Design Space Exploration and Implementation of Efficient Memory Snapshots

A Thesis
Submitted in Partial Fulfilment of the Requirements
for the Degree of
Master of Science (M.Sc.)

At
Regensburg University of Applied Sciences

Student Name: Mario Mintel
Student Number: 3238103

Primary Supervisor: Prof. Dr. rer. nat. Wolfgang Mauerer
Secondary Supervisor: Prof. Dr. rer. nat. Jan Dünnweber

Submission Date: 8th August 2022

Abstract

Modern operating systems leverage the copy-on-write technique to efficiently manage their memory resources. Copy-on-write can significantly reduce the demand on the memory management system, avoiding copies of entire data blocks whenever memory should be duplicated. Instead, the process is delayed until a modification is made. The Linux kernel applies copy-on-write in its fork() system call. There are also many other appearances of copy-on-write, for example, for persistence in database systems, for snapshots of file systems or even for reducing memory usage in interpreted languages.

However, even though there are many use cases of copy-on-write, the Linux kernel does not provide a dedicated interface to create such mappings. Either, the application developer has to implement a copy-on-write concept, or as many applications do, the application exploits fork() for its copy-on-write. As a last resort many applications even modify their system's kernel. This work proposes a solution to the lack of such a dedicated mechanism by extending the mremap() system call with a new flag.

This new flag lets users of mremap() create a new copy-on-write protected memory mapping from an existing one. Thus, allowing users of mremap() to only snapshot selected data instead of all data. Additionally, this work also explores and evaluates existing mechanisms in the field of memory snapshots based on quantitative methods. The results show that an inclusion in the Linux kernel can also discourage applications from modifying the kernel, perhaps introducing severe security risks. Therefore, the extension of the Linux kernel with a new mremap() flag enhances the operating system by providing a dedicated interface for users to create efficient page-granular memory snapshots.

Kurzfassung

Moderne Betriebssysteme nutzen verzögerte Initialisierung (Copy-on-Write), um ihre Speicherressourcen effizient zu verwalten. Verzögerte Initialisierung kann die Belastung eines Speicherverwaltungssystem erheblich reduzieren, indem Kopien ganzer Datenblöcke vermieden werden, sobald Speicher dupliziert werden soll. Stattdessen wird der Vorgang verzögert, bis eine Änderung vorgenommen wird. Der Linux-Kernel wendet das Konzept der verzögerten Initialisierung in seinem fork() Systemaufruf an. Darüber hinaus gibt es auch viele andere Anwendungen die verzögerte Initialisierung nutzen, z. B. für die Persistenz in Datenbanksystemen, für eine Speichermomentaufnahmen von Dateisystemen oder sogar für die Reduzierung des Speicherverbrauchs in interpretierten Sprachen.

Obwohl es viele Anwendungsfälle für verzögerte Initialisierung gibt, bietet der Linux-Kernel keine spezielle Schnittstelle zur Erstellung solcher Speicherregionen. Entweder muss der Anwendungsentwickler ein eigenes Konzept der verzögerten Initialisierung implementieren, oder, wie es viele Anwendungen tun, die Anwendung nutzt den Systemaufruf fork() für seine verzögerte Initialisierung aus. Andere Anwendungen modifizieren sogar den Kernel ihres Betriebssystems. Diese Arbeit präsentiert eine Lösung für den Mangel eines solchen speziellen Mechanismus vor, indem sie den Systemaufruf mremap() um ein neues Markierungsbit erweitert.

Mit diesem neuen Markierungsbit können Benutzer von mremap() eine neue, mit verzögerter Initialisierung ausgestattete Speicherregion, aus einer bestehenden erstellen. Auf diese Weise können Benutzer von mremap() nur ausgewählte Daten anstatt aller Daten in einer Speichermomentaufnahme sichern. Darüber hinaus werden in dieser Arbeit bestehende Mechanismen im Bereich der Speichermomentaufnahmen mit Hilfe von quantitativen Methoden untersucht und evaluiert. Die Ergebnisse zeigen, dass eine Erweiterung des Linux-Kernels auch Anwendungen davon abhalten kann, den Kernel zu modifizieren, was möglicherweise ernsthafte Sicherheitsrisiken mit sich bringt. Daher verbessert die Erweiterung um ein neues Markierungsbit im Systemaufruf mremap() das Linux Betriebssystem, indem es den Benutzern ermöglicht, effiziente seitengranulare Speichermomentaufnahmen zu erstellen.

Contents

1 Introduction

Operating systems strive to abstract the complexity of a system's hardware by acting as an intermediary moderator between applications and computer hardware. Historically, computers were built to perform a series of single tasks, like a calculator. This constraint is lifted by dividing available processor time between multiple processes. The task of an operating system is, among other things, to provide interfaces for hardware functions such as input and output, and memory allocation, while also supporting multi-tasking. Operating systems did not exist in their modern and more complex forms until the early 1960s [1]. However, their impact on our modern world is undeniable. Today, operating systems are found on many electrical processing devices—from mobile phones and video game consoles to web servers and supercomputers.

The Linux kernel is one of the most famous operating systems. Although estimates suggest that Linux is used on only 2.43% of all desktop (or laptop) PCs [2], it has been widely adopted by servers and embedded systems like mobile phones [3]. Linux is developed as an open source project. This means, the source code is available for study and modification, which is one of the reasons for the public interest on the operating system. Another aspect of open source projects is that everyone can propose changes.

This work proposes an extension of the existing system call `mremap()`. Traditionally, the system call is used to either resize an existing memory mapping or move it to another location in its process's virtual memory. However, this work proposes the extension of the system call with a new flag called `MREMAP_COW`. The `MREMAP_COW` flag instructs the Linux kernel to create a new memory mapping backed by an existing one. Moreover, this new memory mapping is created using the copy-on-write (COW) resource management technique.

The COW resource management technique is used to efficiently implement a copy operation on a modifiable resource. If a resource is requested to be copied, but is not modified, it is sufficient to simply share the resource between the copy and the original. Whenever the copy or the original modifies the shared resource, a *real* copy must be created. The copy operation is therefore deferred until the first write access to one of the two memory regions. With COW, the resource consumption of unmodified resources is significantly reduced, while adding a small overhead to the write operations.

The Linux kernel already implements the COW technique in the `fork()` system call. The system call is used to create new processes. Typically, new processes are spawned to execute new programs. As processes reside in-memory, a process created through the use of `fork()` replaces their entire address space when loading a new program. Thus, it would be wasteful to copy all of a process's memory during a `fork()`. Instead, the COW technique is applied.

However, even though the Linux kernel implements COW in its `fork()` system call, it does not offer a dedicated interface to create such a mapping. Nevertheless, many applications [4–10] reasonably want to take advantage of this technique. Some of these applications even implement their own kernel modification to achieve the goal of creating a COW mapping. Others exploit the `fork()` system call for its implemented COW technique. This introduces an overhead, since `fork()` snapshots the whole address room. A dedicated mechanism like the proposed extension of `mremap()` can instead address the memory mapping directly. This allows for a page-granular snapshot mechanism. The results of this work show that the extension of `mremap()` does not only provide a dedicated alternative, but can also improve performance and prevent faulty implementations.

In summary, this work aims to address the following two research questions in the field of open source operating systems:

- How do applications that require COW semantics overcome the lack of a dedicated COW mechanism in the Linux kernel, and what mechanisms do they use?
- How does the proposed `MREMAP_COW` flag enhance the Linux kernel compared to the investigated mechanisms, based on a quantitative analysis?

Structure

The remainder of this work is structured as follows: Chapter 2 describes how this work relates to prior research and the motivation of the thesis. Chapter 3 gives an overview of virtual memory and further describes the internals of the Linux kernel, ranging from data structures to system calls. Chapter 4 provides an overview of different possibilities to implement COW mechanisms in the Linux operating system. Chapter 5 shows the results of the measurements made with these COW mechanisms. Chapter 6 then discusses these results and their consequences. Finally, Chapter 7 concludes this work.

2 Motivation & Related Work

COW is a resource management technique that has been a topic of extensive prior research across multiple fields, such as operating systems [11–13], interpreted programming languages [14, 15], file systems [6, 7], key-value stores [4, 9] and database systems [5, 8, 10]. It is applied in various layers ranging from kernel space implementations to user space implementations.

In [11] Smith *et al.* describe how the implementation of COW reduces the real time required to perform a Unix `fork()`. The implementation uses advanced paging strategies to perform address space inheritance. The authors show that for large processes, the time required to perform a Unix `fork()` is proportional to the fraction of write references, so that a child process which updates half (0.5) of its address space will spend half the time that another process updating all (1.0) of its address space will. Thus, by applying COW to `fork()` a reduction of 50 percent or more of the system time devoted to copying data can be achieved, depending on the fraction of write references. The Linux kernel, for example, implements COW in its `fork()` system call.

Another application of COW is proposed by Kemper *et al.* In [8] they report that by using `fork()` and its inherited COW they implement persistent snapshotting of their in-memory database system. They also use COW to implement simultaneous Online Transaction Processing and Online Analytical Processing.

There are also commercial products that implement COW to guarantee efficient usage of resources. Microsoft SQL Server uses COW to create database snapshots [5]. Redis [4] also uses `fork()` to implement its persistence [16].

However, the intended semantic of a `fork()` is to create new processes, not to share data. In [15] Korb *et al.* propose a new system call named `mmapcopy()` that offers an alternative to directly create a COW mapping. The authors use this system call in the interpreted programming language R to gain significant improvements in CPU time compared to Kernel Samepage Merging. Korb *et al.* report that their method increases memory savings by up to 11.7pp compared to the generic approach of Kernel Samepage Merging, but outperforms it on runtime.

Another alternative to `fork()` is provided by Sharma *et al.* In [10] they describe `vmcopy()`, a new system call that directly creates a COW mapping. They implement a multi-version concurrency control database system through the extensive use of COW. Multiple snapshots of different points in time are created by using `vmcopy()`.

This work contributes to existing research in the field of the Linux operating system by proposing the extension of the `mremap()` system call with the `MREMAP_COW` flag in several ways. First, it offers an alternative to applications exploiting `fork()` for its inherited COW cap-

abilities. The intent when calling `fork()` in the presented examples is not to create a new process, it is to create a COW mapping. Since these cases can easily be emulated by a thread and a mechanism to explicitly create a COW mapping, the use of `fork()` in these applications can be described as exploitative. This also contradicts one of the Unix philosophies, which emphasises to "make each program do one thing well" [17] (*e.g.*, create processes).

Another contribution of this work is solving the problem that dedicated COW mechanisms require a custom kernel modification. As lined out before, multiple implementations exist to solve the problem of explicitly creating a COW mapping. Such kernel modifications can induce severe security issues. By extending the Linux kernel with a dedicated mechanism to create COW mappings, these security issues are limited to a single implementation. Introducing the changes to the upstream Linux kernel can prevent faulted custom implementations, while also removing the need of a custom kernel.

Finally, this work also contributes to the existing research in the field of the Linux operating system by evaluating the proposed mechanism and other related mechanisms. Unit tests are run for each implemented mechanism to verify their correctness. Different measurements are made to compare runtime and effectiveness of each mechanism. These units are then used to evaluate the usability of each individual implementation.

3 Virtual Memory Management

Today, almost all computer systems—desktops, tablets, wearables and often even embedded systems—rely on a multi-layer storage concept. The first layer consists of the main memory, while the second layer consists of auxiliary storage media such as hard disks. One or more processors have direct access to the main memory, but not to the auxiliary memory; therefore information may only be processed in the main memory, and information that is not being processed can reside in the auxiliary storage media. This approach offers several advantages: (1) the ability to store data non-volatilely in auxiliary storage; (2) the ability to access data in main memory with low latency; (3) the ability to use more space than actually available in main memory.

Although it has some very important advantages, the problem of how to relocate the data properly between main memory and auxiliary storage arises. The operating system solves this problem by applying a memory management concept that allows for an idealised abstraction of the storage resources, called *Virtual Memory (VM)* [18]. This chapter will introduce the key concepts of VM. Additionally, Chapter 3.2 provides an overview of the data structures and concepts applied by the Linux kernel.

The concept of VM lets the user access memory as if it was one piece of large main memory. To do this, it is obligatory to use a set of addresses different from that provided by the physical memory and to provide a mechanism that translates between them. Such an address is called a *virtual address* and the set of these addresses is called the *virtual address space*. Similarly, the physical memory uses a *physical address*, thus, the set of physical addresses is called the *physical address space*. Note that, by using this abstraction, we can map an individual virtual address space to every process. For future reference, we denote the virtual address space by $N = \{0, 1, \ldots, n-1\}$ and the physical address space by $M = \{0, 1, \ldots, m-1\}$, assuming $n > m$ unless stated otherwise.

Since the virtual address space contains a collection of *potentially* usable virtual addresses, there is no requirement that every virtual address actually represents a location in the physical address space. This induces an increase in complexity in the addressing mechanism, as there is no a priori affinity between virtual addresses and physical addresses. Such an addressing mechanism can be described by a function $f\colon N \mapsto M \cup \{\emptyset\}$ such that for each moment in time

$$f(a) = \begin{cases} a' & \text{if } a \text{ is mapped in M at location } a', \\ \emptyset & \text{if } a \text{ is missing from M.} \end{cases}$$

This function f is known as the *address map* [19]. Thus, an addressing mechanism can resolve an address a to a' if there is a mapping in the address map f, and it can not if there is no mapping in the address map f.

Figure 1 depicts an example mapping of f, where an arrow (a, a') for a in N and a' in M indicates that information a is stored in location a', while the absence of an arrow indicates that information a is not present in M.

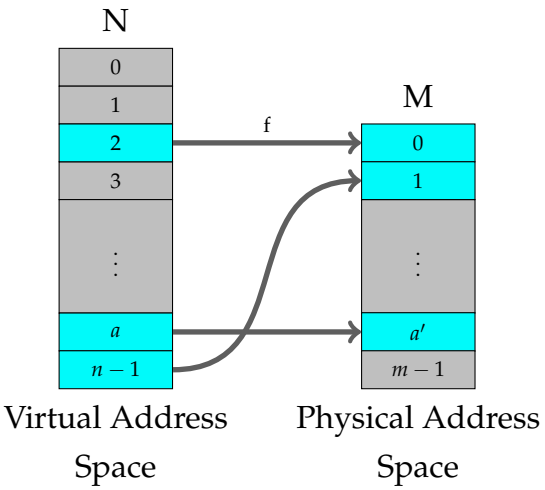


Figure 1: Mapping from virtual address space to physical address space.

Because of the mapping f, the user may think that consecutive data in N is stored consecutively in M, despite the fact the data may actually be stored arbitrarily. This creates a contiguous memory when only considering the virtual address space.

The address map is implemented in a hardware component called the *memory management unit (MMU)*. The MMU is typically located within the computer's CPU but can also operate in a separate integrated chip [20]. When VM is used, all memory references are passed through the MMU, which maps virtual addresses to physical addresses, as illustrated in Figure 2.

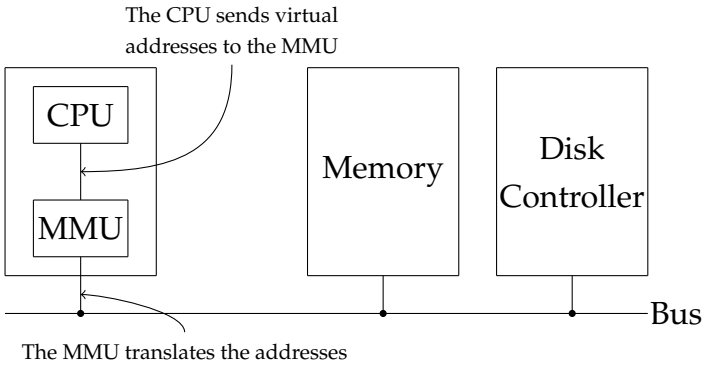


Figure 2: The position and function of the MMU [20].

3.1 Paging

To solve the problem of relocating data properly between main memory and auxiliary storage, most VM operating systems use a technique called *paging* [21]. With paging, the virtual address space consists of fixed-length units called *pages*. The corresponding units in the physical address space are called *page frames* [20].

Pages on contemporary systems are usually of at least 4 KiB in size; recent x86-64 processors, such as the AMD64 or the Intel 64 processors also support page sizes of 2 MiB and 1 GiB [22, 23], that are called *huge pages*. Pages are stored in a data structure known as the *page table* and can either be *paged in* or *paged out* depending on the available physical memory. When the physical memory is full and a new page is required, a particular page is chosen and replaced by the new page. The new page is paged in, while the old page is paged out, as depicted in Figure 3.

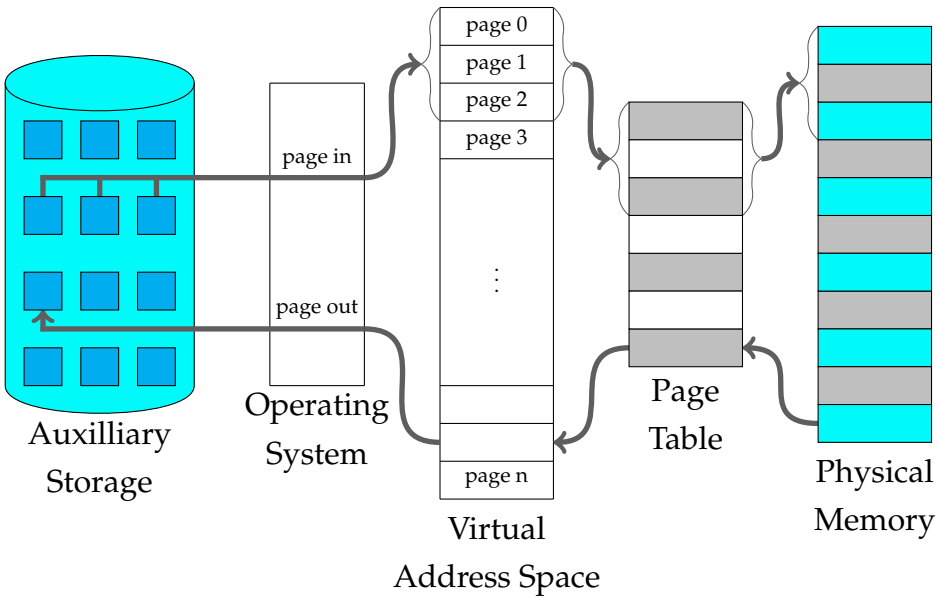


Figure 3: The operating system uses the concept of paging to swap data in and out of an auxiliary storage and the physical memory.

The page table maintains an overview of which pages currently reside in physical memory and which do not, by marking pages with a *present bit*. This present bit is set, if a page a has a mapping in f so that $f(a) = a'$. Therefore, if the present bit is not set—that means that the page is not present in physical memory—the MMU will detect this, since $f(a) = \emptyset$, and will cause the CPU to trap to the operating system. This trap is called a *page fault* [20]. To handle this page fault, the operating system follows a series of steps as illustrated in Figure 4.

First, the operating system checks whether or not the virtual address of the just occurred page fault refers to the virtual address space of the process. If it does not, it is an invalid

fault and the operating system will terminate the process. However, if the faulting virtual address is valid, that is it is part of the process's virtual address space, the operating system has to check if the page corresponds to a page frame, that is it is present in the physical memory. If it is not present, the operating system has to page in the appropriate page from the auxiliary storage. As access to the auxiliary storage takes a long time, the process has to wait until the page has been fetched. While doing so, the operating system can schedule other processes, until the page is fetched. When the page is fetched, it is written into an unused physical page frame and a corresponding entry to the process's page table is created. The trap is released and the process is restarted at the machine instruction where the page fault originally occurred. Although this time the MMU can translate without faulting and thus the process can continue to execute.

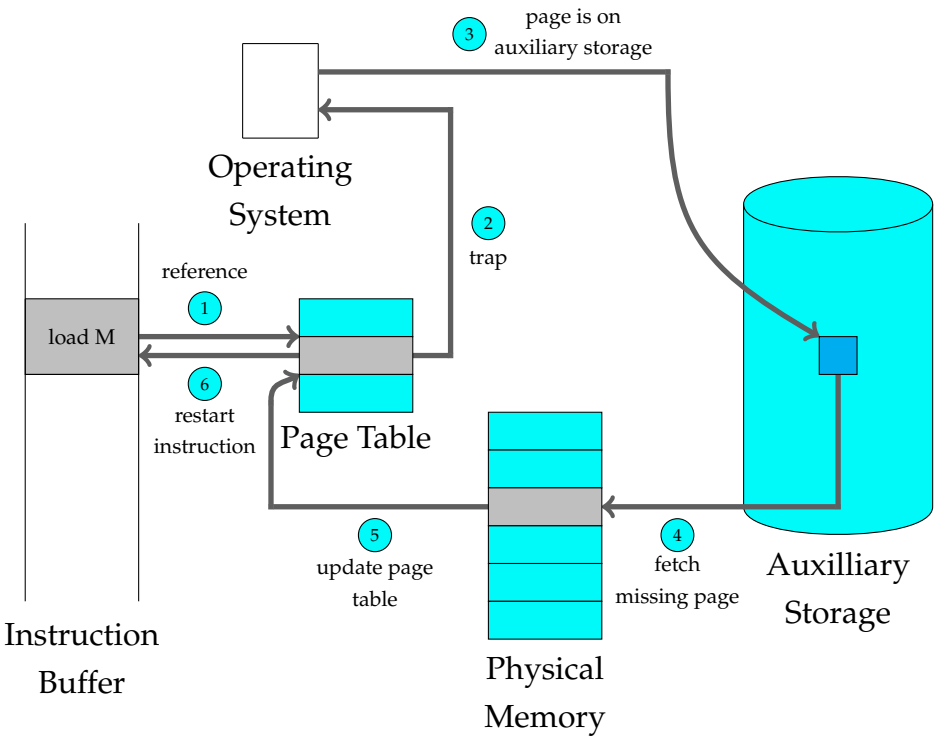


Figure 4: The operating system handles a missing page fault by fetching the page from the auxiliary storage. It then writes the fetched page into the physical memory and updates the page table. Finally, the instruction is restarted. [24].

However, if there are no free physical page frames left, the operating system has to swap pages in and out of the physical memory. To page out, the operating system has to pick a fitting page frame based on the implemented *page replacement algorithm* in order to persist the contents of the pages in the auxiliary storage. It then handles the page fault as described.

