapping containing three pages is created and middle page is unmapped, which splits the mapping into two. Then the mapping containing the first page is expanded so that it fills the created hole. The two parts should merge creating a single mapping with three pages as in the beginning.

3.1.2 Page offset update merge test

This test checks that for not shared mappings, page offset update happens and can lead to a previously impossible merge. The test creates a mapping and unmaps its middle part, creating two separate mappings (with an identical AV) with a space between them. It then tries to move the second mapping next to the first one to merge them. There is a subvariant of the test for shared mappings that verifies that the merge does not happen for shared mappings.

3.1.3 Different AV merge test

This test verifies that for not shared mappings with different AVs, merge is possible. It creates two mappings with a gap between them to force the kernel to assign different AVs to the mappings. It then joins the mappings by enlarging the first one so that the only remaining obstacle is the difference between AVs. There is a subvariant of the test for shared mappings that verifies that the merge does not happen for shared mappings.

3.1.4 Different AV and pgoff update merge tests

These tests check that for not shared mappings with different AVs, page offset update happens and can lead to a merge. They are basically a combination of the previous two tests.

3.1.5 Spacing test

The test creates mappings with spaces between them and then moves the mappings together. They should all merge into a single mapping if supported by the kernel.

3.1.6 Regroup test

This test creates two separate mappings (with different AVs) with a space between them. It then tries to move the second mapping next to the first one to merge them.

3.2 Analytic helper programs

Scripts and programs used for testing how the kernel works. They all reside in the `sources/analytic` folder in the attachment.

3.2.1 Print memory script

A simple shell script that prints the memory mappings of a process with the given name.

3.2.2 Process memory structure print

This program creates and prints pointers to different data types to examine the process memory structure.

3.2.3 VMA, AV and AVC print

The root process is forked and then its child is forked again, creating parent, child and grandchild processes. The parent, child and grandchild VMAs are printed including their supporting structures - AVs and AVCs. Printing is done using patched `mremap()` call that, if used correctly, only prints the information and does not actually affect the mappings in any way. The patch file is named 0001-Printing.patch.

3.3 Other tests

These tests cover corner cases including page offset update not necessarily linked to a merge, merging of partially shared mappings and merging of incompatible predecessor and successor mappings. They are again included in the patches as kernel self-tests.

3.3.1 Page offset update test

This test checks that for not shared mappings, page offset update happens even when a merge is not possible for other reasons. The test creates a mapping and unmaps its middle part, creating two separate mappings (with an identical AV) with a space between them. It changes the memory protection of the second mapping and then moves the second mapping next to the first one. The page offset should be updated but no merge will happen, because the memory protection does not match. To check that the page offset update actually happened, the memory protection is unified to allow a merge. If the page offset update did not happen, the merge will not happen. There is a subvariant of the test for shared mappings that verifies that the page offset update does not happen for shared mappings.

3.3.2 Page offset and AV update partially shared merge test

The test proves that even shared memory areas can be merged, provided they are merged with a non-shared memory area in the correct order. The test creates a mapping and forks to make it a COW shared mapping. It then creates another mapping that is not shared and moves it next to the shared one. The merge should be possible, because only the moved mapping cannot be shared.

3.3.3 Filling a gap between incompatible predecessor and successor

This test creates two blocks with a gap in between and forks to create a parent-child relation where memory is shared. Then it creates a new block that can fill the gap. The block is created elsewhere and is moved into the gap after the new block has been faulted. Merge should be possible only with one of the blocks, namely the predecessor, not both of them. This happens because shared memory between two processes makes it impossible to merge predecessor and successor, which have different AVs.

4. Results

This chapter summarizes performance improvements and successful merge count results and also elaborates the advantages and limitations of the changes.

4.1 Software overview

This section contains an overview of software needed for most of the measurements.

4.1.1 ftrace

Tracing framework `ftrace` listens to trace calls inside the Linux kernel. Specific trace calls can be exposed e.g. by the `trace-cmd` command that interacts with the `ftrace` tracer itself in the running kernel.