3.1.1 Page Tables

The page table is an array of *page table entries (PTEs)* that hold the mapping of a virtual page to its corresponding physical *page frame number (PFN)*. Such a PTE is highly machine dependent, but the information present is roughly the same from machine to machine. Note that, as each process has its own virtual address space, it also needs its own page table. Table 1 is an excerpt of the available bits and provides an overview of the present meta information in a PTE used by a x86-64 Linux system. These bits are defined in the header file `<asm/pgtable_types.h>`.

Table 1: Functional overview of Linux page table entry protection and status bits.

Bit	Function
`_PAGE_PRESENT`	Page is resident in memory and not swapped out.
`_PAGE_RW`	Set if the page may be written to.
`_PAGE_USER`	Set if the page is accessible from user space.
`_PAGE_ACCESSED`	Set if the page is accessed.
`_PAGE_DIRTY`	Set if the page is written to.
`_PAGE_PROTNONE`	Page is resident, but not accessible.

All these bits are self-explanatory except for the `_PAGE_PROTNONE`. This bit is set by protecting a region with `mprotect()` using the `PROT_NONE` flag. Protecting a page this way, clears the `_PAGE_PRESENT` bit and sets the `_PAGE_PROTNONE` bit. Linux can now enforce the inaccessibility, as the `_PAGE_PRESENT` bit is clear, and consequently an access of the page will raise a page fault. This makes the page inaccessible to user space, while still maintaining the knowledge that the page is resident if it needs to be swapped out or the process exits [21].

3.1.2 Optimisations

After acquiring a basic understanding of VM and paging, we can now have a detailed look at the optimisations of these concepts. Any paging system has to address the following two major difficulties:

- ❑ The translation from virtual address to physical address has to be fast.
- ❑ The virtual address space is proportional to the implemented management structures.

The first point is a consequence of the fact that every memory access has to be translated by the MMU. As every instruction has to be fetched from memory and most instructions reference data from memory, multiple page table translations have to be executed per instruction.

Let us assume that an instruction takes 1 ns, then the page table lookup must occur within 0.1 ns to prevent the mapping from becoming a bottleneck.

The second point is by virtue of the fact that modern computer systems typically use at least 32 bit architectures. Assuming a page size of 4 KiB, a 32 bit virtual address space has over one million ($2^{20} - 1$) pages. When using a 64 bit address space, a page table greater than 10^{15} is required. Also remember that each process has its own page table, thus making it unfeasible to simply use unoptimised page tables in a multi-process operating system.

The requirements of such a fast translation for large page tables is a significant constraint on the design of VM. The simplest approach is to have a single page table in main memory, with one entry for each virtual page, indexed by the virtual page number. The operating system now has two possibilities: (1) load every PTE of the page table into a hardware register; (2) only have a reference to the start of the page table in a hardware register.

The evident advantage of the first approach is that during process execution, no more memory references are needed for the page table. However, if the page table is large, this is unbearably expensive. Furthermore, at every context switch the operating system has to load the entire page table, which completely negates performance.

The second approach circumvents the need to load the entire page table at every context switch, as only a single register has to be reloaded. Nevertheless, the disadvantage of this approach is that during execution one or more memory references are required to read the page table, thus, introducing additional operations.

Let us now have a look at more sophisticated solutions to speed up paging and handle large virtual address spaces, starting with the former.

Translation Lookaside Buffer

As the page table typically resides in main memory, the design of paging has a huge impact on performance. Consider, for example, a 1 byte instruction that copies data from one register to another. Without paging, this instruction only has to fetch the instruction from memory, consequently only a single memory reference is needed. While with paging, we also have to reference the page table, thus, at least one additional memory reference is needed. Since the performance of modern computer systems is bound by the Von-Neumann bottleneck [25]—that is the limited throughput between the CPU and memory compared to the amount of memory—having to do twice as many references per memory reference, cuts performance in half. Under these conditions, using paging may never be considered. The key to improving access performance is to rely on locality of reference to the page table. When a translation for a virtual page number is used, it will probably be needed again in the near future, because the references to the words on that page have both temporal and spatial locality.

Accordingly, modern processors include a special cache for mapping virtual addresses to physical addresses without using the page table [26]. This cache, called (for historical reasons [27]) *Translation Lookaside Buffer (TLB)* is illustrated in Table 2 [28]. Typically, the TLB is part of the MMU and consists of a small number of entries, often between 16 and 512 [29], but eight in our example. Each entry in the TLB holds the mapping of a virtual page to its physical frame. Additionally, the TLB includes other status bits, such as the modified and protection bits, because the TLB is accessed *instead* of the page table on every reference.

Table 2: An excerpt of a TLB to speed up paging [20].

Valid	Virtual Page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RWX	14
1	861	1	RWX	75

Let us now see how the TLB functions. When a virtual address is presented to the MMU, the hardware first checks the TLB if its virtual page number is present. If a valid match is found, that is a so-called *TLB hit*, and the access does not violate the protection bits, the physical page frame is used to form the address, without using the page table. However, if the virtual address is not present in the TLB, a so-called *TLB miss* occurs. The MMU now determines if it is merely a TLB miss or also a page fault by doing an ordinary page table lookup. If the page is not present in memory, then the TLB miss indicates an actual page fault, which will be handled by the operating system as explained earlier. The MMU then evicts one of the entries from the TLB and replaces it with the PTE just looked up. Because the TLB has much less entries than the number of pages in main memory, TLB misses will be much more frequent than actual page faults [29].

As our more and more sophisticated paging concept increases in complexity, Figure 5 depicts an interim flowchart, which includes the previous optimisation strategies. The MMU first checks if the TLB holds the required PTE. If it does, then the MMU can generate the physical address. However, if it does not, then the page table has to be examined whether or not the page is present in physical memory. If the page is not in physical memory, a page fault is generated and handled by the operating system. With the page present in physical memory, the MMU updates the TLB accordingly and generates the correct physical address.

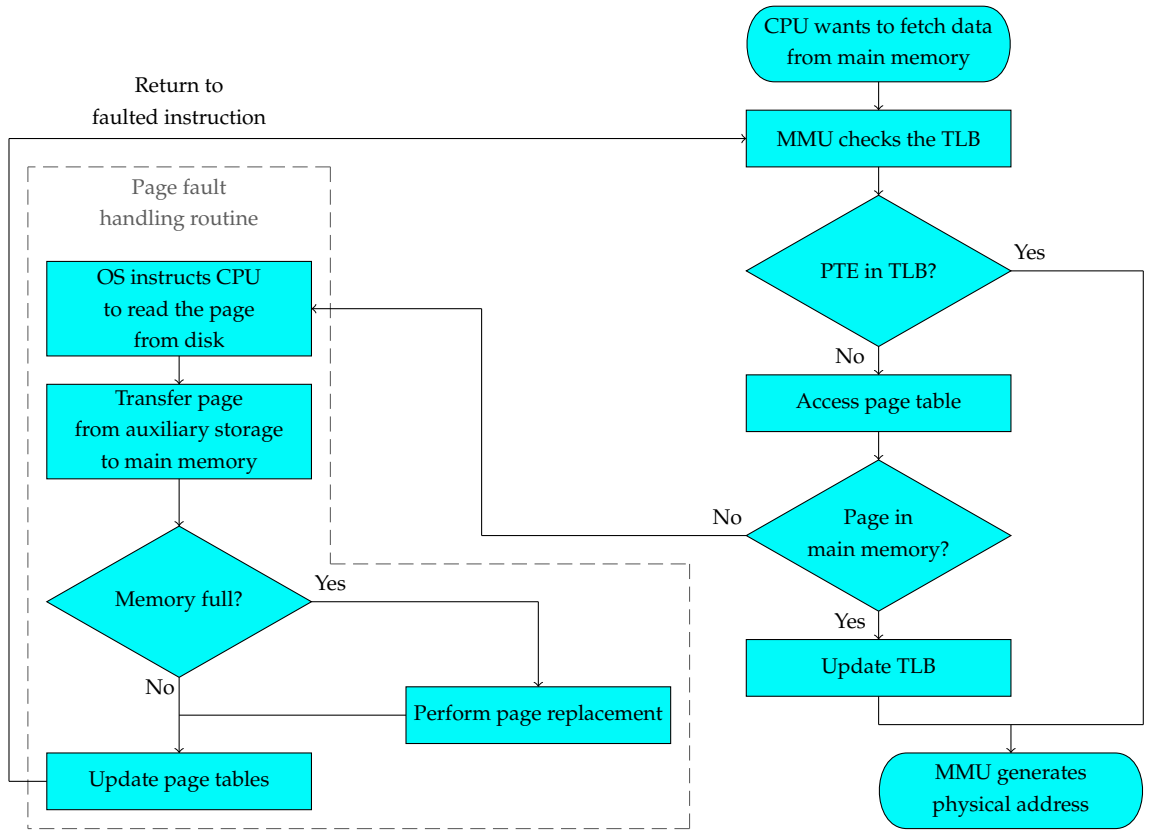


Figure 5: A flowchart depicting how paging uses a TLB.

Overall, the TLB increases the performance of paging by bypassing the page table lookup whenever possible and therefore, reducing the time needed to translate between virtual addresses and physical addresses. But that is not the only problem we have to address. Another problem is how to deal with very large virtual address spaces. Below we will discuss one way of dealing with them.

Multi-Level Page Tables

To tackle the issue of handling large virtual address spaces while also applying paging, the use of a *multi-level page table* can be considered [20]. Multi-level page tables avoid keeping all page tables in memory. Especially those that are not needed should not be kept around. This reduces the size of the required management structure.

Let us consider a 32 bit virtual address space. We want to be able to address every single byte, that means that there are 2^{32} addressable bytes or 4 GiB. Suppose, for example, that a process needs 12 MiB: the bottom 4 MiB of memory for code, the next 4 MiB of memory for data, and the top 4 MiB of memory for the stack. Between the bottom of the stack and the top of the data there is a huge hole that is unused.

Instead of using a single page table that holds every PTE, we can partition our 32 bit address as illustrated in Figure 6. This partition divides a 32 bit address into a 10 bit *PT1* field, a 10 bit *PT2*, and a 12 bit *Offset* field. Since we use an offset of 12 bits, the page size has to be 4 KiB, as every byte has to be addressable. This leaves us with 2^{20} available pages.

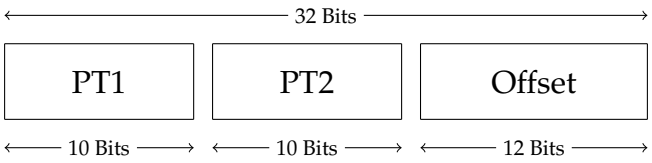


Figure 6: A 32 bit address partitioned with two page table fields.

In Figure 7 we can see how the two-level page table works using the just discussed partitioning. On the left side we see the top-level page table with 1024 entries. Each entry in the top-level page table basically represents a block of 4 MiB. This page table is indexed by the *PT1* field. The entries in the top-level page table reference the second-level page tables shown on the right side. The second-level page tables, that are represented by the *PT2* field, reference to the physical page frames. The other (shaded) entries are not used.

So when presented with a virtual address, the MMU first extracts the *PT1* field and uses this value to index into the top-level page table. The entry located by indexing into the top-level page table then yields the address of a second-level page table. The value of the *PT2* field is then used to index into the selected second-level page table to find the PFN for the page itself. Finally the *Offset* field is used to address each individual byte in the selected page.

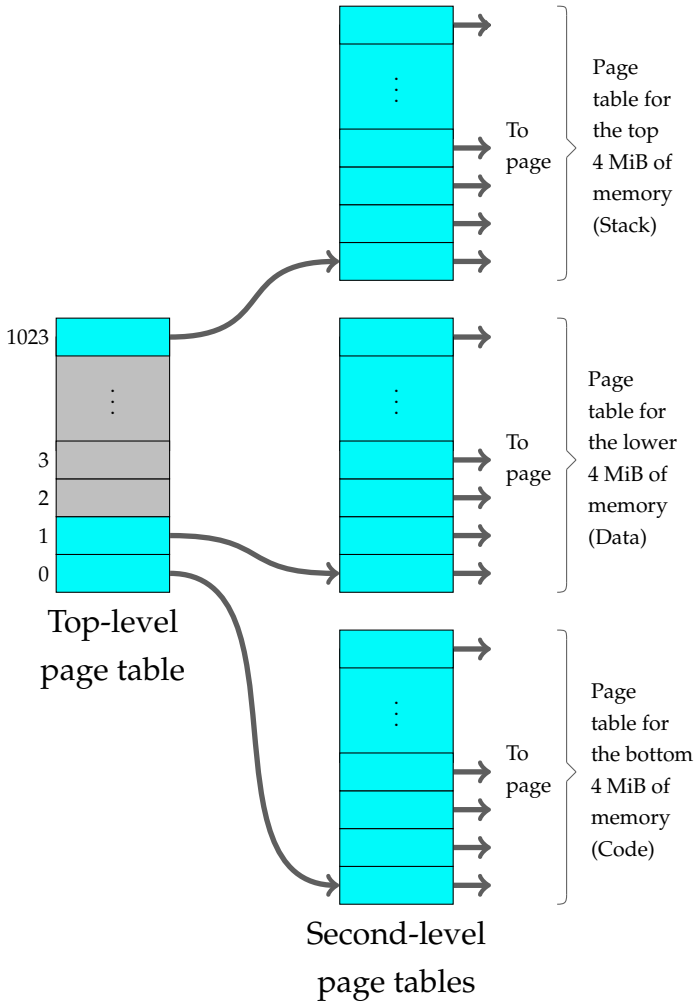


Figure 7: A two-level page table can be used to decrease the total size of the required page tables. Instead of using one page table for all pages, the top-level page table does only index the second-level page tables. The second-level page table then addresses the individual pages. In this example 12 MiB are allocated for the program, using a total of 4 page tables.

As an example, let us consider the 32 bit address 0x00403316 (4,207,382 in decimal). This virtual address corresponds to *PT1* = 1, *PT2* = 3, and *Offset* = 790. The MMU first uses the *PT1* field to index into the top-level page table and obtains entry 1, which corresponds to 4 MiB to 8 MiB - 1 (*i.e.*, absolute addresses 4,194,304 to 8,338,607). It then uses the *PT2* field to index the second-level page table and obtains entry 3, which corresponds to addresses 12288 to 16383 within its 4 MiB block (*i.e.*, absolute addresses 4,206,592 to 4,210,687). This entry contains the PFN of the page containing virtual address 0x00403316. If this page is in memory, the PFN is combined with the offset to obtain the physical address. If not, a page fault is generated.

Note that, even though the virtual address space spans over a million pages, with multi-level page tables we only need four page tables with 1024 entries: the top-level page table and the second-level page tables for 0 to 4 MiB (for the code), 4 MiB to 8 MiB (for the data) and the top 4 MiB (for the stack). Using this approach, any given number of virtual address space bits can be achieved by simply adding more and more levels. Linux, for example, uses a four-level page table [30], more details in Chapter 3.2.3.

By using multi-level page tables we reduce the number of page tables that have to be present in main memory, even if the virtual address space is large. There are also different approaches to handling large virtual address spaces, many of which can be found in Talluri *et al.* [31]

Demand Paging

Another VM optimisation strategy uses a design pattern called *lazy loading*. With lazy loading, data initialisation is deferred until the point at which it is needed. When this design pattern is applied, the memory is not allocated instantly. Instead, the definition of the allocation is simply stored until the memory is accessed. The access generates a page fault and it is during its handling, that the real mapping is made [32]. In practice, most real processes do not need all their pages, or at least not all at once, for several reasons [33]: (a) error handling code is not needed unless that specific error occurs, some of which are quite rare; (b) arrays are often oversized for worst-case scenarios; (c) certain data of certain programs is not always needed, as in querying only a portion of a database. This approach of only loading pages as they are demanded is called *demand paging* [34].

Demand paging offers several advantages opposed to loading all pages immediately. First, there is more space in main memory as pages are only loaded when demanded. This means that more processes can be loaded, reducing context switching time which utilises large amounts of resources [35]. Second, less loading latency occurs at startup as less information has to be paged in from auxiliary storage. The overhead of paging is moved from program startup to the first program execution, decreasing the startup time of programs.

But demand paging also has its drawbacks. To begin, programs face an extra latency whenever they access a page for the first time, because the overhead is moved from startup to execution time. Another disadvantage is that it also induces possible security risks. Percival describes a timing attack that can be used to create a covert communication channel using demand paging [36]. Demand paging also slightly increases the complexity of memory management with page replacement algorithms. Such page replacement algorithms can, for example, account for the fact that demand paging changes the occurrence of page faults.

Page Replacement Algorithms

The implemented page replacement algorithm also has a great impact on paging performance. When a page fault occurs, the operating system has to chose which page will be evicted from main memory so that the incoming page can be written to main memory. Furthermore, if the page to be replaced in main memory has been modified, it must be updated on the auxiliary storage whenever it is paged out in order to not lose the modification. However, if the page has not been modified (*e.g.*, the executable of a process), no rewrite is needed.

Although it is possible to pick random pages to replace at each page fault, doing so will decrease system performance heavily. Consider a page that is heavily used, if this page gets replaced it will probably be paged in again quickly, resulting in extra overhead. The subject of page replacement algorithms has been addressed in multiple works, both theoretical and experimental [20, 24, 37]. Below, a short overview of the page replacement algorithm used by the Linux kernel is provided.

Least Recently Used Page Replacement Algorithm

Linux literature makes heavy mention of the *least recently used (LRU)* page replacement algorithm in the context of memory management [21, 24, 29]. However, Mel from the Linux-mm mailing list—when confronted with the question what page replacement algorithm is used—responded with: "The current reclaim algorithm is a mash of a number of different algorithms with a number of modifications for catching corner cases and various optimisations [38]".

The LRU in Linux uses two lists called `active_list` and `inactive_list`. The goal of this separation is simple: the `active_list` contains the *working set* [19] of all processes and the `inactive_list` contains the replacement candidates.

The lists are similar to a simplified LRU 2Q [39] where two lists called `Am` and `A1` are managed. The algorithm describes how the size of the two lists have to be tuned but Linux takes a simpler approach by using `refill_inactive()` to move pages from the bottom of `active_list` to `inactive_list` to keep the `active_list` about two thirds the size of the total page cache.

In summary, the algorithm does exhibit LRU-like behaviour and it has been shown by benchmarks to perform well in practice [21]. As discussing all corner cases and optimisations is out of scope for this work, we assume that Linux uses the LRU page replacement algorithm in order to further improve its paging performance.

Copy-on-Write

While *copy-on-write (COW)* is not an optimisation strategy to speed up paging and neither to handle large page tables, it is a general optimisation strategy when copies of a modifiable resource have to be created. It is also the key concept of this work, hence it is covered in this chapter.

When using VM, multiple virtual addresses can be mapped to the same physical address. Note that, our previously defined function f allows this, as we defined $n > m$ and there is no restraint on mapping the same m twice. This introduces several advantages, one of them being COW.

COW is a technique to efficiently implement a copy operation on a modifiable resource. If a resource is copied but not modified, it is not necessary to create a new resource; instead the resource can be shared between the origin and the copy. The copy operation is deferred until a modification is made. By sharing resources this way, the resource consumption of unmodified copies is significantly reduced.

The Linux kernel makes use of COW semantics to safe memory in its `fork` system call. Instead of copying the entire address space, the kernel just copies the page tables. The address space of the parent and the child process now both point to the same physical pages. Of Course, both processes may not modify each other's pages, which is why all PTEs in both page tables are marked as read-only, even though they could be written to in normal circumstances. As soon as one of the processes attempts to modify a copied page, a page fault occurs. The kernel then checks additional memory management structures (introduced later in Chapter 3.2) to check if the page can be accessed with read and write operations or only read operations. If only read operations are permitted, a *segmentation fault* is raised. However, if read and write operations are permitted although the PTE only indicates a read-only page, the kernel now recognises that it must be a COW page [40]. It therefore creates a new writable page assigned exclusively to the process. The other page stays read-only until only a single owner remains. Then the write-protection on the page is removed [41].

COW may also be used in other areas such as file systems or database servers. For example Btrfs and ZFS use a COW mechanism for creating snapshots of the file system [6, 7], while Microsoft SQL Server uses COW to create snapshots of their databases [5].

Kernel Samepage Merging

Kernel Samepage Merging (KSM) is a mechanism implemented in the Linux kernel, which is primarily used to share pages with identical contents [42]. However, unlike with `fork()` where the kernel knows that the contents of the pages in child and parent process must be identical, with KSM the kernel does not have such knowledge. KSM can be enabled by

`CONFIG_KSM=y` and is controlled by either `madvise()` or by directly accessing the sysfs interface via `/sys/kernel/mm/ksm/`. KSM uses two separate red-black trees (see Chapter 3.2), the *unstable tree* and the *stable tree*. Pages tracked by KSM are initially stored in the unstable tree; this means that KSM considers their contents to be volatile. Placement in the tree is determined by a simple `memcmp()` of the page's contents. Essentially, the content of the page is treated as a huge number. The unstable tree is suitable for finding pages with duplicate contents; a relatively quick traversal of the tree will turn up the only candidates.

However, KSM does not place every page it scans in the unstable tree. If the contents of a page change over the course of one memory scanning cycle, the page will not really be a good candidate for sharing anyway. So pages that change are not represented in the unstable tree. Each scan cycle the unstable tree is dumped and rebuilt from the beginning. That deals with the problem of pages which, as a result of modifications, find themselves in the wrong location in the tree.

The other pages which are not found in the unstable tree, are those which have actually been merged with duplicates. Those pages are put into a separate stable tree and are marked read-only. The stable tree is also a red-black tree, but it is not rebuilt regularly since pages cannot become misplaced there. Once a page goes into the stable tree, it stays there until all users have either modified or unmapped it [43].

To conclude, paging is a key concept of VM. It has a major impact on a system's performance, therefore it is optimised in many different ways. The most impactful optimisations include the TLB and multi-level page tables, as both tackle the main issues of paging: fast translation and large page tables, respectively. But there are also other optimisations for VM like demand paging, the implemented page replacement algorithm, COW or KSM. Demand paging lowers the amount of pages present in main memory, while the implemented page replacement algorithm decides which page is to be evicted whenever the main memory is full. With COW, resources can be shared efficiently by deferring the copy operation until a modification is done. KSM on the other hand is a memory-saving de-duplication feature, which can reduce the amount of present pages by merging pages with identical content. Most operating systems, including Linux [30] and Windows [44], implement these optimisations to guarantee the performance of their VM.