4.1.2 Running benchmarks

All scripts and other files needed to measure the results are located in the attachment's `sources/benchmark` directory. Most important is the `sources/benchmark/README` file which contains step by step instructions to run the benchmarks. In order to measure the performance and merges for both modified and mainline versions, there are two sets of patches located in the `sources/benchmark/patches` that both enable tracing. For a simple demonstration there is also a prepared virtual machine image, it is referenced from the `sources/README` file.

4.2 Device overview

Device overview describes devices that are later referenced in individual tests. There are always two versions of the kernel used for measuring results. One is a mainline version and the other is the same version with applied patches. Specific versions for specific devices can be seen in Table 4.1.

Name	Orion	Apollo
Motherboard	ASROCK B450M Pro4	Dell Inspiron 15R 5521
CPU	AMD Ryzen 5 3600 (6 cores)	Intel Core i7-3517U (2 cores)
GPU	NVIDIA T600 4GB GDDR6	AMD Radeon HD 8730M
RAM	G.Skill Aegis 2x16GB 3200 MHz	Samsung SODIMM 2x4GB 1600MHz
Disk	WD Blue SN570 1TB	Samsung SSD 850 EVO 500GB
OS	Gentoo Linux	Gentoo Linux
Kernel	v5.18.10	v5.18.0

Table 4.1: Device overview.

4.3 Performance and merge success rate results

These tests measure performance and number of successful merges compared to mainline version. Results are used to calculate confidence intervals using confidence level 95 % and presuming normal distribution. All measured times are in seconds.

4.3.1 New merge opportunities

As the following benchmark results show, there can be quite a lot of new merges when it is possible to merge two VMAs with different AVs, even if only for the simpler case where physical pages are not shared. Originally, the motivation was to increase the number of merges during `mremap()`, specifically in the `copy_vma()` function, but the introduced changes enable more merges even for `mmap()`, `mprotect()` and other system calls modifying mapping's flags.

In the case of `mmap()`, opportunity arises when a newly created VMA fills a hole between two other VMAs. The newly created VMA itself has not been faulted and therefore has no AV linked to it, but the two neighbouring VMAs might be linked to different AVs and hence we get a merge situation involving two different AVs.

In the case of `mremap()`, new merge can happen when an already-existing VMA is moved next to or between two other VMAs. This merge situation involves up to three different VMAs with up to three different AVs.

In the case of `mprotect()`, `madvise()`, `mlock()` and `mbind()`, the opportunity arises when an already-existing VMA's protection is modified and merging with one or two neighbouring VMAs is possible. This merge situation involves up to three different VMAs with up to three different AVs.

In the case of `brk()` syscall, new merge can happen when we decide to expand an already-existing mapping during `do_brk_flags()` and call `vma_merge()` to do so. If the expansion causes the expanded mapping to become adjacent to another VMA, we again get a merge situation with possibly two different AVs.

4.3.2 Spacing speed test

This test performs VMA merging for VMAs linked to different or the same AVs. It creates memory mappings with spaces between them and then moves the mappings together. The mappings will merge into one big mapping if it is supported by the kernel. Only the actual moving and merging of the mappings is measured, the creation of the mappings and spaces is not. The Spacing speed test takes two arguments, the number of mappings to create and if they should have different or the same AV. The results are displayed in Table 4.2 and Table 4.3.