3.2 Linux Kernel Internals

In this chapter different specifics of the Linux kernel are introduced. These core-concepts help to understand how different COW mechanisms can be implemented in the Linux kernel, whether in the user space or the kernel space.

3.2.1 Memory Management

The Linux kernel defines many data structures for its memory management. Introducing all of them is out of scope for this work, so we focus on the relevant ones. We will start off with describing most of the defined fields and discuss their usage and interactions later. The two central data structures used in Linux's VM are the memory descriptor and the virtual memory area—let us start with the former.

Memory Descriptor

The kernel represents a process's address space using a data structure called the *memory descriptor*. This structure contains all the information regarding a process's address space. The memory descriptor data structure is defined as follows—in simplified form:

```
<linux/mm_types.h>
struct mm_struct {
        struct vm_area_struct   *mmap;
        struct rb_root          mm_rb;
...
        unsigned long           mmap_base;
        unsigned long           highest_vm_end;
...
        pgd_t *                 pgd;
        atomic_long_t           pgtables_bytes;
        atomic_t                tlb_flush_pending;
...
        spinlock_t              page_table_lock;
        struct rw_semaphore     mmap_lock;
...
        atomic_t                mm_users;
        atomic_t                mm_count;
...
        int                     map_count;
        unsigned long           total_vm;
        unsigned long           hiwater_rss;
        unsigned long           hiwater_vm;
...
        unsigned long           def_flags;
        unsigned long           flags;
        unsigned long           data_vm;
        unsigned long           exec_vm;
        unsigned long           stack_vm;
...
        unsigned long           start_code, end_code, start_data, end_data;
        unsigned long           start_brk, brk, start_stack;
        unsigned long           arg_start, arg_end, env_start, env_end;
...
};
```

Admittedly, even in its simplified form, the amount of information in this structure can be overwhelming. However, the data structure can be broken down into sections, as hinted by the ellipsis:

- `mmap` and `mm_rb`, link together all the virtual memory areas in a doubly linked list and a red-black tree structure, respectively.
- `mmap_base` indicates the base of the memory descriptors address space, while the field `highest_vm_end` indicates the end of it.
- `pgd` is a pointer to the page global directory. The `pgtables_bytes` field represents the allocated PTE page table pages. The `tlb_flush_pending` field indicates that an operation with batched TLB flushing is going on. Anything that can move process memory needs to flush the TLB when moving a `PROT_NONE` mapped page.
- `page_table_lock` and `mmap_lock` are locking mechanisms to prevent deadlocks or memory corruption when accessing the respective data structures.
- `mm_users` is the number of processes using this address space. For example, if two threads share this address space, `mm_users` is equal to two. The `mm_count` field is the primary reference count for the `mm_struct`. All `mm_users` equate to one increment of `mm_count`.
- `map_count` is the number of virtual memory areas and `total_vm` is the number of total pages mapped. The `hiwater_rss` and `hiwater_vm` fields represent the high-watermark, that is the highest peak, of the resident set size (RSS) and virtual memory usage, respectively.
- `def_flags` and `flags` define the default access flags and actual access flags of the memory descriptor. The `{data,exec,stack}_vm` fields represent the access flags of the respective memory section.
- `{start,end}_{code,data,brk,stack}` and `{arg,env}_{start,end}` are the start and end of the respective memory section.

The usage of most of the described fields is intuitive. However, with `mm_users` and `mm_count` it is not clear why two different counters are used. The reason is simple though: having two counters enables the kernel to differentiate between the main usage counter (`mm_count`) and the number of processes using the address space (`mm_users`). Only when `mm_users` reaches zero is `mm_count` decremented. When `mm_count` finally reaches zero, there are no remaining references to this `mm_struct` and it is released from memory [45]: `linux/mm_types.h:518`.

Virtual Memory Area

The Linux kernel implements a data structure called *virtual memory area (VMA)* to distinguish between different logical memory areas (*e.g.*, code, stack, heap). Each VMA describes a piece of a process's address space; that piece is a (usually contiguous) series of pages with a uniform set of permissions. In Linux, every VMA is typically either a *file-backed mapping* that is backed by a device (*e.g.*, shared libraries), or an *anonymous mapping* (*e.g.*, stack, heap) [46]. Anonymous mappings are just in-memory, there is no block device backing up the contents of the page as a device backed mapping has. Below is an excerpt of the VMA data structure:

```
<linux/mm_types.h>
struct vm_area_struct {
        struct mm_struct            *vm_mm;
...
        unsigned long               vm_start;
        unsigned long               vm_end;
...
        struct vm_area_struct       *vm_next, *vm_prev;
        struct rb_node              vm_rb;
...
        pgprot_t                    vm_page_prot;
        unsigned long               vm_flags;
...
        struct file *               vm_file;
        unsigned long               vm_pgoff;
...
        struct vm_userfaultfd_ctx   vm_userfaultfd_ctx;
};
```

Once again we divide the data structure into sections hinted by the ellipsis:

- `vm_mm` points to the memory descriptor of the VMA.
- `vm_start` is the initial (lowest) address in the interval and `vm_end` is the first byte after the final (highest) address in the interval.
- `vm_next` and `vm_prev` are pointers to the next and previous element in a doubly linked list, while `vm_rb` points to the node in the red-black tree structure.
- `vm_page_prot` represents the access permissions for an individual page. `vm_flags` represent the flags set for the VMA. See Table 3 for details.
- `vm_file` is the mapped file of the VMA (can be a null pointer). `vm_pgoff` is the offset (within `vm_file`) in `PAGE_SIZE` units.
- `vm_userfaultfd_ctx` is the context used to enable userfaultfd capabilities.

Using the `vm_start` and `vm_end` field, the length of each VMA can be calculated by subtracting `vm_end` − `vm_start`, which exists over the interval [`vm_start`, `vm_end`[. VMAs are unique to their referenced memory descriptor. Consequently, even if two different processes map the same file into their respective address space, both have a unique VMA linked to their respective memory descriptor.

The behaviour of different VMAs can be specified by setting pre-defined bits in the field `vm_flags`. The following paragraph describes different flags, while also presenting a table with all (as of version 5.18) available values.

Virtual Memory Area Flags

The possible bit values for `vm_flags` are defined in `<linux/mm.h>`. These flags specify behaviour for which the kernel is responsible and apply for the whole VMA. Table 3 is a listing of the possible `vm_flags` values.

Table 3: Functional overview of virtual memory area flags.

Flag	Effect on VMA
VM_NONE	No flags are set.
VM_READ	Pages can be read from.
VM_WRITE	Pages can be written to.
VM_EXEC	Pages can be executed.
VM_SHARED	Pages are shared.
VM_MAYREAD	The VM_READ flag can be set.
VM_MAYWRITE	The VM_WRITE flag can be set.
VM_MAYEXEC	The VM_EXEC flag can be set.
VM_MAYSHARE	The VM_SHARE flag can be set.
VM_GROWSDOWN	The area can grow downward.
VM_UFFD_MISSING	Userfaultfd triggers on missing pages.
VM_PFNMAP	The area is purely managed by PFN.
VM_UFFD_WP	Userfaultfd triggers on write-protected pages.
VM_LOCKED	The pages in this area are locked.
VM_IO	The area maps a device's I/O space.
VM_SEQ_READ	The pages are accessed sequentially.
VM_RAND_READ	The pages are accessed randomly.
VM_DONTCOPY	The area must not be copied on fork().
VM_DONTEXPAND	This area cannot expand with mremap().
VM_LOCKONFAULT	The pages must be locked when they fault.
VM_ACCOUNT	This area is a VM accounted object.
VM_NORESERVE	This area suppresses VM accounting.
VM_HUGETLB	This area uses huge pages.
VM_SYNC	This area uses synchronous page faults.
VM_ARCH_1	This is an architecture-specific flag.
VM_WIPEONFORK	This area's contents are wiped in child on fork().
VM_DONTDUMP	This area is not included in core dump.
VM_SOFTDIRTY	This area is not soft dirty clean.
VM_MIXEDMAP	This area is managed by PTEs and PFNs.
VM_HUGEPAGE	This area is marked by MADV_HUGEPAGE.
VM_NOHUGEPAGE	This area is marked by MADV_NOHUGEPAGE.
VM_MERGEABLE	KSM may merge identical pages in this area.

Now let us look in-depth at the more important VMA flags listed above. The VM_READ, VM_WRITE and VM_EXEC flags specify the familiar read, write and execute permissions for the pages in this particular VMA. These flags can be combined to form the desired access permissions that a process must respect when accessing the VMA. For example, the code segment for a process may be mapped with VM_READ and VM_EXEC, but not with the VM_WRITE flag. Conversely, the data section may be mapped with VM_READ, VM_WRITE and even VM_SHARED—that is, if the mapping is to be shared between processes. If the VM_SHARED flag is set, the mapping is intuitively called a *shared mapping*. Vice versa, if the flag is not set, the mapping is called a *private mapping* [41].

The VM_UFFD_MISSING and VM_UFFD_WP flags are set by the *userfaultfd()* system call. The system call allows the user space to intervene in the handling of page faults [47]. Both flags specify different behaviour of the userfaultfd page fault routine:

- VM_UFFD_MISSING specifies that page faults get raised to user space whenever a page is missing, that is not present.
- VM_UFFD_WP specifies that page faults get raised to user space whenever a previously write-protected page is accessed.

Userfaultfd can be used, for example, to implement live snapshotting of running processes by marking memory regions write-protected, and consequently, generating page faults on write-access. Those faults can then be used to copy the modified pages (and only those) to the snapshot [48].

The VM_DONTCOPY, VM_WIPEONFORK, VM_HUGEPAGE, VM_NOHUGEPAGE and VM_MERGEABLE flags are set by calling the madvise() system call. The madvise() system call is used to give advice or directions to the kernel about a memory region, specified by the address and the length arguments. To set the flags, madvise() must be called with the following advice values, respectively:

- MADV_DONTFORK advises the kernel to not copy the memory region on a fork() call. This is useful to prevent COW semantics from changing the physical location of a page if the parent writes to it after a fork().
- MADV_WIPEONFORK advises the kernel to wipe the contents of the memory region (*i.e.*, filling the pages with zero) on a fork() call. This is useful in forking servers in order to ensure that sensitive per-process data (for example, pseudo-random number generator seeds, cryptographic secrets, and so on) is not handed to child processes.
- MADV_HUGEPAGE advises the kernel to replace private anonymous pages with huge pages. This is useful for applications that use large mappings of data and access large regions of that memory at a time (*e.g.*, virtualisation systems such as QEMU).

- ❑ `MADV_NOHUGEPAGE` advises the kernel to ensure that the memory region is not backed by a huge page.
- ❑ `MADV_MERGEABLE` advises the kernel to enable KSM for the memory region. This is useful for applications that generate many instances of the same data (*e.g.*, virtualisation systems such as the kernel-based virtual machine).

In most cases, the goal of such advice is to improve system or application performance [49].

Lists and Trees

As discussed earlier, VMAs are accessed via the `mmap` and `mm_rb` fields, which represent a doubly linked list and a red-black tree, respectively. These two data structures independently point to all VMAs associated to a memory descriptor. In fact, they point to the same VMAs, merely using different data structures.

The first field, `mmap`, points to the first element in the doubly linked list. Each VMA is linked into the list in ascending order of start address using both, the `vm_next` and `vm_prev`, fields. These fields point to the next element and the previous element (or `NULL` if there is none). The memory descriptor always has a reference (`mmap`) to the first element in this list.

The second field, `mm_rb`, points to the root of the red-black tree. Each VMA is linked into the tree via the `vm_rb` field.

Red-black trees are self-balancing binary search trees. Balanced tree structures have the advantage of guaranteeing a $\mathcal{O}(\log n)$ time for basic dynamic-set operations (*e.g.*, SEARCH, PREDECESSOR, SUCCESSOR, MINIMUM, MAXIMUM, INSERT and DELETE) [50].

The doubly linked list is used when every node needs to be traversed. The red-black tree is used when locating a specific VMA in the address space (*e.g.*, when adding a new region) the kernel first searches the red-black tree for the region immediately preceding the new region. Thus, the kernel uses the redundant data structures to provide optimal performance regardless of the operation performed on the VMAs. Figure 8 illustrates the correlation of a process, its respective memory descriptor and the VMAs linked to it.

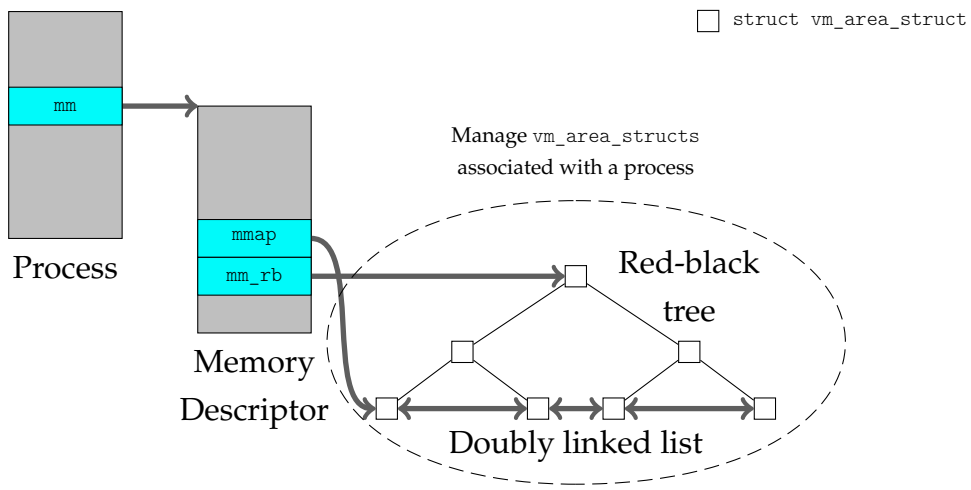


Figure 8: The VMAs of a process are organised two different data structures: a red-black tree and a doubly linked list. The entry pointers to each of these data structures is in the respective memory descriptor of the process [40].

The Process Memory Map

Linux discloses the VMAs of a process's address space via the `/proc/<pid>/maps` file. Consider the following simple user space program:

```
int main (int argc, char *argv[])
{
        return 0;
}
```

The annotated output from `/proc/<pid>/maps` lists all VMAs in this process's address space:

```
$ cat /proc/38297/maps
start            end              perm offset   device inode    file
564f32db9000-564f32dba000 r--p 00000000 103:08 1855001  /home/mario/src/main
564f32dba000-564f32dbb000 r-xp 00001000 103:08 1855001  /home/mario/src/main
564f32dbb000-564f32dbc000 r--p 00002000 103:08 1855001  /home/mario/src/main
564f32dbc000-564f32dbd000 r--p 00002000 103:08 1855001  /home/mario/src/main
564f32dbd000-564f32dbe000 rw-p 00003000 103:08 1855001  /home/mario/src/main
7f484ea00000-7f484ea28000 r--p 00000000 103:08 3411360  /usr/lib/libc.so.6
7f484ea28000-7f484eba0000 r-xp 00028000 103:08 3411360  /usr/lib/libc.so.6
7f484eba0000-7f484ebf8000 r--p 001a0000 103:08 3411360  /usr/lib/libc.so.6
7f484ebf8000-7f484ebf9000 ---p 001f8000 103:08 3411360  /usr/lib/libc.so.6
7f484ebf9000-7f484ebfd000 r--p 001f8000 103:08 3411360  /usr/lib/libc.so.6
7f484ebfd000-7f484ebff000 rw-p 001fc000 103:08 3411360  /usr/lib/libc.so.6
7f484ebff000-7f484ec0c000 rw-p 00000000 00:00  0
7f484ecb4000-7f484ecb8000 rw-p 00000000 00:00  0
7f484ecd6000-7f484ecd8000 r--p 00000000 103:08 3411330  /usr/lib/ld-linux-x86-64.so.2
7f484ecd8000-7f484ecff000 r-xp 00002000 103:08 3411330  /usr/lib/ld-linux-x86-64.so.2
7f484ecff000-7f484ed0a000 r--p 00029000 103:08 3411330  /usr/lib/ld-linux-x86-64.so.2
7f484ed0b000-7f484ed0d000 r--p 00034000 103:08 3411330  /usr/lib/ld-linux-x86-64.so.2
7f484ed0d000-7f484ed0f000 rw-p 00036000 103:08 3411330  /usr/lib/ld-linux-x86-64.so.2
7ffd49307000-7ffd49328000 rw-p 00000000 00:00  0        [stack]
7ffd493ad000-7ffd493b1000 r--p 00000000 00:00  0        [vvar]
7ffd493b1000-7ffd493b3000 r-xp 00000000 00:00  0        [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0 [vsyscall]
```

We already know that there must be a code, a data and a stack segment. Each VMA can be correlated to a segment, based on the set permissions. The first VMA is a read-only, private mapping for the `main` file. This is a private data segment, which most probably contains global variables or constants and is therefore read-only. The second VMA is a readable and executable, private mapping, again for the `main` file. Given that it is executable, this VMA is the code segment. The next differing VMA is the one that is a readable and writable, private mapping, once again for the `main` file. This VMA represents a data segment in which the process can store and update variables. Next, there is a mixture of readable, writable and executable VMAs that are related to libc, a shared library. There are also additional helpful pseudo-paths, for example, `heap`, `stack`, `vdso` or `vsyscalls` [51]. The pseudo-path `vdso` stands for virtual dynamic shared object. It is used by system calls to switch to kernel mode [52]. The pseudo-path `vsyscalls` is also used to accelerate certain system calls in Linux [53].

Note the memory areas without a mapped file that are on device 00:00 and inode zero. This is the zero page. The zero page is a mapping that consists of all zeros. By mapping the zero page over a writable memory area, the area is in effect initialised to all zeros. Because the mapping is not shared, as soon as the process writes to this data a copy is made (just like with COW) and the value is updated from zero.

There is also the `pmap` utility [54] that displays a formatted listing of a process's VMAs. It is a bit more readable than the `/proc/<pid>/maps` output, but it is the same information:

```
$ pmap -x 38297
38297:   ./main
Address           Kbytes     RSS   Dirty Mode  Mapping
0000564f32db9000       4       4       0 r---- main
0000564f32dba000       4       4       0 r-x-- main
0000564f32dbb000       4       0       0 r---- main
0000564f32dbc000       4       4       4 r---- main
0000564f32dbd000       4       4       4 rw--- main
00007f484ea00000     160     160       0 r---- libc.so.6
00007f484ea28000    1504     728       0 r-x-- libc.so.6
00007f484eba0000     352       0       0 r---- libc.so.6
00007f484ebf8000       4       0       0 ----- libc.so.6
00007f484ebf9000      16      16      16 r---- libc.so.6
00007f484ebfd000       8       8       8 rw--- libc.so.6
00007f484ebff000      52      16      16 rw--- [ anon ]
00007f484ecb4000      16      12      12 rw--- [ anon ]
00007f484ecd6000       8       8       0 r---- ld-linux-x86-64.so.2
00007f484ecd8000     156     156       0 r-x-- ld-linux-x86-64.so.2
00007f484ecff000      44      44       0 r---- ld-linux-x86-64.so.2
00007f484ed0b000       8       8       8 r---- ld-linux-x86-64.so.2
00007f484ed0d000       8       8       8 rw--- ld-linux-x86-64.so.2
00007ffd49307000     132      12      12 rw--- [ stack ]
00007ffd493ad000      16       0       0 r---- [ anon ]
00007ffd493b1000       8       4       0 r-x-- [ anon ]
ffffffffff600000       4       0       0 --x-- [ anon ]
---------------- ------- ------- -------
total kB            2516    1196      88
```

Again the output can be analysed line by line and the given memory regions can be identified based on the set permissions (annotated as *Mode* when using `pmap`). It goes without

saying that both outputs represent the same VMAs, as both outputs are based on the same process (which is in the same state). Therefore, every mapping that can be found in the `/proc/<pid>/maps` output must also be contained in the `pmap` output. These utilities can be of great use to get a detailed overview of the allocated VMAs.

3.2.2 System Calls

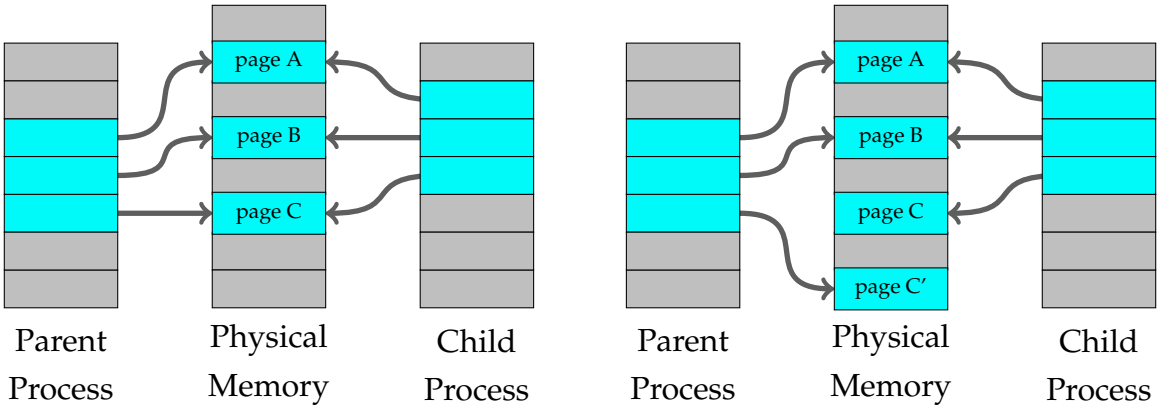
The Linux kernel uses *system calls*, often abbreviated as *syscalls*, as the fundamental interface between user space and kernel space. There are hundreds of system calls for more than 40 architectures available in the Linux operating system [55]. As covering all system calls clearly is out of scope for this work, we will only have a detailed look into a few relevant ones. We already briefly covered two of them in Chapter 3.2.1: `userfaultfd()` and `madvise()`. In the following paragraphs four more will be discussed: `fork()`, `mmap()`, `munmap()` and `mremap()`

Fork

```
pid_t fork(void);
```

The `fork()` system call is used to create a new process by duplicating the calling process. The new process is referred to as the *child* process. The calling process is referred to as the *parent* process.

The child process and the parent process run in separate memory spaces, that is they use separate memory descriptors. At the time of `fork()` both memory spaces have the same content. Under Linux, `fork()` is implemented using COW pages, so the only penalty that it incurs is the time and memory required to copy the parent's page tables, and to create a unique task structure for the child [56]. This means that pages are simply shared until one of the processes modifies a page. Whenever a page is modified, the page gets duplicated as depicted in Figure 9.



(a) Before parent process modifies page C. (b) After parent process modifies page C.

Figure 9: Parent and child process sharing data using COW. When one process modifies the data the respective page is duplicated.

Typically, the vast majority of `fork()` calls are followed nearly immediately by an exec system call [57]—that is only a few pages actually get modified in the child before its entire memory space ceases to exist and gets replaced by a new program. COW ensures that `fork()` does not have to copy vast amounts of data, only to have them destroyed moments later. The `fork()` system call is often exploited for its COW semantics [8, 16].

Mmap

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

The `mmap()` system call is used to create a new mapping in the virtual address space of the calling process. The system call either creates a new VMA or expands an existing one, if the created address is adjacent to an existing address and shares the same permissions [58]. Memory mapped by `mmap()` is preserved across `fork()`, with the same attributes. These attributes include the memory protection of the mapping (available values shown in Table 4) and the type of mapping (*e.g.*, shared or private). The protection bits are defined in `<asm/mman.h>`.

Table 4: Mmap page access protection bits.

Bit	Effect on the pages of the VMA
`PROT_READ`	Data can be read.
`PROT_WRITE`	Data can be written.
`PROT_EXEC`	Data can be executed.
`PROT_NONE`	Data can not be accessed.

As discussed earlier, VMAs are typically either file-backed mappings or anonymous mappings. The `mmap()` system call can create both mapping types, depending on whether a file descriptor greater than one or the flag `MAP_ANONYMOUS` is passed. The flags passed to `mmap` are highly architecture dependent, the most common ones are shown in Table 5 and are defined in `<asm-generic/mman-common.h>`.

Table 5: Mmap VMA handling flags.

Flag	Effect on the VMA
`MAP_SHARED`	The mapping can be shared.
`MAP_PRIVATE`	The mapping can not be shared.
`MAP_FIXED`	The new mapping *must* start at the given address.
`MAP_ANONYMOUS`	The mapping is not file-backed, but is anonymous.
`MAP_POPULATE`	Populate (pre-fault) page tables.
`MAP_HUGETLB`	The mapping uses huge pages.

Another limitation of mmap is that mappings are page-granular. The kernel can manage virtual addresses only at the level of page tables; therefore, the mapped area must be a multiple of `PAGE_SIZE` and must live in physical memory starting at an address that is a multiple of `PAGE_SIZE`. The kernel forces size granularity by making a region slightly bigger if its size is not a multiple of the page size. For example, for a file that is not a multiple of the page size, the remaining bytes in the partial page at the end of the mapping are zeroed when mapped, and modifications to that region are not written out to the file [59].

Munmap

```
int munmap(void *addr, size_t length);
```

The `munmap()` system call is used to delete an existing mapping in the virtual address space of the calling process. This causes further references to addresses within the range to generate invalid memory references. The region is also automatically unmapped when the process is terminated.

Mremap

```
void *mremap(void *old_addr, size_t old_size, size_t new_size,
             int flags, ... /* void *new_addr */);
```

The `mremap()` system call is used to expand or shrink an existing memory mapping, potentially moving it at the same time (controlled by the `flags` argument and the available virtual address space). The `flags` bit-mask argument is used to control the behaviour of the system call, as the functionality of the system call is quite dynamic. The effects of these flags are shown in Table 6.

Table 6: Mremap behaviour modification flags.

Flag	Effect on the system call
`MREMAP_MAYMOVE`	If necessary, the kernel may relocate the mapping to a new virtual address.
`MREMAP_FIXED`†	If supplied, the fifth argument `*new_addr` is accepted and specifies a page-aligned address to which the mapping must be moved. Any previous mapping at the new address range is unmapped.
`MREMAP_DONTUNMAP`†	Remaps a mapping to a new address but does not unmap the mapping at `*old_addr`. Any access to the old range of memory will result in a page fault, which can be handled by a userfaultfd. Otherwise, the kernel allocates a zero-filled page to handle the fault.

†Must be used in conjunction with `MREMAP_MAYMOVE`.

The `mremap()` system call can, for example, be used to resize an existing VMA, to move an existing VMA to another virtual address, or to move an existing VMA to another virtual address, while keeping the old VMA mapped as write-protected, so that any access to it will result in a page fault. This page fault can be handled by a userfaultfd. Otherwise, the kernel allocates a zero-filled page to handle the fault.

To move an existing VMA to another virtual address, the kernel first copies the VMA to a new virtual address. Then it moves the page tables to the new VMA by creating new PTEs and dropping the old ones. If there is no error, the old VMA is dropped and the new VMA is returned.

As part of this thesis, a new flag called `MREMAP_COW` is proposed, which also requires `MREMAP_DONTUNMAP` to be specified. However, instead of simply write-protecting the old VMA, with `MREMAP_COW` the old VMA is COW-protected. This means that any read-access to it will resolve to the same data as accessing the new VMA would have. When either the old VMA or the new VMA is modified, the COW gets triggered and thus copies for the respective pages are created for the unmodified VMA. More details on the implementation of `MREMAP_COW` can be found in Chapter 4.1.2.

3.2.3 Four-Level Page Tables

As discussed earlier, operating systems can use the concept of multi-level page tables to handle large virtual address spaces by avoiding to keep all page tables in memory. The Linux kernel maintains the concept of a four-level page table in its architecture independent code. These four levels consist of the *page global directory (PGD)*, the *page upper directory (PUD)*, the *page middle directory (PMD)* and finally the already known page table entry (PTE). Each of the directories has a maximum size of 512 entries, which is defined in `<asm/pgtable_64_types.h>`. Architectures that manage their MMU differently are expected to emulate the four-level page table. For example, a 32 bit architecture that only uses two page table levels defines the PUD and PMD to be of size 1 and thus, they will be optimised out at compile time [21].

Unlike the 32 bit case, the 64 bit memory map is a direct reflection of hardware constraints. Linux's four-level page table concept can provide access to a 256 TiB subset at any given time. This is not a limiting factor though, as current x86_64 processors support a physical address space of up to 2^{48} bytes of RAM, or 256 TiB [23].

However, Intel has already implemented a scheme with a five-level page table, which allows Intel 64 processors to support a 57 bit virtual address space [60]. But Linux is already prepared, as the code for five-level page tables is already available.

The x86_64 Virtual Memory Map

Currently, the Linux x86_64 virtual memory map implementation splits the address space into two: the lower region (with the top bits set to 0) is user space, the upper region (with the top bits set to 1) is kernel space. Note that x86_64 defines canonical lower half and higher half addresses as an address where the address bits from the most-significant implemented bit up to bit 63 are all ones or all zeros [23]. This effectively limits the number of implemented bits to 48 or 57, as each level typically is of size 512.

The Linux kernel defines all page directory types (`pte_t`, `pmd_t`, `pud_t` and `pgd_t`) in `<asm/pgtable_64_types.h>` as `unsigned long` [45]. There are multiple macros and functions which simplify the process of walking the page table. They all follow the same scheme and are defined as follows (using the PGD as example):

```
<linux/pgtable.h>
#define PGDIR_SHIFT     39
#define PTRS_PER_PGD    512

#define pgd_index(a)    (((a) >> PGDIR_SHIFT) & (PTRS_PER_PGD - 1))

static inline pgd_t *pgd_offset(pgd_t *pgd, unsigned long address)
{
        return (pgd + pgd_index(address));
};
```

The function `pgd_offset()` basically calculates the pointer to the correct PGD entry. Another frequently used macro is the `pgd_addr_end()`.

```
<linux/pgtable.h>
#define PGDIR_SHIFT     39
#define PGDIR_SIZE      (_AC(1, UL) << PGDIR_SHIFT)
#define PGDIR_MASK      (~(PGDIR_SIZE - 1))

#define pgd_addr_end(addr, end)                                         \
({      unsigned long __boundary = ((addr) + PGDIR_SIZE) & PGDIR_MASK;  \
        (__boundary - 1 < (end) - 1) ? __boundary : (end);              \
})
```

This macro basically calculates the address of the next boundary of the PGD. As stated earlier, these macros and functions are available for each page directory, that is the PGD, the PUD, the PMD and the PTE.

When presented a virtual address, the Linux kernel steps through the page tables as depicted in Figure 10.

1. The `mm_struct` of a process has a pointer to the first entry of the PGD.
2. The kernel combines the `pgd_t` pointer to the first entry and the index using the `pgd_offset()` function to find the correct PGD entry.
3. The kernel combines the `pud_t` pointer to the first entry and the index using the `pud_offset()` function to find the correct PUD entry.
4. The kernel combines the `pmd_t` pointer to the first entry and the index using the `pmd_offset()` function to find the correct PMD entry.
5. The kernel combines the `pte_t` pointer to the first entry and the index using the `pte_offset()` function to find the correct PTE entry.
6. The calculated `pte_t` entry contains the PFN.
7. The PFN is finally indexed by the 12 least significant bits of the address.

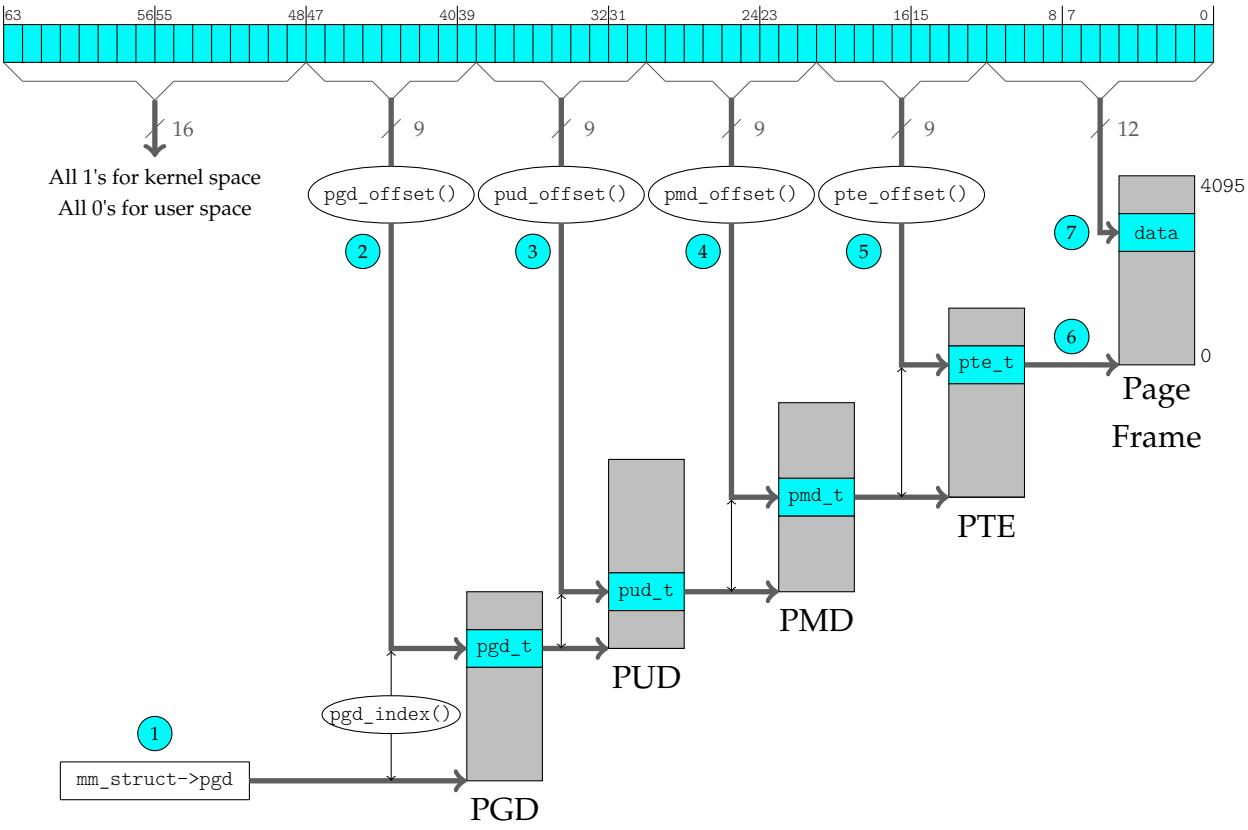


Figure 10: Linux resolving a virtual address using a four-level page table. A 64 bit address is resolved by calculating the respective page directory index and indexing the consecutive page directory. The last 12 bit are then used to index the physical PFN.

Now that we know how Linux translates a virtual address into a physical address, we can conclude the virtual memory chapter. Virtual memory is indispensable in today's computer systems. It allows to abstract the difficult problems of physical memory management in a way that the programmer does not have to worry about it. To ensure that this abstraction does not compromise performance, many different optimisations are applied. Among them are the TLB, multi-level page tables, different page replacement algorithms and others.

The Linux kernel uses different data structures to implement its virtual memory management, such as the `mm_struct` and the `vm_area_struct`. It also leverages different page sizes, offers interfaces and tools to represent the virtual address space of processes, and offers system calls to interact with its virtual memory. By implementing a four-level page table and using the TLB, if available, translations from virtual addresses to physical addresses are guaranteed to be fast.

These are not the only benefits of virtual memory. Mechanisms such as KSM and COW are only possible by applying a virtual memory concept and using paging. In the next chapter we will investigate different COW mechanisms, in both, the user space and the kernel space.

4 Copy-on-Write Mechanisms

To develop a fast and efficient dedicated mechanism to create a COW based snapshot of a VMA, multiple mechanisms were implemented and investigated. For future reference, we denote *origin* as the original VMA and *snapshot* as the VMA that is a copy of the original VMA. The requirements of such a mechanism are straightforward: it has to create a snapshot from an origin, where any modification in the origin triggers the duplication. The triggered duplication maps the modified data to the origin, while the data in the snapshot remains unmodified. This duplication must be handled by the Linux kernel. This is typically done, as stated earlier, by having read- and write-access to a VMA, while the respective PTEs indicate a read-only page. Additionally, the mechanism shall be as performant as possible. However, even though the requirements are straightforward, the implementations get quite sophisticated as we will see with the kernel space mechanisms.

This chapter will introduce the implemented and investigated user space and kernel space COW mechanisms, and provide an in-depth explanation of the respective concepts. In Chapter 5 we will measure and compare each mechanism, and also discuss advantages and disadvantages. Let us now start with the kernel space mechanisms.

4.1 Kernel Space

When implementing a dedicated COW mechanism, the kernel space offers several advantages. First off, system calls are usually atomic in the sense that they either succeed or fail. If they fail, they do a rollback and have no effect other than returning an error to the caller. They are also atomic in the sense that they try hard not to expose any intermediate state between the initial state and the final state to other threads or processes running on the system. For example, a file is either created or it does not exist.

But not only that, since the creation of a new VMA always demands a context switch to kernel mode, time can also be saved by completing the required operations while still in kernel mode. So let us investigate the first kernel space mechanism called mmapcopy.

4.1.1 Mmapcopy

```
void* mmapcopy(void *dup, void *orig,
               size_t len, int flags);
```

The system call `mmapcopy()` is a kernel extension by Korb *et al.* [15]. This system call uses the existing COW functionality of the Linux kernel to avoid duplicating memory when data is copied. In their work they describe how they use `mmapcopy()` in the R language to gain significant improvements in CPU time compared to KSM without compromising the amount of memory saved. The R language [61] is commonly used for processing large vector data structures [62]. It uses pass-by-value semantics, while also implementing garbage collection, which further increases memory pressure.

Implementation

Korb *et al.* provided their source code upon request based on a Linux kernel version 4.4. However, since Linux kernel version 5.18 is used throughout this thesis, the sources were updated to that version.

To create a new COW mapping, `mmapcopy()` induces the following steps: First, it searches for an unmapped and correct sized VMA, using the `dup` as a hint if not `NULL`. Next, it copies the structure of the origin to the found VMA in the same memory descriptor. Then, it copies the page tables from the origin to the snapshot using the virtual addresses of the new VMA for the copied PTEs. Finally, all PTEs are write-protected, so that the kernel can detect the COW pages.

The `mmapcopy()` system call fulfils the requirements we have set. However, in its current implementation it does not support huge pages. This is not a direct limitation but the COW mechanism shall be as generic as possible. Therefore, supporting huge pages is a reasonable feature. Additionally, the snapshot creation time of conducting a `mmapcopy()` is actually significantly higher than a `fork()`, as we will see in Chapter 5.

4.1.2 Mremap

```
void *mremap(void *old_addr, size_t old_size, size_t new_size,
             int flags, ... /* void *new_addr */);
```

The main contribution of this thesis is the proposal of adding the `MREMAP_COW` flag to the Linux kernel. Supplying `mremap()` with this flag copies an already existing VMA from `old_addr` to `new_addr`, while ensuring that both copies support COW. This means that after the `mremap()` call, both VMAs share the same data. However, any modification to any page—whether in the origin or the snapshot—triggers the duplication of that respective page.

The kernel modifications needed for extending Linux with `MREMAP_COW` are minimal. Appendix A is the required patch file for the kernel extension. It includes 93 insertions and 71 deletions.

Implementation

To increase the chances of a upstream merge, it is important to reuse available functions in the Linux kernel. Since `fork()` shares some semantics with our requirements, it must use a function which at least copies PTEs to a new destination, because `fork()` copies all existing VMAs to its child process. And indeed, there is a function called `copy_page_range()`.

copy_page_range

```
int copy_page_range(struct vm_area_struct *dst_vma,
                    struct vm_area_struct *src_vma);
```

This function is defined in <mm/memory.c> and takes two VMAs as arguments. According to the set requirements for a COW mechanism, the src_vma already exists. Thus, a dst_vma has to be created that is linked to the same mm_struct as the src_vma. This is done by calling mremap() and specifying the MREMAP_DONTUNMAP flag. When supplied with this flag, the mremap() system call internally creates a new VMA through copy_vma() starting at new_addr that is linked to its respective mm_struct. The code path of mremap() then calls move_page_tables() to move the PTEs accordingly. However, when calling mremap() with the new proposed flag MREMAP_COW, the kernel now calls copy_page_range() instead of move_page_tables().

Nevertheless, this still does not create the requested COW-mapping since the function copy_page_range() is traditionally only called by fork(). Because fork() creates a new process from an existing one, all VMAs of the existing process are copied to the new child process. Since every process has its own virtual address space, the copied VMAs are represented in each process's respective virtual address space. Therefore, the VMAs span the same virtual addresses, in both, the child process and the parent process. That is why copy_page_range() does not respect the dst_vma virtual addresses, since they are expected to be the same as the src_vma virtual addresses.

However, this problem is also addressed by the kernel modification required for the MREMAP_COW flag. To understand the modifications proposed to copy_page_range(), let us now have a look at how the function traditionally operates. Algorithm 1 is a pseudo-code representation of the unmodified copy_page_range() function. The function basically iterates through the four-level page table and allocates a new respective entry for the dst_vma. It then establishes the COW-protection by setting both, the src_pte and the dst_pte, to write-protected. This effectively tells the kernel that these pages are COW pages, as discussed in Chapter 3.1.2.

Algorithm 1 Copy page tables without addressing

```
 1: function COPY_PAGE_RANGE(dst_vma, src_vma)
 2:     addr ← start of src_vma
 3:     end ← end of src_vma
 4:     src_mm ← memory descriptor of src_vma
 5:     dst_mm ← memory descriptor of dst_vma
 6:     while addr ≠ end do                                   ▷ iterate PGDs
 7:         dst_pgd ← PGD_ALLOC(dst_mm, addr)
 8:         src_pgd ← PGD_OFFSET( (src_mm, addr)
 9:         while addr ≠ end do                               ▷ iterate PUDs
10:             dst_pud ← PUD_ALLOC(dst_mm, dst_pgd, addr)
11:             src_pud ← PUD_OFFSET(src_pgd, addr)
12:             while addr ≠ end do                           ▷ iterate PMDs
13:                 dst_pmd ← PMD_ALLOC(dst_mm, dst_pud, addr)
14:                 src_pmd ← PMD_OFFSET(src_pud, addr)
15:                 while addr ≠ end do                       ▷ iterate PTEs
16:                     dst_pte ← PTE_ALLOC(dst_mm, dst_pmd, addr)
17:                     src_pte ← PTE_OFFSET(src_pmd, addr)
18:                     WRITE_PROTECT(src_pte)
19:                     WRITE_PROTECT(dst_pte)                ▷ establish COW
20:                     SET_PTE_AT(dst_mm, addr, dst_pte)
21:                     addr ← addr + PAGE_SIZE
22:                 end while
23:                 addr ← PMD_ADDR_END(addr, end)
24:             end while
25:             addr ← PUD_ADDR_END(addr, end)
26:         end while
27:         addr ← PGD_ADDR_END(addr, end)
28:     end while
29: end function
```

The proposed changes to `copy_page_range()` are intelligible, yet effective: instead of solely relying on the `addr` of the `src_vma`, the code also involves using the `addr` of the `dst_vma`. Algorithm 2 shows the modified pseudo-code. Basically, `copy_page_range()` now respects the addressing of the `dst_vma`, which was unnecessary before, as in the case of `fork()` both VMAs represent the same virtual addresses.