platform	AV	times	CI (95 %)	diff m.	CI (95 %)
Apollo	same	5.51s, 5.38s, 5.50s	5.40–5.53	0, 0, 0	0
p. Apollo	same	2.57s, 2.64s, 2.65s	2.58–2.66	0, 0, 0	0
Apollo	diff	5.53s, 5.52s, 5.52s	5.52–5.53	0, 0, 0	0
p. Apollo	diff	2.53s, 2.52s, 2.55s	2.52–2.55	20000, 20000, 20000	20000
Orion	same	3.41s, 3.36s, 3.37s	3.36–3.40	0, 0, 0	0
p. Orion	same	1.65s, 1.66s, 1.67s	1.65–1.67	1, 0, 0	-0.20–0.87
Orion	diff	3.44s, 3.41s, 3.49s	3.41–3.48	0, 0, 0	0
p. Orion	diff	1.65s, 1.65s, 1.67s	1.65–1.67	20000, 20000, 20000	20000

Table 4.2: Results for Spacing speed test with 20000 mappings (part A)

platform	AV	pgoff m.	CI (95 %)	failed m.	CI (95 %)
Apollo	same	0, 0, 0	0	40018, 40018, 40018	40018
p. Apollo	same	19989, 19989, 19989	19989	0, 0, 0	0
Apollo	diff	0, 0, 0	0	40018, 40018, 40018	40018
p. Apollo	diff	19989, 19989, 19989	19989	0, 0, 0	0
Orion	same	0, 0, 0	0	40018, 40024, 40022	40019–40024
p. Orion	same	19996, 19992, 19992	19991–19996	1, 0, 0	-0.2–0.9
Orion	diff	0, 0, 0	0	40024, 40022, 40026	40022–40026
p. Orion	diff	19995, 19989, 19994	19990–19996	20000, 20000, 20000	20000

Table 4.3: Results for Spacing speed test with 20000 mappings (part B)

4.3.3 jemalloc stress tests

The `jemalloc` is a general purpose allocator trying to promote fragmentation avoidance and scalable concurrency support. It is used by the Firefox browser and others. In order to verifiably measure the number of merge counts, `jemalloc` self-tests were used.

The source files can be downloaded from the jemalloc's GitHub repository[1]. After unzipping the folder, continue with `./configure`, `make` and then e.g. `gmake tests_stress`.

Results

The results can be seen in Table 4.4 and Table 4.5. `Jemalloc` version 5.3.0 was used.

[1]Jemalloc releases - `https://github.com/jemalloc/jemalloc/releases`

test	platform	times	CI (95 %)	diff m.	CI (95 %)
unit	Apollo	238.9, 238.7, 238.4	238.4–238.9	0, 0, 0	0
unit	p. Apollo	240.0, 239.4, 239.1	239.1–239.9	460, 460, 460	460
integ.	Apollo	114.0, 114.3, 114.3	114.0–114.4	0, 0, 0	0
integ.	p. Apollo	114.6, 114.6, 114.9	114.5–114.9	310, 310, 310	310
analyze	Apollo	107.4, 107.6, 107.5	107.4–107.6	0, 0, 0	0
analyze	p. Apollo	107.8, 107.8, 107.7	107.7–107.8	274, 274, 274	274
stress	Apollo	212.4, 212.8, 213.0	212.5–213.0	0, 0, 0	0
stress	p. Apollo	213.6, 213.9, 213.4	213.4–213.9	530, 530, 530	530
unit	Orion	129.9, 130.2, 130.6	129.9–130.6	0, 0, 0	0
unit	p. Orion	131.3, 131.5, 131.2	131.2–131.5	487, 492, 498	487–497
integ.	Orion	61.5, 61.3, 61.2	61.2–61.5	0, 0, 0	0
integ.	p. Orion	61.9, 62.3, 61.6	61.6–62.3	317, 319, 322	317–322
analyze	Orion	57.7, 57.3, 58.0	57.3–58.0	0, 0, 0	0
analyze	p. Orion	58.1, 58.4, 58.5	58.1–58.5	274, 285, 293	275–293
stress	Orion	113.8, 115.0, 114.2	113.8–114.9	0, 0, 0	0
stress	p. Orion	115.7, 115.9, 115.5	115.5–115.9	546, 552, 563	546–562