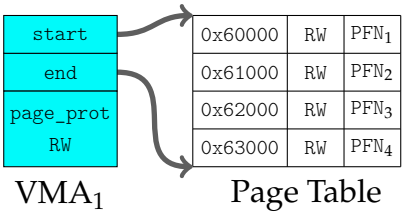
Algorithm 2 Copy page tables with addressing

```
 1: function COPY_PAGE_RANGE(dst_vma, src_vma)
 2:     src_addr ← start of src_vma
 3:     src_end ← end of src_vma
 4:     dst_addr ← start of dst_vma
 5:     dst_end ← end of dst_vma
 6:     src_mm ← memory descriptor of src_vma
 7:     dst_mm ← memory descriptor of dst_vma
 8:     while src_addr ≠ src_end || dst_addr ≠ dst_end do                ▷ iterate PGDs
 9:         dst_pgd ← PGD_ALLOC(dst_mm, dst_addr)
10:         src_pgd ← PGD_OFFSET( (src_mm, src_addr)
11:         while src_addr ≠ src_end || dst_addr ≠ dst_end do            ▷ iterate PUDs
12:             dst_pud ← PUD_ALLOC(dst_mm, dst_pgd, dst_addr)
13:             src_pud ← PUD_OFFSET(src_pgd, src_addr)
14:             while src_addr ≠ src_end || dst_addr ≠ dst_end do        ▷ iterate PMDs
15:                 dst_pmd ← PMD_ALLOC(dst_mm, dst_pud, dst_addr)
16:                 src_pmd ← PMD_OFFSET(src_pud, src_addr)
17:                 while src_addr ≠ src_end || dst_addr ≠ dst_end do    ▷ iterate PTEs
18:                     dst_pte ← PTE_ALLOC(dst_mm, dst_pmd, dst_addr)
19:                     src_pte ← PTE_OFFSET(src_pmd, src_addr)
20:                     WRITE_PROTECT(src_pte)
21:                     WRITE_PROTECT(dst_pte)                           ▷ establish COW
22:                     SET_PTE_AT(dst_mm, dst_addr, dst_pte)
23:                     src_addr ← src_addr + PAGE_SIZE
24:                     dst_addr ← dst_addr + PAGE_SIZE
25:                 end while
26:                 src_addr ← PMD_ADDR_END(src_addr, end)
27:                 dst_addr ← PMD_ADDR_END(dst_addr, end)
28:             end while
29:             src_addr ← PUD_ADDR_END(src_addr, end)
30:             dst_addr ← PUD_ADDR_END(dst_addr, end)
31:         end while
32:         src_addr ← PGD_ADDR_END(src_addr, end)
33:         dst_addr ← PGD_ADDR_END(dst_addr, end)
34:     end while
35: end function
```

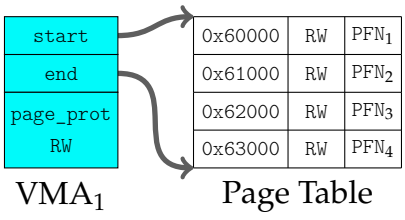
The modification has no effect on a `fork()` using `copy_page_range()`, since `src_addr` and `dst_addr` are the same. However, the modified version can now also handle deviating virtual addresses in the `src_vma` and the `dst_vma`, since it implements the addressing of the `dst_vma`.

Figure 11 depicts how a `mremap()` call using `MREMAP_COW` modifies the page tables to create a COW mapping. The page table is illustrated in simplified form as a single table. Figure 11a shows the state of the page table before calling `mremap()`. The page table is populated with VMA_1 that ranges from $[0x60000, 0x64000[$. Figure 11b shows the state of the page table after `mremap()` internally calls `copy_vma()`. The VMA_2 belongs to the same memory

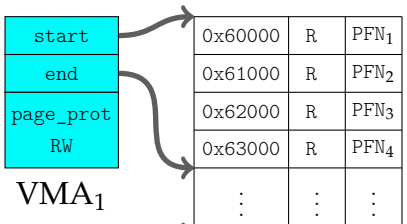
descriptor as VMA_1, and also has the same `page_prot` set. However, there are no entries for VMA_2 in the page table yet. Figure 11c shows the state of the page table after `mremap()` internally calls the modified `copy_page_range()`. The page table is now populated with the entries belonging to VMA_2. Note that, the protection bits of the individual pages changed to read-only.



(a) Page table before calling `mremap()` with `MREMAP_COW`.



(b) Page table while calling `mremap()` with `MREMAP_COW`; after `copy_vma()`.



(c) Page table after successfully calling `mremap()` with `MREMAP_COW`.

Figure 11: Series of page table states when calling `mremap()` with `MREMAP_COW`.

There is another subtle detail that can be missed easily. Whenever any address space is modified, the TLB has to be flushed, because entries in the x86 TLB are not associated with any particular address space; they implicitly refer to the current address space. The system call `mremap()` flushes the TLB whenever calling `move_page_tables()`. However, with `MREMAP_COW` the function `copy_page_range()` is called instead, which does not flush the TLB. Therefore, an explicit flush of the TLB is required after calling `copy_page_range()`.

Huge page support

The kernel extension of `MREMAP_COW` also supports huge pages through Appendix B. Once again, the functions implemented in the `fork()` system call shall be used. When `fork()` is

confronted with huge pages, it calls `copy_hugetlb_page_range()`. This function shares the same semantics as the previously used function `copy_page_range()` but uses huge pages (2 MiB or 1 GiB) instead of default pages (4 KiB). Since huge pages require a lot less PTEs to map the same amount of memory, the Linux kernel only uses the PUD and the PMD to manage huge pages. However, the pseudo-code for the `copy_hugetlb_page_range()` still is approximately the same.

The changes proposed to `copy_hugetlb_page_range()` are analogous to the proposed changes to `copy_page_range()`. The elementary modifications made are, once again, using the `addr` of the `dst_vma` instead of solely relying on the `addr` of the `src_vma`.

The extension of the `mremap()` system call with `MREMAP_COW` reuses already existing kernel code. The proposed changes do not interfere with any existing mechanisms. The COW protection applies to both, the origin and the snapshot. Therefore, the kernel extension meets the set requirements. In addition, it also supports huge pages. The snapshot creation time of calling `mremap()` with `MREMAP_COW` is mainly bound by creating the copies of the page tables through the use of `copy_page_range()`, as we will see in Chapter 5. This makes `MREMAP_COW` a serious competitor in the field of COW mechanisms.

4.1.3 AnKer

`void *vmcopy(void *src_addr, size_t len);`

The `vmcopy()` system call is part of a kernel modification of the AnKer [63] processing concept. It is used to separate Online Transaction Processing (OLTP) from Online Analytical Processing (OLAP) by using virtual snapshotting [10]. Once again, it uses the internal COW functionality of the Linux kernel by write-protecting the respective PTEs.

Implementation

The original patch [63] is based on a Linux kernel 4.16. Therefore, the patch was updated to Linux kernel 5.18, as the other mechanisms are based on this version.

To accomplish the COW-protection, the mechanism first searches for an unmapped VMA. However, the location of the unmapped VMA is determined by the kernel as there is no extra argument available to provide a hint to the kernel. When an unmapped VMA is found, AnKer copies the protection bits and links the VMA into the memory descriptor and also sets up the respective management structures. After that, AnKer also iterates through the page tables and copies each PTE as well as write-protecting them. Finally, the TLB is flushed, as the page tables are modified.

AnKer also supports the use of huge pages. In their repository they even require them via the kernel configuration setting `CONFIG_TRANSPARENT_HUGEPAGE=y`. This kernel setting is an alternative mean of using huge pages for the backing of virtual memory with huge pages that supports the automatic promotion and demotion of page sizes [64].

The AnKer kernel modification meets the set requirements of a COW mechanism. Both VMAs, the origin and the snapshot, support COW. However, the changes made to the kernel are substantial. This eliminates the possibilities of an upstream merge. Therefore, a custom kernel is required to make use of the `vmcopy()` system call. So let us now have a look at the possibilities to create a COW mapping implemented in the user space.

4.2 User Space

There are not many options when to implement a COW mechanism that operates on a VMA-granular level in user space. Granted, there is always the option to implement an application-specific COW memory management. However, this thesis aims to find a generic solution and solve the problem in the operating systems domain. So, let us start with our first user space mechanism.

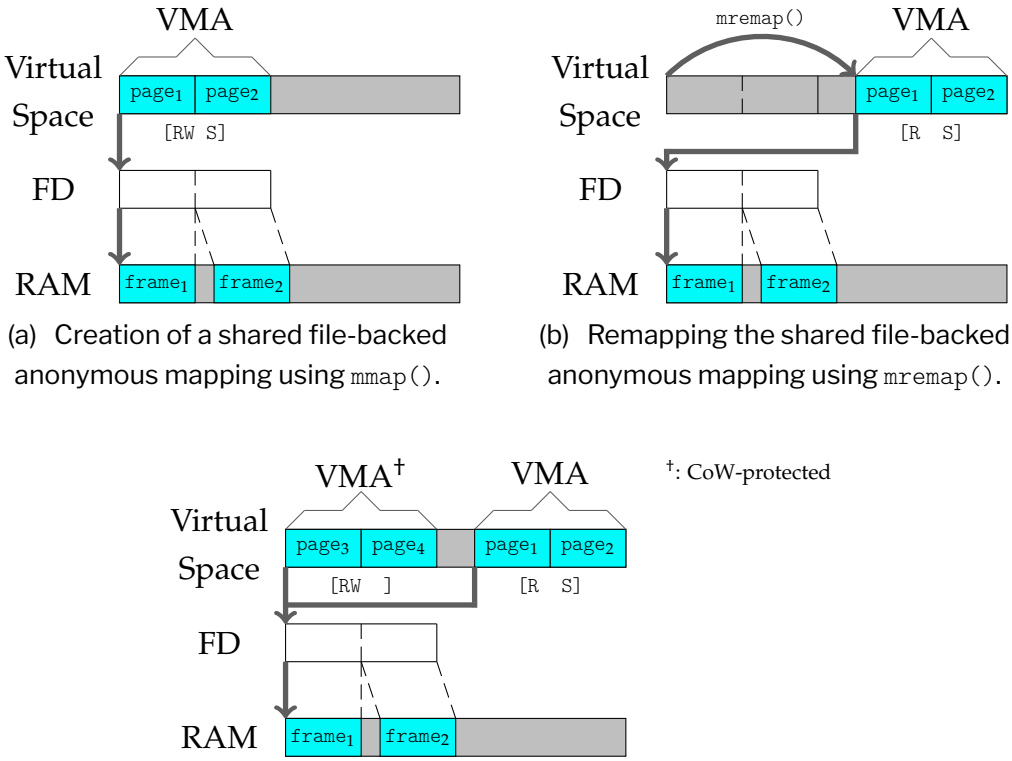
4.2.1 Scoot

The `scoot()` mechanism is a solution to provide a one-sided COW-protected VMA of an already existing VMA. The term *one-sided* means, that only one of the two VMAs is actually COW-protected. Thus, a write-access to one VMA triggers the duplication, while a write-access to the other VMA simply passes through the modification to both VMAs. This can be a limitation, but in many contexts—such as database persistence or simultaneous OLTP and OLAP—it is sufficient, as these operations only require read-access.

Implementation

To create such a mapping, the Linux kernel offers the possibility to create a private COW mapping using the `mmap()` system call. However, a prerequisite to this method is to have a shared file-backed anonymous mapping, which can be mapped by two VMAs. This is done by utilising the `memfd_create()` system call. The system call creates an anonymous file and returns a file descriptor that refers to it. The file behaves like a regular file, and so it can be modified, truncated, memory-mapped, and so on. However, unlike a regular file, it lives in RAM and has a volatile backing storage. Anonymous memory is used for all backing pages of the file [65].

With this file descriptor, `scoot()` uses a series of system calls to create a COW mapping as depicted in Figure 12. First, the origin is created through a shared file-backed anonymous mapping by calling `mmap()` with the file descriptor as outlined in Figure 12a. To create the snapshot mapping, a new private mapping has to be created by calling `mmap()` with the same file descriptor. However, only the new mapping is COW-protected. This does not meet the requirements as the origin requires the COW-protection, not vice versa. Therefore, before creating the snapshot, the origin is remapped by using `mremap()` to a different location in the virtual address space as depicted in Figure 12b. This VMA now serves as the snapshot. Consequently, the origin has to be recreated by calling `mmap()` and requesting the address of the former origin VMA. Figure 12c illustrates the state after the snapshot creation. Two different VMAs map to the same physical addresses, while the origin is COW-protected. Any modification to the origin triggers the duplication. The `scoot()` mechanism also fully supports the use of huge pages, since all system calls that are used in `scoot()` support huge pages (*i.e.*, `memfd_create()`, `mmap()` and `mremap()`).



(a) Creation of a shared file-backed anonymous mapping using `mmap()`.

(b) Remapping the shared file-backed anonymous mapping using `mremap()`.

(c) Creation of a COW-protected private file-backed anonymous mapping using `mmap()`.

Figure 12: Scoot using a series of system calls to create a COW mapping.

However, having only a one-sided COW-protection is not the only caveat of `scoot`. During the creation of the snapshot, the origin gets moved via `mremap()`. This means that the origin is not mapped until the final `mmap()` call. Therefore, accessing the origin in this interval causes a segmentation fault, as there is no VMA that maps the virtual addresses. This may not be a problem in a single-threaded application. However, any multi-threaded application may not be able to guarantee that there is no access to the origin memory while creating the snapshot. So, to avoid any memory corruption during the `scoot()` snapshot creation, the mechanism has to block other threads to achieve atomicity.

Another limitation of `scoot()` is by virtue of moving the origin via `mremap()`. As introduced in Chapter 3.2.2 `mremap()` moves the page tables whenever it has to move an existing VMA to a new location. This drops all existing PTEs of the old VMA. Therefore, the page table is not populated when moving the origin using `mremap()`. But not only that, the page table of the snapshot also is not populated, since Linux postpones the process of populating page tables until they are accessed. So any access of any page in either the origin or the snapshot results in a page fault, introducing additional overhead.

Consider an in-memory database application using the `scoot()` mechanism to snapshot its contents. It has two threads: the main thread handles database queries and the second thread writes the contents to a file on a disk. Whenever the database creates a snapshot, the main thread has to be blocked until the snapshot has been successfully created. But not only that, the main thread now has a reference to an origin with unpopulated page tables. Thus, the main thread gets slowed down as the operating system has to resolve the page fault for every accessed page.

Finally, the `scoot()` mechanism can only be used to create a single memory snapshot, because it can only create a COW mapping from an existing *shared* mapping. However, to create the COW mapping, `scoot()` remaps the origin as a *private* mapping. Therefore, only one COW-protected snapshot can exist at any time.

These limitations restrict the performance of an application severely. However, there is another possibility to implement COW in user space called userfaultfd.

4.2.2 Userfaultfd

The *userfaultfd (UFFD)* is a file descriptor in the Linux kernel. The file descriptor is created by the system call `userfaultfd()`. As stated in Chapter 3.2.1, the system call allows the user space to intervene in the handling of page faults, something otherwise only the kernel can do. Basically, a thread in a multi-threaded application can perform paging for other threads in the process.

The UFFD system call returns a file descriptor that refers to an UFFD object. Memory regions which should be supervised have to be registered to this object. The UFFD file descriptor can be polled by using `read()`, either blocking or non-blocking based on the flags set in the UFFD object. UFFD supports two operation modes: `MODE_MISSING` and `MODE_WP`. When registered with `MODE_MISSING`, a page fault notification is sent to the user space each time a missing page, that is there is no physical block backing the page, is accessed. When registered with `MODE_WP`, the notification is sent each time a write-protected page is written to. Write-protected in this context means, that the pages are marked through the use of an `uffdio_writeprotect` object. In both cases the faulted thread stops until the user space either resolves the page fault by filling the page or unprotects the write-protected page, respectively [47].

This mechanism can be used to implement COW in user space by registering the `MODE_WP` mode and protecting the desired regions in the UFFD object. Whenever one of the regions is written to, the handling thread needs to duplicate the written pages and then unprotect these pages to resolve the fault. However this introduces either a context switch as memory needs to be allocated and filled or it requires the memory to be allocated beforehand which would basically renders the COW useless.

Since the mechanism operates on page-granularity, it is more appropriate to use it in an application-specific context. The application itself can decide how to resolve the page faults, for example, it may only duplicate some of the pages, not all of them.

The possibilities of implementing a dedicated COW snapshot mechanism are limited in user space. Not only that, they also coincide with a multitude of constraints. On the one hand, UFFD offers the possibility to handle page faults in user space, but also induces the problem of how to properly duplicate the pages. On the other hand, `scoot()` provides a more generic approach, but has problems with unpopulated page tables and missing atomicity. The kernel already deals with these problems in other contexts. For example, when `fork()` creates a new process, it also has to atomically copy the page tables of one process to another.

In conclusion, there are many different ways to create a COW-protected VMA within the Linux operating system. Whether in user space (`scoot()` or UFFD) or in kernel space (`mmapcopy()`, `mremap()` or `vmcopy()`), every mechanism also comes with different caveats. In the next chapter, we will measure different scenarios and also discuss advantages and disadvantages of each mechanism.

5 Evaluation

In this chapter, we will discuss all presented COW mechanisms of Chapter 4, except UFFD, since it requires application-specific knowledge to implement a reasonable page fault handling routine. Every other mechanism is tested and measured with specific tools developed throughout this thesis. The functionality of each tool is described in the respective chapters. Finally, every tested mechanism is compared and evaluated using different charts.

5.1 Unit Test

To ensure that each mechanism works as expected, a unit test is run for every implementation. Algorithm 3 shows a pseudo-code of the steps executed by this unit test. Basically, the algorithm first allocates a fixed number of pages in a single VMA called `origin`. It then initialises every page of `origin` to `INIT_VAL`. Then, the `origin` gets duplicated by the respective snapshot mechanism (*e.g.*, `scoot()`, `mremap()`, *etc.*). Subsequently, the algorithm iterates through all pages and validates that the initialisation is correct. This means, that every page in the `origin` and the `snapshot` has the same data and shares a PFN. Finally, the algorithm sets each page in the `origin` to `TEST_VAL` which triggers the duplication of the respective page. Although this time, the algorithm validates that every page in the `origin` and the `snapshot` contains the correct data, and also differs in the PFN, respectively. To validate that every page differs in the PFN the previously described interface of `/proc/<pid>/maps` is used. The file is indexed using the virtual page number—which uses the system's default page size, that is typically 4 KiB.

Algorithm 3 Test copy-on-write

```
 1: INIT_VAL := 0xAA
 2: TEST_VAL := 0x55
 3: origin ← ALLOCATE(page_count)
 4: INITIALISE(origin, INIT_VAL)
 5: snapshot ← SNAPSHOT(origin, page_count)
 6: for page := 1 to page_count do                    ▷ validate initialisation
 7:     ASSERT(origin[page] == INIT_VAL)
 8:     ASSERT(snapshot[page] == INIT_VAL)
 9:     ASSERT(GET_PFN(origin[page]) == GET_PFN(snapshot[page]))
10: end for
11: for page := 1 to page_count do                    ▷ validate COW
12:     origin[page] ← TEST_VAL
13:     ASSERT(snapshot[page] == INIT_VAL)
14:     ASSERT(GET_PFN(origin[page]) ≠ GET_PFN(snapshot[page]))
15: end for
```

The unit test supports different page sizes, that is pages of size 4 KiB, 2 MiB and 1 GiB. Table 7 shows a matrix of each COW mechanism and every page size supported by the unit test. The unit test can either be *passed* or *failed*. There is also the keyword *not supported*.

Table 7: Copy-on-write mechanisms unit test matrix.

Mechanism	4 KiB page size	2 MiB page size	1 GiB page size
`scoot()`	passed	passed	passed
`mmapcopy()`	failed	not supported	not supported
`mremap()`	passed	passed	passed
`vmcopy()`	passed	passed	failed

Admittedly, it is surprising that some mechanisms fail in these tests. The `mmapcopy()` mechanism fails the test even with default page size because it does not flush the TLB properly. Other page sizes are not supported by the mechanism. The other kernel mechanism `vmcopy()` only fails the COW unit test when using pages of size 1 GiB. The kernel reports an error that the found PUD is a bad entry. Another issue when using 1 GiB pages with `vmcopy()` is that the de-allocation of such pages sometimes fails. This issue escalates in a kernel panic, whenever all of the available huge pages are not de-allocated properly. The implications of these results are discussed in Chapter 6.

5.2 Measurements

The measuring tool supports three different measurement variants for each mechanism. These are the snapshot creation time, the access time, and the duplication time. Additionally, the tool also supports pages of size 4 KiB, 2 MiB and 1 GiB. The measurement variants are each described in their respective chapter below.

The data was gathered on a system running the latest (v5.18 at the time of writing) Linux kernel. The system is equipped with a Intel Xeon Gold 5118 CPU and 32 GiB of DDR4 RAM clocking at 2666 MT/s. Every measurement was run 1000 times. Also, for every measurement there are two cases for allocating the origin differently. In the first case the origin was allocated using the `MAP_POPULATE` flag, which ensures that the data is actually allocated in the physical memory. This is indicated by the keyword *populated*. In the second case the origin was allocated without this flag, which means that the data will be brought into memory through a page fault. This is indicated by the keyword *unpopulated*.

5.2.1 Snapshot Creation Time

In this measurement the snapshot creation time is evaluated, that is the time it takes to create a snapshot from an origin. Basically this is the time the main program is blocked by calling the mechanism. For kernel space mechanisms this means the time the actual system call takes to complete, while for the only measured user space mechanism `scoot()`, this means the time the series of system calls takes to complete. Table 8 shows the average snapshot creation time for each mechanism with an origin of size 2 GiB.

Table 8: Average snapshot creation time of each mechanism with an origin of size 2 GiB.

	Populated			**Unpopulated**		
Mechanism	**4 KiB**	**2 MiB**	**1 GiB**	**4 KiB**	**2 MiB**	**1 GiB**
`fork()`	19.57 *ms*	297.19 *µs*	76.87 *µs*	0.06 *ms*	69.19 *µs*	56.64 *µs*
`mmapcopy()`	51.82 *ms*	∅	∅	3.81 *ms*	∅	∅
`mremap()`	19.52 *ms*	283.57 *µs*	20.30 *µs*	0.01 *ms*	20.87 *µs*	6.01 *µs*
`vmcopy()`	31.08 *ms*	806.68 *µs*	∅	0.03 *ms*	26.58 *µs*	∅
`scoot()`	9.18 *ms*	149.22 *µs*	148.66 *µs*	0.01 *ms*	54.89 *µs*	55.20 *µs*

Let us first have a look at the populated case. When using 4 KiB pages, `scoot()` is at least twice as fast as the other mechanisms, because `scoot()` does not populate the page tables. Therefore, the time is hidden in the access time as we will see in Chapter 5.2.2. Both, `fork()` and `mremap()`, have around the same snapshot creation time. This is unsurprising, since both internally use `copy_page_range()`, which determines the runtime when copying a substantial amount of pages. The other two kernel space mechanisms, `vmcopy()` and `mmapcopy()`, are slower than `fork()` and `mremap()`. They both implement their own function to copy the page tables instead of extending decades-long optimised code.

The measured snapshot creation times of the mechanisms using 2 MiB pages show a similar result. While `mmapcopy()` does not support huge pages at all, the `vmcopy()` mechanism is the slowest measured mechanism. Once again `scoot()` is the fastest mechanism, compared to `fork()` or `mremap()` the `scoot()` mechanism is about twice as fast.

However, when using 1 GiB pages, the `scoot()` mechanism appears to have hit its lower bound, since the reduction in page count has not lowered its snapshot creation time. Both, `fork()` and `mremap()`, are unexpectedly faster than `scoot()`. The `mremap()` mechanism even surpasses the `fork()` mechanism, since it only has to copy two pages. Therefore, the process creation of `fork()` dominates its snapshot creation time when the page count is low.