Table 4.4: Stress tests results for jemalloc (part A)

test	platform	pgoff m.	CI (95 %)	failed m.	CI (95 %)
unit	Apollo	0, 0, 0	0	460, 460, 460	460
unit	p. Apollo	0, 0, 0	0	0, 0, 0	0
integration	Apollo	0, 0, 0	0	310, 310, 310	310
integration	p. Apollo	0, 0, 0	0	0, 0, 0	0
analyze	Apollo	0, 0, 0	0	274, 274, 274	274
analyze	p. Apollo	0, 0, 0	0	0, 0, 0	0
stress	Apollo	0, 0, 0	0	530, 530, 531	530–531
stress	p. Apollo	0, 0, 0	0	0, 0, 0	0
unit	Orion	0, 0, 0	0	1368, 1276, 1450	1284–1445
unit	p. Orion	352, 376, 439	347–431	67, 67, 74	66–73
integration	Orion	0, 0, 0	0	637, 645, 689	631–683
integration	p. Orion	168, 150, 178	152–178	35, 28, 30	28–34
analyze	Orion	0, 0, 0	0	624, 648, 664	626–664
analyze	p. Orion	164, 158, 183	156–180	24, 22, 28	22–27
stress	Orion	0, 0, 0	0	1308, 1264, 1290	1267–1308
stress	p. Orion	312, 306, 365	298–358	61, 44, 39	37–59

Table 4.5: Stress tests results for jemalloc (part B)

4.3.4 Redis

`Redis` is an open source in-memory database. As such, it works quite extensively with memory. `Redis` self-tests were used to measure the results. Running the test is quite easy, all you have to do is run `make` and then `./runtest`.

Results

The results are summarized in Table 4.6 and Table 4.7. `Redis` version 7.0.2, which was used to run the test, is available at Github[2].

platform	times	CI (95 %)	diff m.	CI (95 %)
Apollo	394.3, 398.8, 397.5	394.7–399.0	0, 0, 0	0
patched Apollo	395.6, 393.2, 393.0	392.6–395.3	2, 1, 2	1–2
Orion	498.8, 499.0, 499.1	498.8–499.1	0, 0, 0	0
patched Orion	499.0, 498.8, 498.8	498.8–499.0	1, 3, 1	1–3

Table 4.6: Stress tests results for Redis (part A)

platform	pgoff m.	CI (95 %)	failed m.	CI (95 %)
Apollo	0, 0, 0	0	41, 59, 44	39–57
patched Apollo	20, 19, 29	18–28	1, 0, 0	0–1
Orion	0, 0, 0	0	36, 34, 26	27–37
patched Orion	15, 13, 18	13–18	5, 7, 10	5–10

Table 4.7: Stress tests results for Redis (part B)

4.3.5 kcbench

The `kcbench` is a simple benchmark script that takes given kernel source files and compiles them several times using a different number of threads to thoroughly measure the compile times.

Results

The test is again run on patched and unpatched kernel, performance results are in Table 4.8 and merge count results are in Table 4.9 and Table 4.10. It is also important to specify the kernel version used as the input source files for the compilation. In our case it was version v5.18[3]. The used `kcbench` version v0.9.5 is available at GitLab[4].

[2]Redis releases - `https://github.com/redis/redis/releases`

[3]Linus's git repository - `https://github.com/torvalds/linux/releases/tag/v5.18`

[4]Kcbench releases - `https://gitlab.com/knurd42/kcbench/-/releases`

threads	platform	measured times	CI (95 %)
2	Apollo	1080.49, 1080.25, 1080.13, 1080.62, 1080.62, 1081.31	1080.27–1080.87
2	patched Apollo	1090.04, 1090.09, 1090.72, 1090.76, 1089.80, 1089.68	1089.85–1090.52
4	Apollo	911.75, 911.48, 912.07, 911.27, 911.51, 912.30	911.44–912.02
4	patched Apollo	921.08, 920.67, 920.11, 920.67, 920.24, 919.52	919.98–920.78
6	Apollo	918.25, 918.55, 918.74, 918.19, 918.44, 918.56	913.30–918.61
6	patched Apollo	928.60, 927.92, 928.41, 927.44, 928.36, 928.31	927.86–928.48
6	Orion	180.29, 180.28, 181.05, 180.61, 181.34, 181.51	180.46–181.23
6	patched Orion	184.08, 183.81, 183.73, 183.84, 183.82, 183.42	183.63–183.94
9	Orion	156.60, 156.58, 156.98, 157.66, 157.36, 157.57	156.78–157.47
9	patched Orion	158.86, 158.83, 158.99, 158.90, 158.97, 158.86	158.85–158.95
12	Orion	137.59, 137.89, 138.06, 138.49, 138.97, 138.62	137.90–138.65
12	patched Orion	139.65, 139.42, 139.33, 139.58, 139.27, 139.36	139.33–139.54
15	Orion	138.72, 138.82, 139.51, 139.34, 139.23, 139.01	138.88–139.33
15	patched Orion	140.39, 140.50, 140.23, 140.33, 140.31, 140.13	140.22–140.41

Table 4.8: Performance results for kcbench.

platform	diff m.	CI (95 %)	pgoff m.	CI (95 %)
Apollo	0, 0, 0	0	0, 0, 0	0
p. Apollo	19964, 19964, 19964	19964	245, 245, 245	245
Orion	0, 0, 0	0	0, 0, 0	0
p. Orion	23900, 23896, 23897	23896–23900.0	347, 348, 344	344–348

Table 4.9: Kcbench merge results (part A)

platform	expand	CI (95 %)	failed m.	CI (95 %)
Apollo	0, 0, 0	0	20777, 20776, 20776	20776–20777
p. Apollo	7, 7, 7	7	98, 98, 98	98
Orion	0, 0, 0	0	25018, 25029, 25026	25019–25030
p. Orion	9, 9, 9	9	178, 179, 175	175–179

Table 4.10: Kcbench merge results (part B)

Comparison

The `kcbench` results are probably most indicative to compare the patched and unpatched versions and their performance. Upper and lower bounds of confidence intervals are used to calculate a possible minimal and maximal slow down. E.g. for a–b mainline CI and c–d patched CI the comparison is calculated as $\frac{c}{b} - \frac{d}{a}\%$. We can see that the patched kernel in this scenario is slower by 0.5–2 %, as can be seen in the Table 4.11.

platform	threads	mainline CI	patched CI	comparison
Apollo	2	1080.27–1080.87	1089.85–1090.52	100.831–100.949 %
Apollo	4	911.44–912.02	919.98–920.78	100.873–101.025 %
Apollo	6	913.30–918.61	927.86–928.48	101.007–101.662 %
Orion	6	180.46–181.23	183.63–183.94	101.324–101.928 %
Orion	9	156.78–157.47	158.85–158.95	100.876–101.384 %
Orion	12	137.90–138.65	139.33–139.54	100.490–101.189 %
Orion	15	138.88–139.33	140.22–140.41	100.639–101.102 %

Table 4.11: Kcbench comparison.

4.4 Evaluation

As can be deducted from results above benchmark result from Apollo are much more compact than those from Orion. This is probably some interference from other applications running on Orion.

The benchmarks confirm that most of the cases where merging is not possible in the mainline kernel are resolved by this patch series, solving more than 90% of the cases. It also turns out that allowing different AV merging alone (not in combination with page offset update) is responsible for the vast majority of the newly possible merges. The changes have a small impact on the performance.

However, each workload behaves differently and even small changes in benchmarked application or choosing different application at all might produce different results. Developers usually do not deal with memory mappings alignment directly and therefore most of the merge opportunities depend on interactions between compilers, libraries and the kernel itself.