Now let us have a look at the unpopulated case. The `fork()` system call has a nearly constant snapshot creation time among all page sizes, since with unpopulated pages once

again the process creation dominates the snapshot creation time. However, the `mremap()` system call—even though it also uses the same `copy_page_range()` internally—does not create a process and thus has the fastest snapshot creation time across all huge page sizes. Instead of copying the present pages when using `copy_page_range()`, the kernel vigorously only allocates the page table structures. It then lets the page fault handling routine fill up the respective page table structures. The `scoot()` mechanism once again appears to have hit a lower bound when using huge pages.

Interestingly, though, this does not apply when using pages of size 4 KiB. The `mmapcopy()` system call is once again the slowest mechanism. Finally, the `vmcopy()` system call is right between `fork()` and `mremap()`. This is unsurprising, as `fork()` is bound by the process creation runtime and `mremap()` uses optimised code.

These measurements show that `scoot()` is the fastest mechanism whenever copying a substantial amount of pages. Meanwhile the `mremap()` system call is about as fast as the `fork()` system call. However, when the page count gets lower, the `mremap()` system call surpasses the `fork()` system call because it does not create an additional process. The other two mechanisms `mmapcopy()` and `vmcopy()` are both slower in every measured scenario.

5.2.2 Access Time

Let us now have a look at the access time measurement. The access time denotes the time it takes to read every page allocated in the origin. Additionally, in the populated case the data can either be *cached* or *uncached*, indicating whether or not the data was cached by the CPU.

This measurement is made to show the differences of the `scoot()` mechanism and all other mechanisms. Since `scoot()` provides mapping with unpopulated page tables, it is expected that it has a higher access time on uncached access. Because caching involves reading the data, the page tables must also be populated. Therefore, it is foreseeable that every mechanism averages the same access time in this case. Table 9 shows the average access time for a payload of 2 GiB.

Table 9: Average access time of each mechanism with an origin of size 2 GiB.

	Populated			Unpopulated		
Mechanism	**4 KiB**	**2 MiB**	**1 GiB**	**4 KiB**	**2 MiB**	**1 GiB**
Uncached						
`fork()`	27.64 *ms*	52.03 *µs*	0.33 *µs*	596.88 *ms*	157.34 *ms*	155.61 *ms*
`mmapcopy()`	25.95 *ms*	∅	∅	595.41 *ms*	∅	∅
`mremap()`	25.95 *ms*	50.97 *µs*	0.34 *µs*	595.01 *ms*	157.16 *ms*	155.48 *ms*
`vmcopy()`	25.95 *ms*	50.59 *µs*	∅	595.46 *ms*	157.30 *ms*	∅
`scoot()`	109.81 *ms*	1708.46 *µs*	15.13 *µs*	1406.99 *ms*	157.35 *ms*	0.25 *ms*
Cached						
`fork()`	26.77 *ms*	55.66 *µs*	40.30 *ns*			
`mmapcopy()`	25.85 *ms*	∅	∅			
`mremap()`	25.97 *ms*	50.60 *µs*	38.63 *ns*			
`vmcopy()`	25.94 *ms*	50.58 *µs*	∅			
`scoot()`	25.76 *ms*	48.06 *µs*	39.55 *ns*			

In the populated and uncached case the hidden snapshot creation time of the `scoot()` mechanism becomes apparent. The page tables of `scoot()` are not populated, while with all other mechanisms they are. Thus, it is no surprise that the other mechanisms have about the same access time.

In the unpopulated and uncached case, the `scoot()` mechanism is slower than the other mechanisms when using 4 KiB pages. However, as the page size increases and therefore the page count decreases, the `scoot()` mechanism gets faster when using 1 GiB pages. The preliminary results of this behaviour observed in the `scoot()` can be addressed in future research. All the other mechanisms have, as expected, the same access time.

Finally, in the populated and cached case, every mechanism averages about the same access time, independent of the page size used. This is no surprise, as the data is directly accessed from the cache.

The visualised data in Figure 13 highlights the hidden snapshot creation time of the `scoot()` mechanism. It shows the access time on the x-axis and the snapshot creation time on the y-axis. Moreover, the axis are scaled logarithmically and the x-axis scale differs based on the used page size. Instead of plotting every measurement, a random sample of 10 measurements is taken to decrease visual clutter.

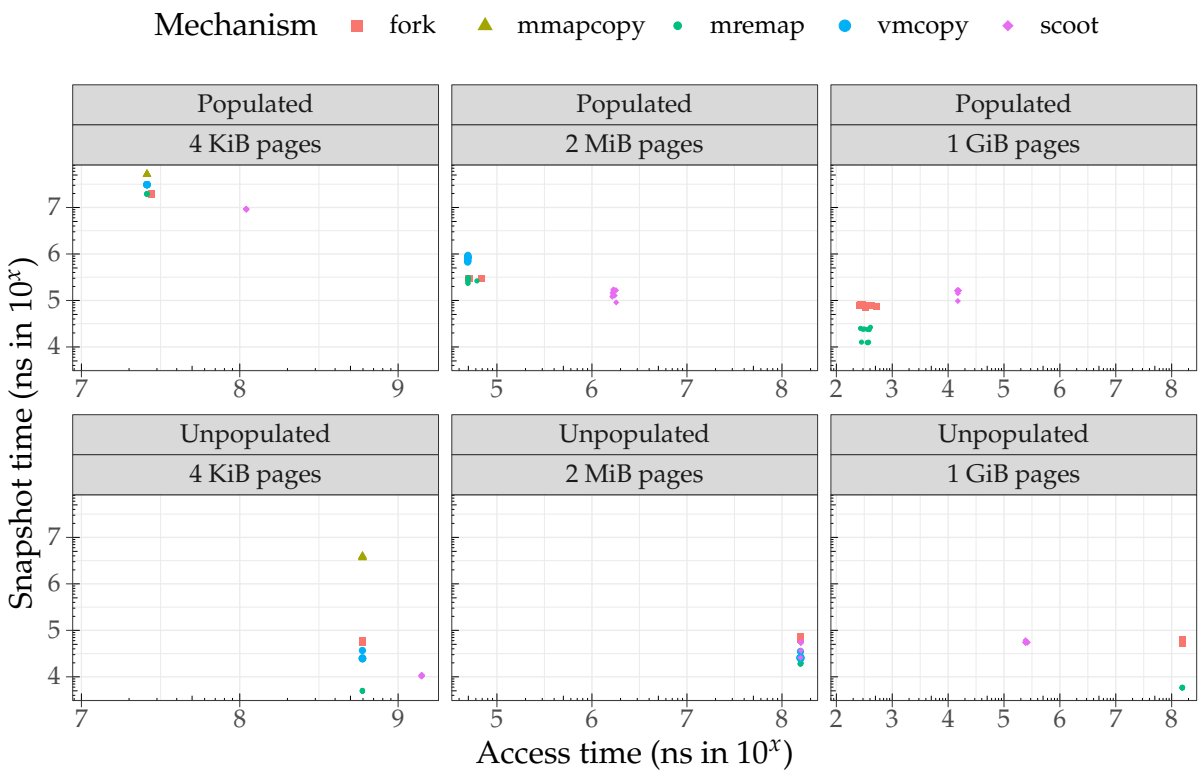


Figure 13: Snapshot creation time of each mechanism with an origin of size 2 GiB. Access time of reading every page in the origin of a 2 GiB sized snapshot. Both axis are scaled logarithmically. Note that, the scaling of the x-axis differs between page sizes. The keyword *Populated* indicates that the allocation of the origin was made using `MAP_POPULATE`, while the keyword *Unpopulated* indicates that the allocation of the origin was made without it. A bigger page size inherently means a smaller page count when allocating the same amount of memory and therefore less iterations. To decrease visual clutter a random sample of 10 measurement for each mechanism is plotted.

These measurements confirm the claim that `scoot()` hides its snapshot creation time in the access time. As expected, the other mechanisms have the same access times, as they all have populated page tables. Some applications can benefit from hiding the snapshot creation time in the access time, more on this in Chapter 6.

5.2.3 Duplication Time

The last measure variant measures the duplication time. The term *duplication* means the process of creating the actual copy whenever a COW-protected page is modified. Thus, the duplication time denotes the time it takes to modify every page in the payload, which triggers the duplication of the respective pages. Once again two cases are measured, the populated case and the unpopulated case. However, since the mechanism `mmapcopy()` does not pass the COW unit test, it is not measured. Instead, `memcpy()` is used as a kind of baseline, since

`memcpy()` shows the time of simply copying the data without using any COW protection. Table 10 shows the duplication time for modifying every page in a snapshotted payload of 2 GiB. For kernel space mechanisms, the duplication time is expected to average to the same, since all use the same internal kernel mechanism to guarantee their COW protection.

Table 10: Average duplication time of each mechanism with an origin of size 2 GiB.

	Populated			**Unpopulated**		
Mechanism	**4 KiB**	**2 MiB**	**1 GiB**	**4 KiB**	**2 MiB**	**1 GiB**
`fork()`	1375.37 *ms*	344.20 *ms*	342.04 *ms*	906.55 *ms*	158.98 *ms*	158.17 *ms*
`memcpy()`	59.16 *ms*	0.08 *ms*	293.52 **ns**	1075.96 *ms*	1.55 *ms*	0.01 *ms*
`mremap()`	1366.44 *ms*	372.13 *ms*	361.82 *ms*	904.76 *ms*	152.92 *ms*	153.37 *ms*
`vmcopy()`	1364.91 *ms*	368.89 *ms*	∅	899.66 *ms*	154.73 *ms*	∅
`scoot()`	1228.43 *ms*	363.04 *ms*	0.77 *ms*	1462.92 *ms*	153.00 *ms*	0.21 *ms*

The measurements confirm the assumption that the duplication times for the kernel space mechanisms average to about the same. However, what is interesting is that the `scoot()` mechanism duplicates faster than the other COW mechanisms when using huge pages. Again further investigations have to be conducted to explain why the `scoot()` mechanism duplicate faster under the use of huge pages.

The result of the evaluation is that the behaviour of the individual mechanisms varies based on multiple factors. First, `scoot()` differs from the other mechanisms in that the page tables are not populated. This makes its snapshot creation time the fastest among all tested. However, this does not mean that the `scoot()` mechanism actually saves time. It basically hides the supposed time savings in the access time. The measurements confirm this claim.

Second, when comparing the kernel space mechanisms, `fork()` and `mremap()` have the lowest snapshot creation time. They both internally use the same `copy_page_range()` function to copy their PTEs. The other kernel space mechanisms have implemented their own functions to copy the page tables, which appear to be not as efficient as `copy_page_range()`. The `mremap()` even surpasses the `fork()` when using huge pages, because it does not have the overhead of creating a new process.

Finally, the duplication time and the access time in the unpopulated case of `scoot()` surprisingly are faster than the other mechanisms when using huge pages. However, as this work focuses on kernel space mechanisms future work must address the surprising result of `scoot()`.

6 Discussion

This work examined what mechanisms other applications that require COW semantics use to overcome the lack of a dedicated COW mechanism in the Linux kernel. It was found that some applications [4, 8] accept the induced overhead of `fork()`—while also ignoring its semantics—to create a COW-protected memory. Although it is a convenient way to overcome the lack of a dedicated mechanism, it does not allow to select a region for the snapshot. This can be a limiting factor, if the payload that shall be snapshotted is a fraction of the total memory used by the process. However, in the cases of [4, 8], both in-memory database management systems, it is not as costly, as such a database system typically creates snapshots of its data. Since the data generally makes up most of the memory used in such a system, the inability to address the exact area in a `fork()` is not considered as a severe limitation. Additionally, the spawned process can be used to do the required input and output operations when, for example, persisting the system's data. Admittedly, one can also use threads to do the same, however, the additional overhead of creating a new process is not completely wasted in these applications.

Other applications even use their own kernel modification to overcome the lack of a dedicated COW mechanism. Korb *et al.*, for example, implemented their own system call `mmapcopy()` to reduce the memory usage of large vectors in R [15]. This system call provides a dedicated interface to create COW mappings. Another example of a kernel modification was implemented by Sharma *et al.* In [10] they use their system call `vmcopy()` to implement multi-version concurrency control in their database system. Their system call also provides a dedicated kernel interface to create a COW mapping.

Summarising the results for this research question, it is possible to overcome the lack of a dedicated COW mechanism in the Linux kernel. Applications either exploit the existing system call `fork()` to create a COW mapping of a whole process's address space, or implement their own dedicated system call to create a COW mapping of a given VMA. However, this means application developers have to make a choice to either exploit `fork()` and accept the implied possible caveats, or to modify their kernel—which allows for a fine-granular mechanism, but requires their application to run on a modified operating system—which reduces accessibility, portability, and perhaps even compromises security.

The second goal of this work was to answer the research question of how the proposed `MREMAP_COW` flag enhances the Linux kernel compared to the other researched mechanisms under the use of quantitative methods. To answer this question, let us first discuss how the other mechanisms comply with our set of requirements of a COW mechanism. The set of requirements for such a mechanism are as follows. First, the mechanism has to create a snapshot memory mapping from an origin memory mapping, where any modification in

the origin triggers the duplication. Second, the duplication must be handled by the Linux kernel. Third, the mechanism shall be as performant as possible. To aggregate valuable quantitative information about each mechanism, a unit test and a measurement tool were implemented. However, when testing and measuring the presented mechanisms, it was found that nearly every mechanism does not comply with at least one requirement.

Kernel Space Mechanisms

The first analysed kernel space mechanism is the `fork()` system call, which creates a new process from an existing process. When creating a new process, `fork()` copies the whole virtual address space of the existing process to the new process under the use of COW. This behaviour can be exploited to create new COW mappings. The biggest advantage of using `fork()` is that it does not require a kernel modification, nor application-specific handling. The results showed that its performance is among the fastest of all mechanisms. However, the results also revealed that when only copying a small amount of pages, the process creation quickly becomes a significant overhead. Moreover, when exploiting `fork()` to create a COW mapping, there is no option to choose which region to snapshot, the system call simply snapshots the whole virtual address space. Depending on the requirements of the particular application, this can also introduce an undesirable overhead.

The second examined kernel space mechanism is the `mmapcopy()` [15] system call. This mechanism only supports the use of default pages. However, the unit test of this work showed that `mmapcopy()` does not comply to the set of requirements. It failed the unit test because it did not duplicate the pages properly. While investigating it was found that the mechanism does not flush the TLB properly. The undefined behaviour of this faulty implementation can introduce severe security vulnerabilities. Additionally, the measured snapshot creation time is also the slowest of all mechanisms.

The third analysed kernel space mechanism is the `vmcopy()` [10] system call from the AnKer [63] kernel modification. This mechanism supposedly supports huge pages as it requires the kernel configuration setting of transparent huge pages. However, the unit test revealed that the use of 1 GiB pages not only does not work, but also does not de-allocate system resources properly. The improper handling of system resources can lead to a kernel panic, compromising the stability of a system. Nevertheless, the performance measurements showed that `vmcopy()` is not significantly slower than a `fork()`.

Finally, the last researched kernel space mechanism is the extension of `mremap()`. The kernel modification supports all three tested page sizes of 4 KiB, 2 MiB and 1 GiB and passed the unit tests for all these sizes. The results of the conducted measurements show that it is the fastest of all kernel space mechanisms in regards to snapshot creation time. Additionally, it allows to select the VMA that shall be duplicated. This potentially removes an unwanted

overhead. However, when copying a huge amount of pages `fork()` is as fast as `mremap()`, since it uses the same internal function to copy the page tables.

User Space Mechanisms

The first researched user space mechanism is the `scoot()` mechanism. This mechanism uses a series of `mmap()` and `mremap()` to create a COW mapping, thus, it does not require any kernel modification. The mechanism passed the unit test for all page sizes. Additionally, the mechanism also handles its duplication using the Linux kernel. However, `scoot()` requires a file descriptor, for example, through the use of `memfd_create()`. The requirement of a file descriptor and the fact that it remaps the origin to create a snapshot, induce a variety of inconveniences. First, the allocation of memory resources must handle the requirement of a file descriptor. Classic allocators such as glibc's `malloc()` [66] or the general purpose allocator `jemalloc()` [67] typically do not support the use of a file descriptor. This means that users of `scoot()` have to implement their own custom allocation routine to handle the required file descriptor. Second, `scoot()` is limited to a single snapshot, because during the creation of the snapshot, the origin is remapped as a *private* VMA. However, to create a new snapshot the `scoot()` mechanism requires the origin to be mapped as a *shared* VMA. Finally, because `scoot()` changes the location of its VMAs rigorously, the respective page tables are not populated. The results showed that a consequence of the unpopulated page tables is that some of the snapshot creation time of `scoot()` is hidden in the access time. Thus, `scoot()` has the fastest snapshot creation time of all mechanisms among all page sizes, but also a slow access time. Whether an application can benefit from this snapshot creation time hiding or not depends on the application itself. An in-memory database, for example, can benefit from this time hiding, as reducing the time the database gets blocked when creating a snapshot can result in faster response times. The additional time required to access the data is only hindering performance, if the access time of the origin is slowed down. However, with `scoot()` that is the case, as the page tables of origin and snapshot are unpopulated.

The second examined user space mechanism is the UFFD [47] mechanism. The Linux kernel offers this interface to allow user space to intervene in the handling of page faults. With the help of an application-specific implementation, users of UFFD can decide on a page-granular level whether individual pages should be duplicated or not. Based on the implementation this can be very performant. However, as a application-specific implementation is required, no unit tests or measurements were made for the UFFD mechanism. Moreover, comparing such a sophisticated application-specific solution to a general approach solution would not be appropriate.

In summary, the results of this work show that the proposed kernel modification of the system call `mremap()` with an additional flag called `MREMAP_COW` extends the Linux kernel in the following ways.

First, it provides a dedicated interface to create a COW mapping, thus preventing users from exploiting `fork()`. The literature research showed that there are cases where applications exploit the system call to create COW mappings. However, the incapability to select the data regions to create such mappings can introduce unwanted overhead. Furthermore, it was shown that the process creation of `fork()` can also introduce additional overhead when dealing with a small amount of pages. On the contrary, the `mremap()` mechanism can address individual VMAs and operate on a page-granular level, while also skipping process creation.

Second, users are discouraged to implement their own perhaps faulty version of such a mechanism. Both other examined kernel modifications either introduce severe security vulnerabilities or compromise system stability. However, when proposing an extension to the Linux kernel, an iterative review process begins so that any bugs introduced can be found before they are incorporated into the upstream. This can increase the security and stability of applications that implement their own COW mechanism, since the Linux kernel is lacking such a dedicated mechanism.

Finally, further optimisations of the mechanism can be implemented if the extension is merged into upstream. Since the Linux kernel is an open source project, everyone can propose changes to the mechanism. This potentially increases the performance, security and stability of the mechanism, because many developers of different expertise can then propose changes themselves. Overall, the extension of the Linux kernel with an additional flag called `MREMAP_COW` provides a performant and efficient mechanism to create a page-granular COW memory mapping. Moreover, applications that either exploit `fork()` or use their own kernel modification to create snapshots can also benefit from an upstream merge, since these applications could instead use the extended `mremap()` system call to create these snapshots.

7 Conclusion and Outlook

In this work, a new flag for the Linux kernel system call `mremap()` called `MREMAP_COW` was proposed to create COW-protected memory mappings from any existing private anonymous VMA. The work also addressed two research questions related to the field of open source operating system development.

First, it analysed how applications overcome the lack of a dedicated COW mechanism in the Linux kernel. It was found that basically there are two ways: either the application exploits `fork()` for its creation of a COW mapping, or developers implement their own kernel modification. The results of this work showed that both approaches have different caveats. While users of `fork()` do not have the option to choose which VMA to snapshot, users of kernel modifications like `mmapcopy()` or `vmcopy()` are not always safe to use, as the results revealed. They either compromise stability or introduce undefined behaviour into the memory management of the system.

The second research question that was addressed, is how does the proposed Linux kernel extension compare to the existing mechanisms. The conducted experiments of this work suggest that not only is the proposed `MREMAP_COW` among the fastest mechanisms evaluated, but it was also the only kernel space mechanism to pass the implemented unit test in all page sizes. This suggests that it is the fastest mechanism, while also being the safest to use.

Next, in an effort to bring the changes upstream, the extension of `mremap()` through the addition of a new flag called `MREMAP_COW` will be proposed to the Linux kernel memory management mailing list. This initiates an iterative review process, where members of the Linux kernel community will examine the proposed extension thoroughly. This can potentially improve the mechanism in several ways, as developers with different expertise can suggest changes.

Furthermore, the mechanism can also be further optimised for different use cases. For example, in the case of an in-memory database system, it is reasonable that the snapshot creation time shall be as low as possible, while accepting longer access times in the snapshot. To do this, the mechanism can hide some of the snapshot creation time in the access time of the snapshot by creating a COW-protected VMA preemptively and let the page fault handling routine fill the page tables whenever accessed. Therefore, future work can address both identifying different use cases as well as further optimising the mechanism.

List of Figures

List of Tables

List of Algorithms

List of Acronyms

COW copy-on-write
KSM Kernel Samepage Merging
LRU least recently used
MMU memory management unit
OLAP Online Analytical Processing
OLTP Online Transaction Processing
PFN page frame number
PGD page global directory
PMD page middle directory
PTE page table entry
PUD page upper directory
TLB Translation Lookaside Buffer
UFFD userfaultfd
VM Virtual Memory
VMA virtual memory area

A MREMAP_COW Support

```
From 9cb496cad66e78cdfbe0d3ea46cf78e54ad0074a Mon Sep 17 00:00:00 2001
From: Mario Mintel <mariomintel@gmail.com>
Date: Wed, 1 Jun 2022 10:16:00 +0200
Subject: mm: add MREMAP_COW flag to mremap

---
 include/uapi/linux/mman.h       |   1 +
 mm/memory.c                     | 138 +++++++++++++++++---------------
 mm/mremap.c                     |  24 ++++--
 tools/include/uapi/linux/mman.h |   1 +
 4 files changed, 93 insertions(+), 71 deletions(-)

diff --git a/include/uapi/linux/mman.h b/include/uapi/linux/mman.h
index f55bc680b5b0..634217617de5 100644
--- a/include/uapi/linux/mman.h
+++ b/include/uapi/linux/mman.h
@@ -8,6 +8,7 @@
 #define MREMAP_MAYMOVE         1
 #define MREMAP_FIXED           2
 #define MREMAP_DONTUNMAP       4
+#define MREMAP_COW             8

 #define OVERCOMMIT_GUESS               0
 #define OVERCOMMIT_ALWAYS              1
diff --git a/mm/memory.c b/mm/memory.c
index 76e3af9639d9..a89ab8792ba6 100644
--- a/mm/memory.c
+++ b/mm/memory.c
@@ -770,8 +770,9 @@ try_restore_exclusive_pte(pte_t *src_pte, struct vm_area_struct *vma,

 static unsigned long
 copy_nonpresent_pte(struct mm_struct *dst_mm, struct mm_struct *src_mm,
-               pte_t *dst_pte, pte_t *src_pte, struct vm_area_struct *dst_vma,
-               struct vm_area_struct *src_vma, unsigned long addr, int *rss)
+                   pte_t *dst_pte, pte_t *src_pte, struct vm_area_struct *dst_vma,
+                   struct vm_area_struct *src_vma, unsigned long src_addr,
+                   unsigned long dst_addr, int *rss)
 {
        unsigned long vm_flags = dst_vma->vm_flags;
        pte_t pte = *src_pte;
@@ -809,7 +810,7 @@ copy_nonpresent_pte(struct mm_struct *dst_mm, struct mm_struct *src_mm,
                                pte = pte_swp_mksoft_dirty(pte);
                        if (pte_swp_uffd_wp(*src_pte))
                                pte = pte_swp_mkuffd_wp(pte);
-                       set_pte_at(src_mm, addr, src_pte, pte);
+                       set_pte_at(src_mm, src_addr, src_pte, pte);
                }
        } else if (is_device_private_entry(entry)) {
                page = pfn_swap_entry_to_page(entry);
@@ -841,7 +842,7 @@ copy_nonpresent_pte(struct mm_struct *dst_mm, struct mm_struct *src_mm,
                        pte = swp_entry_to_pte(entry);
                        if (pte_swp_uffd_wp(*src_pte))
                                pte = pte_swp_mkuffd_wp(pte);
-                       set_pte_at(src_mm, addr, src_pte, pte);
+                       set_pte_at(src_mm, src_addr, src_pte, pte);
                }
        } else if (is_device_exclusive_entry(entry)) {
                /*
@@ -851,13 +852,13 @@ copy_nonpresent_pte(struct mm_struct *dst_mm, struct mm_struct *src_mm,
                 * (ie. COW) mappings.
                 */
                VM_BUG_ON(!is_cow_mapping(src_vma->vm_flags));
-               if (try_restore_exclusive_pte(src_pte, src_vma, addr))
+               if (try_restore_exclusive_pte(src_pte, src_vma, src_addr))
                        return -EBUSY;
                return -ENOENT;
        }
        if (!userfaultfd_wp(dst_vma))
                pte = pte_swp_clear_uffd_wp(pte);
-       set_pte_at(dst_mm, addr, dst_pte, pte);
+       set_pte_at(dst_mm, dst_addr, dst_pte, pte);
        return 0;
 }

@@ -883,8 +884,8 @@ copy_nonpresent_pte(struct mm_struct *dst_mm, struct mm_struct *src_mm,
  */
 static inline int
 copy_present_page(struct vm_area_struct *dst_vma, struct vm_area_struct *src_vma,
-                 pte_t *dst_pte, pte_t *src_pte, unsigned long addr, int *rss,
-                 struct page **prealloc, pte_t pte, struct page *page)
+                 pte_t *dst_pte, pte_t *src_pte, unsigned long dst_addr, unsigned long src_addr,
```

```
+                int *rss, struct page **prealloc, pte_t pte, struct page *page)
 {
        struct page *new_page;

@@ -913,9 +914,9 @@ copy_present_page(struct vm_area_struct *dst_vma, struct vm_area_struct *src_vma
         * over and copy the page & arm it.
         */
        *prealloc = NULL;
-       copy_user_highpage(new_page, page, addr, src_vma);
+       copy_user_highpage(new_page, page, src_addr, src_vma);
        __SetPageUptodate(new_page);
-       page_add_new_anon_rmap(new_page, dst_vma, addr, false);
+       page_add_new_anon_rmap(new_page, dst_vma, dst_addr, false);
        lru_cache_add_inactive_or_unevictable(new_page, dst_vma);
        rss[mm_counter(new_page)]++;

@@ -925,7 +926,7 @@ copy_present_page(struct vm_area_struct *dst_vma, struct vm_area_struct *src_vma
        if (userfaultfd_pte_wp(dst_vma, *src_pte))
                /* Uffd-wp needs to be delivered to dest pte as well */
                pte = pte_wrprotect(pte_mkuffd_wp(pte));
-       set_pte_at(dst_vma->vm_mm, addr, dst_pte, pte);
+       set_pte_at(dst_vma->vm_mm, dst_addr, dst_pte, pte);
        return 0;
 }

@@ -935,20 +936,20 @@ copy_present_page(struct vm_area_struct *dst_vma, struct vm_area_struct *src_vma
  */
 static inline int
 copy_present_pte(struct vm_area_struct *dst_vma, struct vm_area_struct *src_vma,
-                pte_t *dst_pte, pte_t *src_pte, unsigned long addr, int *rss,
-                struct page **prealloc)
+                pte_t *dst_pte, pte_t *src_pte, unsigned long dst_addr, unsigned long src_addr,
+                int *rss, struct page **prealloc)
 {
        struct mm_struct *src_mm = src_vma->vm_mm;
        unsigned long vm_flags = src_vma->vm_flags;
        pte_t pte = *src_pte;
        struct page *page;

-       page = vm_normal_page(src_vma, addr, pte);
+       page = vm_normal_page(src_vma, src_addr, pte);
        if (page) {
                int retval;

                retval = copy_present_page(dst_vma, src_vma, dst_pte, src_pte,
-                                          addr, rss, prealloc, pte, page);
+                                          dst_addr, src_addr, rss, prealloc, pte, page);
                if (retval <= 0)
                        return retval;

@@ -962,7 +963,7 @@ copy_present_pte(struct vm_area_struct *dst_vma, struct vm_area_struct *src_vma,
         * in the parent and the child
         */
        if (is_cow_mapping(vm_flags) && pte_write(pte)) {
-               ptep_set_wrprotect(src_mm, addr, src_pte);
+               ptep_set_wrprotect(src_mm, src_addr, src_pte);
                pte = pte_wrprotect(pte);
        }

@@ -977,7 +978,7 @@ copy_present_pte(struct vm_area_struct *dst_vma, struct vm_area_struct *src_vma,
        if (!userfaultfd_wp(dst_vma))
                pte = pte_clear_uffd_wp(pte);

-       set_pte_at(dst_vma->vm_mm, addr, dst_pte, pte);
+       set_pte_at(dst_vma->vm_mm, dst_addr, dst_pte, pte);
        return 0;
 }

@@ -1002,8 +1003,8 @@ page_copy_prealloc(struct mm_struct *src_mm, struct vm_area_struct *vma,

 static int
 copy_pte_range(struct vm_area_struct *dst_vma, struct vm_area_struct *src_vma,
-              pmd_t *dst_pmd, pmd_t *src_pmd, unsigned long addr,
-              unsigned long end)
+              pmd_t *dst_pmd, pmd_t *src_pmd, unsigned long dst_addr, unsigned long dst_end,
+              unsigned long src_addr, unsigned long src_end)
 {
        struct mm_struct *dst_mm = dst_vma->vm_mm;
        struct mm_struct *src_mm = src_vma->vm_mm;
@@ -1019,12 +1020,12 @@ copy_pte_range(struct vm_area_struct *dst_vma, struct vm_area_struct *src_vma,
        progress = 0;
        init_rss_vec(rss);

-       dst_pte = pte_alloc_map_lock(dst_mm, dst_pmd, addr, &dst_ptl);
+       dst_pte = pte_alloc_map_lock(dst_mm, dst_pmd, dst_addr, &dst_ptl);
        if (!dst_pte) {
```

```
                ret = -ENOMEM;
                goto out;
        }
-       src_pte = pte_offset_map(src_pmd, addr);
+       src_pte = pte_offset_map(src_pmd, src_addr);
        src_ptl = pte_lockptr(src_mm, src_pmd);
        spin_lock_nested(src_ptl, SINGLE_DEPTH_NESTING);
        orig_src_pte = src_pte;
@@ -1050,7 +1051,7 @@ copy_pte_range(struct vm_area_struct *dst_vma, struct vm_area_struct *src_vma,
                        ret = copy_nonpresent_pte(dst_mm, src_mm,
                                                  dst_pte, src_pte,
                                                  dst_vma, src_vma,
-                                                 addr, rss);
+                                                 dst_addr, src_addr, rss);
                        if (ret == -EIO) {
                                entry = pte_to_swp_entry(*src_pte);
                                break;
@@ -1069,7 +1070,7 @@ copy_pte_range(struct vm_area_struct *dst_vma, struct vm_area_struct *src_vma,
                }
                /* copy_present_pte() will clear `*prealloc` if consumed */
                ret = copy_present_pte(dst_vma, src_vma, dst_pte, src_pte,
-                                      addr, rss, &prealloc);
+                                      dst_addr, src_addr, rss, &prealloc);
                /*
                 * If we need a pre-allocated page for this pte, drop the
                 * locks, allocate, and try again.
@@ -1087,7 +1088,8 @@ copy_pte_range(struct vm_area_struct *dst_vma, struct vm_area_struct *src_vma,
                        prealloc = NULL;
                }
                progress += 8;
-       } while (dst_pte++, src_pte++, addr += PAGE_SIZE, addr != end);
+       } while (dst_pte++, src_pte++, src_addr += PAGE_SIZE, dst_addr += PAGE_SIZE,
+                src_addr != src_end || dst_addr != dst_end);

        arch_leave_lazy_mmu_mode();
        spin_unlock(src_ptl);
@@ -1106,7 +1108,7 @@ copy_pte_range(struct vm_area_struct *dst_vma, struct vm_area_struct *src_vma,
        } else if (ret == -EBUSY) {
                goto out;
        } else if (ret ==  -EAGAIN) {
-               prealloc = page_copy_prealloc(src_mm, src_vma, addr);
+               prealloc = page_copy_prealloc(src_mm, src_vma, src_addr);
                if (!prealloc)
                        return -ENOMEM;
        } else if (ret) {
@@ -1116,7 +1118,7 @@ copy_pte_range(struct vm_area_struct *dst_vma, struct vm_area_struct *src_vma,
        /* We've captured and resolved the error. Reset, try again. */
        ret = 0;

-       if (addr != end)
+       if (src_addr != src_end)
                goto again;
 out:
        if (unlikely(prealloc))
@@ -1126,26 +1128,27 @@ copy_pte_range(struct vm_area_struct *dst_vma, struct vm_area_struct *src_vma,

 static inline int
 copy_pmd_range(struct vm_area_struct *dst_vma, struct vm_area_struct *src_vma,
-              pud_t *dst_pud, pud_t *src_pud, unsigned long addr,
-              unsigned long end)
+              pud_t *dst_pud, pud_t *src_pud, unsigned long dst_addr, unsigned long dst_end,
+              unsigned long src_addr, unsigned long src_end)
 {
        struct mm_struct *dst_mm = dst_vma->vm_mm;
        struct mm_struct *src_mm = src_vma->vm_mm;
        pmd_t *src_pmd, *dst_pmd;
-       unsigned long next;
+       unsigned long src_next, dst_next;

-       dst_pmd = pmd_alloc(dst_mm, dst_pud, addr);
+       dst_pmd = pmd_alloc(dst_mm, dst_pud, dst_addr);
        if (!dst_pmd)
                return -ENOMEM;
-       src_pmd = pmd_offset(src_pud, addr);
+       src_pmd = pmd_offset(src_pud, src_addr);
        do {
-               next = pmd_addr_end(addr, end);
+               src_next = pmd_addr_end(src_addr, src_end);
+               dst_next = pmd_addr_end(dst_addr, dst_end);
                if (is_swap_pmd(*src_pmd) || pmd_trans_huge(*src_pmd)
                        || pmd_devmap(*src_pmd)) {
                        int err;
-                       VM_BUG_ON_VMA(next-addr != HPAGE_PMD_SIZE, src_vma);
+                       VM_BUG_ON_VMA(src_next-src_addr != HPAGE_PMD_SIZE, src_vma);
```

```
                        err = copy_huge_pmd(dst_mm, src_mm, dst_pmd, src_pmd,
-                                            addr, dst_vma, src_vma);
+                                            src_addr, dst_vma, src_vma);
                        if (err == -ENOMEM)
                                return -ENOMEM;
                        if (!err)
@@ -1155,34 +1158,36 @@ copy_pmd_range(struct vm_area_struct *dst_vma, struct vm_area_struct *src_vma,
                if (pmd_none_or_clear_bad(src_pmd))
                        continue;
                if (copy_pte_range(dst_vma, src_vma, dst_pmd, src_pmd,
-                                  addr, next))
+                                  dst_addr, dst_next, src_addr, src_next))
                        return -ENOMEM;
-       } while (dst_pmd++, src_pmd++, addr = next, addr != end);
+       } while (dst_pmd++, src_pmd++, src_addr = src_next, dst_addr = dst_next,
+                src_addr != src_end || dst_addr != dst_end);
        return 0;
 }

 static inline int
 copy_pud_range(struct vm_area_struct *dst_vma, struct vm_area_struct *src_vma,
-              p4d_t *dst_p4d, p4d_t *src_p4d, unsigned long addr,
-              unsigned long end)
+              p4d_t *dst_p4d, p4d_t *src_p4d, unsigned long dst_addr, unsigned long dst_end,
+              unsigned long src_addr, unsigned long src_end)
 {
        struct mm_struct *dst_mm = dst_vma->vm_mm;
        struct mm_struct *src_mm = src_vma->vm_mm;
        pud_t *src_pud, *dst_pud;
-       unsigned long next;
+       unsigned long src_next, dst_next;

-       dst_pud = pud_alloc(dst_mm, dst_p4d, addr);
+       dst_pud = pud_alloc(dst_mm, dst_p4d, dst_addr);
        if (!dst_pud)
                return -ENOMEM;
-       src_pud = pud_offset(src_p4d, addr);
+       src_pud = pud_offset(src_p4d, src_addr);
        do {
-               next = pud_addr_end(addr, end);
+               src_next = pud_addr_end(src_addr, src_end);
+               dst_next = pud_addr_end(dst_addr, dst_end);
                if (pud_trans_huge(*src_pud) || pud_devmap(*src_pud)) {
                        int err;

-                       VM_BUG_ON_VMA(next-addr != HPAGE_PUD_SIZE, src_vma);
+                       VM_BUG_ON_VMA(src_next-src_addr != HPAGE_PUD_SIZE, src_vma);
                        err = copy_huge_pud(dst_mm, src_mm,
-                                           dst_pud, src_pud, addr, src_vma);
+                                           dst_pud, src_pud, src_addr, src_vma);
                        if (err == -ENOMEM)
                                return -ENOMEM;
                        if (!err)
@@ -1192,33 +1197,36 @@ copy_pud_range(struct vm_area_struct *dst_vma, struct vm_area_struct *src_vma,
                if (pud_none_or_clear_bad(src_pud))
                        continue;
                if (copy_pmd_range(dst_vma, src_vma, dst_pud, src_pud,
-                                  addr, next))
+                                  dst_addr, dst_next, src_addr, src_next))
                        return -ENOMEM;
-       } while (dst_pud++, src_pud++, addr = next, addr != end);
+       } while (dst_pud++, src_pud++, src_addr = src_next, dst_addr = dst_next,
+                src_addr != src_end || dst_addr != dst_end);
        return 0;
 }

 static inline int
 copy_p4d_range(struct vm_area_struct *dst_vma, struct vm_area_struct *src_vma,
-              pgd_t *dst_pgd, pgd_t *src_pgd, unsigned long addr,
-              unsigned long end)
+              pgd_t *dst_pgd, pgd_t *src_pgd, unsigned long dst_addr,
+              unsigned long dst_end, unsigned long src_addr, unsigned long src_end)
 {
        struct mm_struct *dst_mm = dst_vma->vm_mm;
        p4d_t *src_p4d, *dst_p4d;
-       unsigned long next;
+       unsigned long src_next, dst_next;

-       dst_p4d = p4d_alloc(dst_mm, dst_pgd, addr);
+       dst_p4d = p4d_alloc(dst_mm, dst_pgd, dst_addr);
        if (!dst_p4d)
                return -ENOMEM;
-       src_p4d = p4d_offset(src_pgd, addr);
+       src_p4d = p4d_offset(src_pgd, src_addr);
        do {
```

```
-               next = p4d_addr_end(addr, end);
+               src_next = p4d_addr_end(src_addr, src_end);
+               dst_next = p4d_addr_end(dst_addr, dst_end);
                if (p4d_none_or_clear_bad(src_p4d))
                        continue;
                if (copy_pud_range(dst_vma, src_vma, dst_p4d, src_p4d,
-                                  addr, next))
+                                  dst_addr, dst_next, src_addr, src_next))
                        return -ENOMEM;
-       } while (dst_p4d++, src_p4d++, addr = next, addr != end);
+       } while (dst_p4d++, src_p4d++, src_addr = src_next, dst_addr = dst_next,
+                src_addr != src_end || dst_addr != dst_end);
        return 0;
 }

@@ -1226,9 +1234,11 @@ int
 copy_page_range(struct vm_area_struct *dst_vma, struct vm_area_struct *src_vma)
 {
        pgd_t *src_pgd, *dst_pgd;
-       unsigned long next;
-       unsigned long addr = src_vma->vm_start;
-       unsigned long end = src_vma->vm_end;
+       unsigned long src_next, dst_next;
+       unsigned long src_addr = src_vma->vm_start;
+       unsigned long src_end = src_vma->vm_end;
+       unsigned long dst_addr = dst_vma->vm_start;
+       unsigned long dst_end = dst_vma->vm_end;
        struct mm_struct *dst_mm = dst_vma->vm_mm;
        struct mm_struct *src_mm = src_vma->vm_mm;
        struct mmu_notifier_range range;
@@ -1268,7 +1278,7 @@ copy_page_range(struct vm_area_struct *dst_vma, struct vm_area_struct *src_vma)

        if (is_cow) {
                mmu_notifier_range_init(&range, MMU_NOTIFY_PROTECTION_PAGE,
-                                       0, src_vma, src_mm, addr, end);
+                                       0, src_vma, src_mm, src_addr, src_end);
                mmu_notifier_invalidate_range_start(&range);
                /*
                 * Disabling preemption is not needed for the write side, as
@@ -1282,18 +1292,20 @@ copy_page_range(struct vm_area_struct *dst_vma, struct vm_area_struct *src_vma)
        }

        ret = 0;
-       dst_pgd = pgd_offset(dst_mm, addr);
-       src_pgd = pgd_offset(src_mm, addr);
+       dst_pgd = pgd_offset(dst_mm, dst_addr);
+       src_pgd = pgd_offset(src_mm, src_addr);
        do {
-               next = pgd_addr_end(addr, end);
+               src_next = pgd_addr_end(src_addr, src_end);
+               dst_next = pgd_addr_end(dst_addr, dst_end);
                if (pgd_none_or_clear_bad(src_pgd))
                        continue;
                if (unlikely(copy_p4d_range(dst_vma, src_vma, dst_pgd, src_pgd,
-                                           addr, next))) {
+                                           dst_addr, dst_next, src_addr, src_next))) {
                        ret = -ENOMEM;
                        break;
                }
-       } while (dst_pgd++, src_pgd++, addr = next, addr != end);
+       } while (dst_pgd++, src_pgd++, src_addr = src_next, dst_addr = dst_next,
+                src_addr != src_end || dst_addr != dst_end);

        if (is_cow) {
                raw_write_seqcount_end(&src_mm->write_protect_seq);
diff --git a/mm/mremap.c b/mm/mremap.c
index 0b93fac76851..94b78b91aa98 100644
--- a/mm/mremap.c
+++ b/mm/mremap.c
@@ -629,12 +629,18 @@ static unsigned long move_vma(struct vm_area_struct *vma,
                return -ENOMEM;
        }

-       moved_len = move_page_tables(vma, old_addr, new_vma, new_addr, old_len,
-                                    need_rmap_locks);
-       if (moved_len < old_len) {
-               err = -ENOMEM;
-       } else if (vma->vm_ops && vma->vm_ops->mremap) {
-               err = vma->vm_ops->mremap(new_vma);
+       if (flags & MREMAP_COW) {
+               err = copy_page_range(new_vma, vma);
+               if (!err)
+                       flush_tlb_range(new_vma, old_addr, old_addr + old_len);
+       } else {
+               moved_len = move_page_tables(vma, old_addr, new_vma,new_addr,
```

```
+                                        old_len, need_rmap_locks);
+               if (moved_len < old_len) {
+                       err = -ENOMEM;
+               } else if (vma->vm_ops && vma->vm_ops->mremap) {
+                       err = vma->vm_ops->mremap(new_vma);
+               }
        }

        if (unlikely(err)) {
@@ -914,7 +920,7 @@ SYSCALL_DEFINE5(mremap, unsigned long, addr, unsigned long, old_len,
         */
        addr = untagged_addr(addr);

-       if (flags & ~(MREMAP_FIXED | MREMAP_MAYMOVE | MREMAP_DONTUNMAP))
+       if (flags & ~(MREMAP_FIXED | MREMAP_MAYMOVE | MREMAP_DONTUNMAP | MREMAP_COW))
                return ret;

        if (flags & MREMAP_FIXED && !(flags & MREMAP_MAYMOVE))
@@ -927,6 +933,8 @@ SYSCALL_DEFINE5(mremap, unsigned long, addr, unsigned long, old_len,
        if (flags & MREMAP_DONTUNMAP &&
                        (!(flags & MREMAP_MAYMOVE) || old_len != new_len))
                return ret;
+       if (flags & MREMAP_COW && !(flags & MREMAP_DONTUNMAP))
+               return ret;


        if (offset_in_page(addr))
@@ -971,7 +979,7 @@ SYSCALL_DEFINE5(mremap, unsigned long, addr, unsigned long, old_len,
                        goto out;
        }

-       if (flags & (MREMAP_FIXED | MREMAP_DONTUNMAP)) {
+       if (flags & (MREMAP_FIXED | MREMAP_DONTUNMAP | MREMAP_COW)) {
                ret = mremap_to(addr, old_len, new_addr, new_len,
                                &locked, flags, &uf, &uf_unmap_early,
                                &uf_unmap);
diff --git a/tools/include/uapi/linux/mman.h b/tools/include/uapi/linux/mman.h
index f55bc680b5b0..634217617de5 100644
--- a/tools/include/uapi/linux/mman.h
+++ b/tools/include/uapi/linux/mman.h
@@ -8,6 +8,7 @@
 #define MREMAP_MAYMOVE         1
 #define MREMAP_FIXED           2
 #define MREMAP_DONTUNMAP       4
+#define MREMAP_COW             8

 #define OVERCOMMIT_GUESS               0
 #define OVERCOMMIT_ALWAYS              1
--
```