4.5 Advantages

In the current kernel, merges of anonymous mappings are mostly limited to VMAs that are either newly created (and therefore it is practically just an expand of an already existing area) or those that are created standalone but not yet faulted and therefore easier to merge. This basically means that any VMA that has been written to and then moved can never merge with anything because the page offset will not match its virtual address. And additionally, even if the page offset is correct, the merge can fail because of the AVs not being identical for both VMAs.

All of the above has been solved in this thesis and implemented in the attached code. This means that the number of VMAs is decreased and the virtual memory is more compact, which simplifies certain operations that must traverse the VMAs. There is also a limited number of VMAs that can exist for one process, which is defined as `DEFAULT_MAX_MAP_COUNT` and is almost equal (with some margin) to unsigned short max value, which is 65535. This is a lot of mappings and for most processes with common memory allocation patterns it is practically unlimited. A typical process usually stores its data in heap and stack which amounts for two quite big areas and maybe a few file mappings. But it is not that hard to waste all the mappings if you want to or if this is a side effect of your code. In fact, with the smallest possible areas of 4096 bytes, it would only take around 255 MB of memory to reach the limit. That might seem like intentionally pushing kernel to its limits, but imagine you do not have just a few or tenths of gigabytes but rather terabytes of memory on some special-purpose server that relies on creating and moving VMAs, for example in a large database, then even with quite big VMAs the limit might run away pretty quickly.

Another added bonus is from a formal point of view. Anonymous VMAs have page offsets calculated as virtual address with removed page bits, this means shifting the virtual address by `PAGE_SHIFT`. When it is moved, the page offset in

the mainline kernel stays the same and does not match the address any more, although it still matches the VMA's page offset (also immutable) needed for Rmap walk (page 33). This is obviously wrong and also confusing not only for new users trying to find a bug or just simply trying to understand the code. And although cosmetic, even this can be viewed as added benefit.

4.6 Limitations

Updating page offset during `mremap()` costs some time. Also, if merge conditions are met and actual merge happens, this also takes some time and as a consequence the kernel as a whole can be a bit slower. On the other hand, this is slightly compensated by the decreased number of VMAs present, which speeds up some walks e.g. during Rmap walk (page 33).

Another limitation is the impossibility to merge VMAs that are shared among several processes. This is unfortunately a hard stop because pages have AV pointer stored in them and can therefore belong only to one AV. A merge would imply that the page could belong to two different AVs for two different processes, which is impossible. It could be solved by duplicating the physical page itself, but that would waste memory and time in case of private COW pages and actually break the shared concept for shared pages.

Conclusion

First, we analysed and described the kernel itself, the problems preventing successful merges for anonymous mappings and all related kernel mechanisms. Next, we determined which functions needed to be modified in order to enable new merges in most of the problematic cases and successfully created a series of patches that solve the individual problems. Finally, the whole solution was tested to verify the merges actually happen when they should happen and speed impact of the changes on the kernel was measured.

Reception

Judging from the limited responses in the Linux kernel mailing list, it seems that improving merge success rate for anonymous mappings does not have very strong support among developers. This is probably because until now, when a process moved an anonymous mapping, there was not a chance that it could ever merge and still the system as a whole worked well, on top of that new code means added complexity and a risk of introducing bugs into the kernel. In most cases, the unmerged mappings did not cause any serious problems and therefore the inclusion of the patch series as-is in the mainline is improbable, but a chance stands in form of adding this as a conditional feature via e.g. the `madvise()` flag. However, a smaller part of the changes, namely (Refactoring vma_merge() (page 49) and Allowing merge during mremap() in-place expansion (page 55), are already in the process of being merged into the mainline kernel.

Possible users

The original task was to verify how much better the merging can get, which was motivated by an undisclosed proprietary workload and it will naturally take longer before its developers will test this solution.

Future extension

This section contains a list of a few possible extensions dealing with memory mappings that arose during the work on this thesis.

Dealing with child and parent processes

This work solves the vast majority of cases where the merge did not happen previously, however, there is still some space for improvements. The shared processes and respectively their mappings cannot merge in every case. This could be theoretically solved by forcing COW mechanism to copy the shared pages and therefore remove this obstacle.