B HUGETLB Support

```
From 9332fc795008267daba90ba7fa90c80099e2e531 Mon Sep 17 00:00:00 2001
From: Mario Mintel <mariomintel@gmail.com>
Date: Thu, 14 Jul 2022 11:17:01 +0200
Subject: mm: add hugetlb support for MREMAP_COW

---
 include/linux/hugetlb.h |  3 ++-
 mm/hugetlb.c            | 47 ++++++++++++++++++++++++++++-------------
 mm/memory.c             |  2 +-
 mm/mremap.c             |  2 ++
 4 files changed, 30 insertions(+), 24 deletions(-)

diff --git a/include/linux/hugetlb.h b/include/linux/hugetlb.h
index ac2a1d758a80..a66262e9a8c4 100644
--- a/include/linux/hugetlb.h
+++ b/include/linux/hugetlb.h
@@ -137,7 +137,8 @@ int move_hugetlb_page_tables(struct vm_area_struct *vma,
                             struct vm_area_struct *new_vma,
                             unsigned long old_addr, unsigned long new_addr,
                             unsigned long len);
-int copy_hugetlb_page_range(struct mm_struct *, struct mm_struct *, struct vm_area_struct *);
+int copy_hugetlb_page_range(struct mm_struct *, struct mm_struct *, struct vm_area_struct *,
+                            struct vm_area_struct *);
 long follow_hugetlb_page(struct mm_struct *, struct vm_area_struct *,
                          struct page **, struct vm_area_struct **,
                          unsigned long *, unsigned long *, long, unsigned int,
diff --git a/mm/hugetlb.c b/mm/hugetlb.c
index 3fc721789743..1eeb8af25bb0 100644
--- a/mm/hugetlb.c
+++ b/mm/hugetlb.c
@@ -4699,23 +4699,24 @@ hugetlb_install_page(struct vm_area_struct *vma, pte_t *ptep, unsigned long addr
 }

 int copy_hugetlb_page_range(struct mm_struct *dst, struct mm_struct *src,
-                            struct vm_area_struct *vma)
+                            struct vm_area_struct *dst_vma, struct vm_area_struct *src_vma)
 {
        pte_t *src_pte, *dst_pte, entry, dst_entry;
        struct page *ptepage;
-       unsigned long addr;
-       bool cow = is_cow_mapping(vma->vm_flags);
-       struct hstate *h = hstate_vma(vma);
+       unsigned long src_addr = src_vma->vm_start;
+       unsigned long dst_addr = dst_vma->vm_start;
+       bool cow = is_cow_mapping(src_vma->vm_flags);
+       struct hstate *h = hstate_vma(src_vma);
        unsigned long sz = huge_page_size(h);
        unsigned long npages = pages_per_huge_page(h);
-       struct address_space *mapping = vma->vm_file->f_mapping;
+       struct address_space *mapping = src_vma->vm_file->f_mapping;
        struct mmu_notifier_range range;
        int ret = 0;

        if (cow) {
-               mmu_notifier_range_init(&range, MMU_NOTIFY_CLEAR, 0, vma, src,
-                                       vma->vm_start,
-                                       vma->vm_end);
+               mmu_notifier_range_init(&range, MMU_NOTIFY_CLEAR, 0, src_vma, src,
+                                       src_vma->vm_start,
+                                       src_vma->vm_end);
                mmu_notifier_invalidate_range_start(&range);
        } else {
                /*
@@ -4727,12 +4728,12 @@ int copy_hugetlb_page_range(struct mm_struct *dst, struct mm_struct *src,
                i_mmap_lock_read(mapping);
        }

-       for (addr = vma->vm_start; addr < vma->vm_end; addr += sz) {
+       for (src_addr = src_vma->vm_start; src_addr < src_vma->vm_end; src_addr += sz, dst_addr += sz) {
                spinlock_t *src_ptl, *dst_ptl;
-               src_pte = huge_pte_offset(src, addr, sz);
+               src_pte = huge_pte_offset(src, src_addr, sz);
                if (!src_pte)
                        continue;
-               dst_pte = huge_pte_alloc(dst, vma, addr, sz);
+               dst_pte = huge_pte_alloc(dst, dst_vma, dst_addr, sz);
                if (!dst_pte) {
                        ret = -ENOMEM;
                        break;
@@ -4753,7 +4754,8 @@ int copy_hugetlb_page_range(struct mm_struct *dst, struct mm_struct *src,
```

```
                dst_ptl = huge_pte_lock(h, dst, dst_pte);
                src_ptl = huge_pte_lockptr(h, src, src_pte);
-               spin_lock_nested(src_ptl, SINGLE_DEPTH_NESTING);
+               if (src_ptl != dst_ptl)
+                       spin_lock_nested(src_ptl, SINGLE_DEPTH_NESTING);
                entry = huge_ptep_get(src_pte);
                dst_entry = huge_ptep_get(dst_pte);
 again:
@@ -4776,10 +4778,10 @@ int copy_hugetlb_page_range(struct mm_struct *dst, struct mm_struct *src,
                                swp_entry = make_readable_migration_entry(
                                                        swp_offset(swp_entry));
                                entry = swp_entry_to_pte(swp_entry);
-                               set_huge_swap_pte_at(src, addr, src_pte,
+                               set_huge_swap_pte_at(src, src_addr, src_pte,
                                                     entry, sz);
                        }
-                       set_huge_swap_pte_at(dst, addr, dst_pte, entry, sz);
+                       set_huge_swap_pte_at(dst, dst_addr, dst_pte, entry, sz);
                } else {
                        entry = huge_ptep_get(src_pte);
                        ptepage = pte_page(entry);
@@ -4794,20 +4796,20 @@ int copy_hugetlb_page_range(struct mm_struct *dst, struct mm_struct *src,
                         * need to be without the pgtable locks since we could
                         * sleep during the process.
                         */
-                       if (unlikely(page_needs_cow_for_dma(vma, ptepage))) {
+                       if (unlikely(page_needs_cow_for_dma(src_vma, ptepage))) {
                                pte_t src_pte_old = entry;
                                struct page *new;

                                spin_unlock(src_ptl);
                                spin_unlock(dst_ptl);
                                /* Do not use reserve as it's private owned */
-                               new = alloc_huge_page(vma, addr, 1);
+                               new = alloc_huge_page(src_vma, src_addr, 1);
                                if (IS_ERR(new)) {
                                        put_page(ptepage);
                                        ret = PTR_ERR(new);
                                        break;
                                }
-                               copy_user_huge_page(new, ptepage, addr, vma,
+                               copy_user_huge_page(new, ptepage, src_addr, src_vma,
                                                    npages);
                                put_page(ptepage);

@@ -4817,13 +4819,13 @@ int copy_hugetlb_page_range(struct mm_struct *dst, struct mm_struct *src,
                                spin_lock_nested(src_ptl, SINGLE_DEPTH_NESTING);
                                entry = huge_ptep_get(src_pte);
                                if (!pte_same(src_pte_old, entry)) {
-                                       restore_reserve_on_error(h, vma, addr,
+                                       restore_reserve_on_error(h, src_vma, src_addr,
                                                                 new);
                                        put_page(new);
                                        /* dst_entry won't change as in child */
                                        goto again;
                                }
-                               hugetlb_install_page(vma, dst_pte, addr, new);
+                               hugetlb_install_page(src_vma, dst_pte, dst_addr, new);
                                spin_unlock(src_ptl);
                                spin_unlock(dst_ptl);
                                continue;
@@ -4837,15 +4839,16 @@ int copy_hugetlb_page_range(struct mm_struct *dst, struct mm_struct *src,
                                 *
                                 * See Documentation/vm/mmu_notifier.rst
                                 */
-                               huge_ptep_set_wrprotect(src, addr, src_pte);
+                               huge_ptep_set_wrprotect(src, src_addr, src_pte);
                                entry = huge_pte_wrprotect(entry);
                        }

                        page_dup_rmap(ptepage, true);
-                       set_huge_pte_at(dst, addr, dst_pte, entry);
+                       set_huge_pte_at(dst, dst_addr, dst_pte, entry);
                        hugetlb_count_add(npages, dst);
                }
-               spin_unlock(src_ptl);
+               if (src_ptl != dst_ptl)
+                       spin_unlock(src_ptl);
                spin_unlock(dst_ptl);
        }

diff --git a/mm/memory.c b/mm/memory.c
```

```
index a89ab8792ba6..cea177e62eb5 100644
--- a/mm/memory.c
+++ b/mm/memory.c
@@ -1256,7 +1256,7 @@ copy_page_range(struct vm_area_struct *dst_vma, struct vm_area_struct *src_vma)
                return 0;

        if (is_vm_hugetlb_page(src_vma))
-               return copy_hugetlb_page_range(dst_mm, src_mm, src_vma);
+               return copy_hugetlb_page_range(dst_mm, src_mm, dst_vma, src_vma);

        if (unlikely(src_vma->vm_flags & VM_PFNMAP)) {
                /*
--
```

References

[1] Per Brinch Hansen. "The evolution of operating systems". In: *Classic Operating Systems*. Springer, 2001, pp. 1–34. ISBN: 0-387-95113-X.

[2] *Desktop Operating System Market Share Worldwide*. URL: https://gs.statcounter.com/os-market-share/desktop/worldwide/#monthly-202107-202206-bar (visited on 29/07/2022).

[3] *Mobile & Tablet Operating System Market Share Worldwide*. URL: https://gs.statcounter.com/os-market-share/mobile-tablet/worldwide/#monthly-202107-202206-bar (visited on 29/07/2022).

[4] Redis Ltd. *Redis*. URL: https://redis.io (visited on 06/07/2022).

[5] Microsoft. *Database Snapshots (SQL Server)*. URL: https://docs.microsoft.com/en-us/sql/relational-databases/databases/database-snapshots-sql-server (visited on 28/06/2022).

[6] Jeff Bonwick. "ZFS". In: *Proceedings of the 21th Large Installation System Administration Conference, LISA 2007, Dallas, Texas, USA, November 11-16, 2007*. Ed. by Paul Anderson. USENIX, 2007. URL: http://www.usenix.org/events/lisa07/htgr_files/bonwick_htgr.pdf.

[7] *btrfs Wiki*. URL: https://btrfs.wiki.kernel.org/index.php/Main_Page (visited on 28/06/2022).

[8] Alfons Kemper and Thomas Neumann. "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots". In: *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*. Ed. by Serge Abiteboul *et al.* IEEE Computer Society, 2011, pp. 195–206. DOI: 10.1109/ICDE.2011.5767867. URL: https://doi.org/10.1109/ICDE.2011.5767867.

[9] Jung-Sang Ahn *et al.* "Jungle: Towards Dynamically Adjustable Key-Value Store by Combining LSM-Tree and Copy-On-Write B+-Tree". In: *11th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2019, Renton, WA, USA, July 8-9, 2019*. Ed. by Daniel Peek and Gala Yadgar. USENIX Association, 2019. URL: https://www.usenix.org/conference/hotstorage19/presentation/ahn.

[10] Ankur Sharma, Felix Martin Schuhknecht and Jens Dittrich. "Accelerating Analytical Processing in MVCC using Fine-Granular High-Frequency Virtual Snapshotting". In: *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. Ed. by Gautam Das, Christopher M.

Jermaine and Philip A. Bernstein. ACM, 2018, pp. 245–258. DOI: 10.1145/3183713.3196904. URL: https://doi.org/10.1145/3183713.3196904.

[11] Jonathan M. Smith and Gerald Q. Maguire Jr. "Effects of Copy-on-Write Memory Management on the Response Time of UNIX Fork Operations". In: *Comput. Syst.* 1.3 (1988), pp. 255–278. URL: http://www.usenix.org/publications/compsystems/1988/sum%5C_smith.pdf.

[12] Richard F. Rashid and George G. Robertson. "Accent: A Communication Oriented Network Operating System Kernel". In: *Proceedings of the Eighth Symposium on Operating System Principles, SOSP 1981, Asilomar Conference Grounds, Pacific Grove, California, USA, December 14-16, 1981*. Ed. by John Howard and David P. Reed. ACM, 1981, pp. 64–75. DOI: 10.1145/800216.806593. URL: https://doi.org/10.1145/800216.806593.

[13] *Memory Management — Memory Protection*. URL: https://docs.microsoft.com/en-us/windows/win32/memory/memory-protection (visited on 25/07/2022).

[14] Akihiko Tozawa *et al.* "Copy-on-write in the PHP language". In: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*. Ed. by Zhong Shao and Benjamin C. Pierce. ACM, 2009, pp. 200–212. DOI: 10.1145/1480881.1480908. URL: https://doi.org/10.1145/1480881.1480908.

[15] Ingo Korb, Helena Kotthaus and Peter Marwedel. "mmapcopy: efficient memory footprint reduction using application knowledge". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*. Ed. by Sascha Ossowski. ACM, 2016, pp. 1832–1837. DOI: 10.1145/2851613.2851736. URL: https://doi.org/10.1145/2851613.2851736.

[16] Redis Ltd. *Redis persistence*. URL: https://redis.io/docs/manual/persistence (visited on 06/07/2022).

[17] M. D. McIlroy, E. N. Pinson and B. A. Tague. "UNIX Time-Sharing System: Foreword". In: *Bell System Technical Journal* 57.6 (1978), p. 1902. DOI: https://doi.org/10.1002/j.1538-7305.1978.tb02135.x.

[18] Abhishek Bhattacharjee and Daniel Lustig. *Architectural and Operating System Support for Virtual Memory*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2017. DOI: 10.2200/S00795ED1V01Y201708CAC042. URL: https://doi.org/10.2200/S00795ED1V01Y201708CAC042.

[19] Peter J. Denning. "Virtual Memory". In: *ACM Comput. Surv.* 2.3 (1970), pp. 153–189. DOI: 10.1145/356571.356573. URL: https://doi.org/10.1145/356571.356573.

[20] Andrew S. Tanenbaum. *Modern operating systems, 3rd Edition*. Pearson Prentice-Hall, 2009. ISBN: 0138134596. URL: https://www.worldcat.org/oclc/254320777.

[21] Gorman Mel. *Understanding the Linux virtual memory manager*. 2007. URL: https://www.kernel.org/doc/gorman/pdf/understand.pdf.

[22] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A-Z*. 2021. URL: https://www.intel.com/content/dam/develop/public/us/en/documents/325383-sdm-vol-2abcd.pdf (visited on 23/06/2022).

[23] Advanced Micro Devices Inc. *Preliminary Processor Programming Reference (PPR) for AMD Family 17h Model 31h, Revision B0 Processors*. 2019. URL: https://developer.amd.com/wp-content/resources/55803_0.54-PUB.pdf (visited on 08/07/2022).

[24] Abraham Silberschatz, James L. Peterson and Peter Baer Galvin. *Operating System Concepts, 3rd Edition*. Addison-Wesley, 1991. ISBN: 978-0-201-51379-0.

[25] Nihar R. Mahapatra and Balakrishna V. Venkatrao. "The processor-memory bottleneck: problems and solutions". In: *XRDS* 5.3es (1999), 2–es. DOI: 10.1145/357783.331677. URL: https://doi.org/10.1145/357783.331677.

[26] John F Couleur and Edward L Glaser. *Shared-access data processing system*. US Patent 3,412,382. Nov. 1968.

[27] Richard P. Case and Andris Padegs. "Architecture of the IBM System/370". In: *Commun. ACM* 21.1 (1978), pp. 73–96. DOI: 10.1145/359327.359337. URL: https://doi.org/10.1145/359327.359337.

[28] John F. Couleur. "The core of the Black Canyon Computer Corporation". In: *IEEE Ann. Hist. Comput.* 17.4 (1995), pp. 56–60. DOI: 10.1109/85.477436. URL: https://doi.org/10.1109/85.477436.

[29] David A. Patterson and John L. Hennessy. *Computer Organization and Design - The Hardware / Software Interface (Revised 4th Edition)*. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, 2012. ISBN: 978-0-12-374750-1. URL: http://www.elsevierdirect.com/product.jsp?isbn=9780123747501.

[30] Raghu Bharadwaj. *Mastering Linux Kernel Development: A kernel developer's reference manual*. Packt Publishing Ltd, 2017.

[31] Madhusudhan Talluri, Mark D. Hill and Yousef Y. A. Khalidi. "A New Page Table for 64-bit Address Spaces". In: *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*. Ed. by Michael B. Jones. ACM, 1995, pp. 184–200. DOI: 10.1145/224056.224071. URL: https://doi.org/10.1145/224056.224071.

[32] Rik Van Riel and Peter W. Morreale. *Documentation for /proc/sys/vm/**. 2008. URL: https://www.kernel.org/doc/Documentation/sysctl/vm.txt (visited on 24/06/2022).

[33] Peter J. Denning. "Working Sets Past and Present". In: *IEEE Trans. Software Eng.* 6.1 (1980), pp. 64–84. DOI: 10.1109/TSE.1980.230464. URL: https://doi.org/10.1109/TSE.1980.230464.

[34] The kernel development community. *Virtual Memory Primer*. URL: https://www.kernel.org/doc/html/latest/admin-guide/mm/concepts.html#virtual-memory-primer (visited on 24/06/2022).

[35] Y Soumya and T Ragunathan. "Lazy Expression Evaluation with Demand Paging In Virtual Memory Management". In: *International Journal of Engineering and Advanced Technology (IJEAT)* 2.1 (2012), pp. 1–3. URL: https://www.ijeat.org/portfolio-item/a0690092112/.

[36] Colin Percival. "Cache Missing for Fun and Profit". In: *In Proc. of BSDCan 2005*. 2005.

[37] Vasundhara Rathod, Monali Chim and Pramila Chawan. "A Survey Of Page Replacement Algorithms In Linux". In: *International Journal of Engineering Research and Applications (IJERA)* 3 (June 2013), pp. 1397–1401.

[38] Gorman Mel. *Re: CLOCK-Pro algorithm*. Jan. 2011. URL: https://marc.info/?l=linux-mm&m=129431080028837 (visited on 27/06/2022).

[39] Theodore Johnson and Dennis E. Shasha. "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm". In: *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*. Ed. by Jorge B. Bocca, Matthias Jarke and Carlo Zaniolo. Morgan Kaufmann, 1994, pp. 439–450. URL: http://www.vldb.org/conf/1994/P439.PDF.

[40] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. John Wiley & Sons, 2010. ISBN: 978-0-470-34343-2.

[41] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel - from I/O ports to process management*. O'Reilly, 2005. ISBN: 978-0-596-00565-8. URL: http://www.oreilly.de/catalog/understandlk/index.html.

[42] Jonathan Corbet. */dev/ksm: dynamic memory sharing*. URL: https://lwn.net/Articles/306704 (visited on 05/07/2022).

[43] Jonathan Corbet. *KSM tries again*. URL: https://lwn.net/Articles/330589 (visited on 05/07/2022).

[44] Sushovon Sinha. *Physical and Virtual Memory in Windows 10*. URL: https://answers.microsoft.com/en-us/windows/forum/all/physical-and-virtual-memory-in-windows-10/e36fb5bc-9ac8-49af-951c-e7d39b979938 (visited on 29/06/2022).

[45] *Linux Kernel Source Code*. Version 5.18. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git.

[46] John Madieu. *Linux Device Drivers Development: Develop customized drivers for embedded Linux*. Packt Publishing Ltd, 2017.

[47] *userfaultfd(2) Linux Programmer's Manual*. 5.18.

[48] Jonathan Corbet. *The next steps for userfaultfd()*. URL: https://lwn.net/Articles/718198 (visited on 04/07/2022).

[49] *madvise(2) Linux Programmer's Manual*. 5.18.

[50] Thomas H. Cormen *et al. Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN: 978-0-262-03384-8. URL: http://mitpress.mit.edu/books/introduction-algorithms.

[51] *proc(5) Linux Programmer's Manual*. 5.18.

[52] *vdso(7) Linux Programmer's Manual*. 5.18.

[53] Jonathan Corbet. *On vsyscalls and the vDSO*. URL: https://lwn.net/Articles/446528 (visited on 05/07/2022).

[54] *pmap(1) User Commands*. 5.18.

[55] Marcin Juszkiewicz. *Linux kernel system calls for all architectures*. URL: https://marcin.juszkiewicz.com.pl/download/tables/syscalls.html (visited on 06/07/2022).

[56] *fork(2) Linux Programmer's Manual*. 5.18.

[57] *Fork and Exec*. URL: http://www-h.eng.cam.ac.uk/help/tpl/unix/fork.html (visited on 06/07/2022).

[58] Robert Love. *Linux Kernel Development*. 3rd. Addison-Wesley Professional, 2010. ISBN: 0672329468.

[59] *mmap(2) Linux Programmer's Manual*. 5.18.

[60] Intel Corporation. *White paper: 5-Level Paging and 5-Level EPT*. Tech. rep. 335252-002. 2017.

[61] *The R Project for Statistical Computing*. URL: https://www.r-project.org/ (visited on 19/07/2022).

[62] Helena Kotthaus *et al.* "Runtime and memory consumption analyses for machine learning R programs". In: *Journal of Statistical Computation and Simulation* 85 (Jan. 2015). DOI: 10.1080/00949655.2014.925192.

[63] *AnKer Kernel Modifications*. URL: https://github.com/BigDataAnalyticsGroup/anker (visited on 24/07/2022).

[64] *Transparent Hugepage Support*. URL: https://www.kernel.org/doc/html/latest/admin-guide/mm/transhuge.html (visited on 24/07/2022).

[65] *memfd_create(2) Linux Programmer's Manual*. 5.18.

[66] *malloc - Glibc source code (glibc-2.36)*. URL: https://elixir.bootlin.com/glibc/glibc-2.36/source/malloc (visited on 31/07/2022).

[67] *jemalloc — a general purpose malloc(3) implementation*. URL: https://github.com/jemalloc/jemalloc (visited on 31/07/2022).