Enabling multi mremap

One of the problems related to unsuccessful merges is that the `mremap()` system call can only work with a single mapping at a time. When merging does not work

as it should, then potentially a single mapping is fragmented into several blocks and if someone wants to move this whole range to another address, he/she has to do it separately for each mapping. This could potentially be solved by adding a new feature to `mremap()`, which would move several mappings at once. This thesis resolves many of the unsuccessful merge cases, however, some of them still remain in the case of shared mappings.

Merge collector

In the mainline kernel, merge is attempted when a mapping is modified and there is a chance it might be possible to merge it. But there are many obstacles preventing merges or making it difficult to check all the conditions necessary like swapping, migration or transparent huge pages. This could possibly be all avoided if the merges did not need to happen immediately after the mapping is changed, but rather at some time in the future. Such a mechanism would be very similar to garbage collectors, however, this change would require significant changes to the kernel merge code and would take a lot of time and effort to debug, validate and persuade the kernel developer community.

Overlapping mremap()

This feature would enable `mremap()` to move a mapping even when the new location would overlap the old one. If this was possible, more flexible operations with mappings would allow users to better organize the mappings and could enable additional merges. This is currently not allowed, because it would require a more complicated approach and probably also a helper memory location, but it is very well possible.

Additional sources

Linux kernel code, comments and Git commit messages[5]
Wikipedia[6]

[5] community [2022a]
[6] community [2022b]

Bibliography

Kernel community. Linux kernel code. https://www.kernel.org/, 2022a.

Wikipedia community. Wikipedia articles. https://www.wikipedia.org/, 2022b.

Jonathan Corbet. The case of the overly anonymous anon_vma. https://lwn.net/Articles/383162/, 2010.

Jonathan Corbet. An introduction to compound pages. https://lwn.net/Articles/619514/, 2014.

Jonathan Corbet. Five-level page tables. https://lwn.net/Articles/717293/, 2017.

Jonathan Corbet. Clarifying memory management with page folios. https://lwn.net/Articles/849538/, 2021.

Gustavo Duarte. Anatomy of a program in memory. https://manybutfinite.com/post/anatomy-of-a-program-in-memory/, 2009a.

Gustavo Duarte. How the kernel manages your memory. https://manybutfinite.com/post/how-the-kernel-manages-your-memory/, 2009b.

William Frazier. Copy on write. https://lwn.net/Articles/717293/, 2022.

Mel Gorman. Understanding the linux virtual memory manager. https://www.kernel.org/doc/gorman/html/understand/understand006.html, 2004a.

Mel Gorman. Understanding the linux virtual memory manager. https://www.kernel.org/doc/gorman/html/understand/understand014.html, 2004b.

Grant Seltzer Richman. Basic guide to linux mailing lists. https://www.grant.pizza/blog/mailing-list-guide/, 2021.

Wenbo Zhang. Linux kernel vs. memory fragmentation (part i). https://en.pingcap.com/blog/linux-kernel-vs-memory-fragmentation-1/, 2021.

List of Figures

List of Tables

Glossary

AV Anonymous VMA structure. 5, 6, 26, 27, 33, 35, 38, 40, 47, 57, 61

brk brk system call is used for extension of process's heap segment. 14

COW Copy on write. 6, 16, 21, 26, 40, 69

GitLab Provider of internet hosting for software development. 11

goto Unconditional jump to a predefined location in the code. 11

grep Grep command used for finding pattern in a text source. 9

KSM Kernel Samepage merging. 20

malloc malloc is basic system call for acquiring memory. 14

OS Operating system. 4, 8

VMA Virtual memory area. 4, 5, 16, 22, 24, 26, 33, 61

A. Attachments

A.1 First Attachment

Attachment of this thesis are all the source codes described in this project as well as electronic version of this thesis.