eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Bachelor's Degree in Informatics Engineering
Computer Engineering

Bachelor's Thesis

Development of file access policies for data-intensive applications in Linux

Author
Mikel Iceta Tena

2020

eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Bachelor's Degree in Informatics Engineering
Computer Engineering

Bachelor's Thesis

Development of file access policies for data-intensive applications in Linux

Author
Mikel Iceta Tena

Directors
Iñaki Morlan Santa Catalina, Jose A. Pascual Saiz

Abstract

Our world has been going through a digitization process for some decades now, with the transformation curve being sharper in the last few years. The rise of this new digital society brings with it a huge amount of information, and so it is paramount to create new architectures and hardware devices capable of storing and analyzing such information in a reasonable time. The advancements in some fields such as persistent storage is notable, while in other fields such as RAM some extra work must be done to counter the lack of physical improvement. For example, NVMe discs offer high transfer speeds and low latency in data access while offering high capacities and moderate price points. The RAM, on the other hand, has advanced less, being the transfer rates the only improved factor, letting both memory capacity and prices almost intact. This results in data intensive systems with insufficient RAM that continuously have to access disk in order to execute their programs, bringing a significant degradation of the overall performance. The objective of this project is to analyze means of operating with files other than the classic ones, analyzing the data access patterns of a given application and developing a specific disk access policy for it to function with less hassle. Specifically, we will focus on converting one existing program to a new memory-interacting paradigm and developing new mapped-file page replacement policies at kernel level.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

From the smallest IoT sensors to the complex systems found in data centers, an increasing number of devices generate and process huge amounts of data continuously. These processing systems must load big amounts of data into their memory in order to process it. As time goes by the amount of data generate by those devices scales up and as a consequence the computers used to treat those data struggle to keep up with the memory requirements [Makrani et al., 2018].

This scenario where the number of users and devices – and hence the amount of data generated – keeps growing and growing becomes unfeasible if advancements of similar magnitude are not done in the underlying technologies. Many factors limit the capacity of a system: power consumption, cooling, bandwidth of the network in use, physical space and price.

All of them are to be considered at the time of setting up such a system, but there is one factor that often goes by unnoticed: main memory ***usage***. Main memory is the place where executing programs store code and data related to the application. Raw computing power (new CPU and GPUs with higher clocks and more cores), persistent storage (NV-DIMM's[1], NVMe SSD's[2]) and network connections (400/800 GbE[3], Infiniband HDR[4] , Slingshot[5]) have improved in a consistent way set against the growth of the needs of the society over time, even in orders of magnitude from a generation to another.

On the other hand, the only advancements seen in the main memory are bandwidth improvements, higher clocks and lower consumption. These improvements are, to an extent, sufficient to keep up with the needs of the average consumer, but when taking an overall look at the "data intensive world", it is just not enough. This - alongside with its high price tag - makes main memory a very scarce and valuable resource in most currently functioning

[1] https://www.zdnet.com/article/first-optane-performance-tests-show-benefits-and-limits-of-intels-nvdimms/
[2] https://es.wikipedia.org/wiki/NVM_Express
[3] https://en.wikipedia.org/wiki/Terabit_Ethernet
[4] https://www.infinibandta.org/infiniband-roadmap/
[5] https://www.cray.com/products/computing/slingshot

systems. It is considered scarce as e.g. a low-end system may have up to 10TB of persistent storage while only having 32GB of RAM.

Modern data-intensive applications to need way more main memory than any mainstream/-commercial system has to offer. This often ends up in situations where the systems need to be constantly bringing data in and out from and to persistent storage, losing a substantial amount of processing time doing so. This does not only physically degrade the memory system (as a big amount of information is being written and read from the storage devices), but it also does negatively affect the performance of the applications. Up until now, almost every major problem encountered in terms of lack of computing resources has been solved by adding more resources. This leads to the misconception that the only way to solve the lack of memory is adding more memory.

In fact, there are several ways of alleviating the impact of the lack of memory. The most obvious one is, indeed, adding more memory to the system. However, this is unfeasible as the price of volatile memory is high and the number of modules that can be physically installed in a computer is limited. The second way is augmenting the memory hierarchy by adding an intermediate memory layer between the main memory and the persistent storage, being it faster than the "big" persistent storage but slower than the RAM. This has already been tested in works such as the one of [Essen et al., 2012], and although it is not as fast as just adding RAM, it significantly increases the storage subsystem performance.

Another option – the core of this work – is to modify the way the OS manages and allocates the memory by changing/optimizing the page allocation policies and algorithms and changing the applications to interact with files in a different way. Linux can only improve access latency for sequential reads. For any other access pattern it simply does nothing. In this work we will try to answer the following questions:

- Can new memory management policies be implemented in order to satisfy the needs of a particular application?
- Can an application be modified in order to process bigger files without a notable performance degradation?

Results obtained throughout this project show that the answer to both questions is ***«yes»***.

CHAPTER 2

Project Goals Document

The goal of the project is to inquire into alternative ways of interacting with files in Linux, as well as modifying the kernel's memory subsystem for it to manage new access patterns.

As any other monolithic kernel, Linux has an integrated memory management mechanism. This allows applications to access the memory in an easy and normalized way. This system brings the data in to main memory from disk as needed. When there is not enough room to store everything in the main memory, data must be evicted from it and brought back later if needed again. This may lead to situations where the memory system does not perform good. The classic file reading logic traverses the whole I/O stack an can be considered too much of a burden in some cases.

As a workaround, files can be mapped into the process address space, avoiding traversing the whole stack. In Linux this can be done via the ***mmap*** interface. This solution can improve timings and even allow the execution of otherwise unusable programs. Data intensive applications tend to be hard on the main memory, and vary in the data accessing patterns. Linux is a general-purpose kernel and, as such, it only includes policies to deal with sequential accesses. This results in a sub-optimal performance as the kernel keeps failing on guessing which information it has to bring in or out from/to disk for non-trivial access patterns.

Several test bench applications will be designed and implemented to test various hypotheses in the area of memory performance. An existing data-intensive program will be executed and measured in order to find a case where it can not be executed on a given machine. Its data-accessing logic will then be changed so it can be executed and the time will be compared with that of a machine that could originally execute it.

After an application has been modified to test the capabilities of mmap, some modifications to the Linux kernel itself will be presented. These modifications will aim to adapt the memory subsystem to the needs of certain applications. Specifically, a new page-fault read-ahead logic/policy will be proposed and implemented. This will reduce the time spent on I/O operations (and thus improve the performance).

The main tasks of the project can be summarized into the next list:

1. To perform a deep study of Linux memory management subsystem.
2. To dive into alternative ways of accessing large disk files.
3. To modify an existent data-intensive application's way of accessing disk files.
4. To create a synthetic data-intensive application and tailoring a new page-faulting policy optimized for it.

Even if this project revolves around specific programs, the approach can be applied to virtually any kind of application. Determining access patterns in common/popular data-intensive applications and tailoring an appropriate algorithm can be taken as long-term or an optional goal for this project. The main activities of the project will be carried in the next way:

First, the Linux kernel has to be understood. After the preliminary study is finished, the Linux code related to memory management has to be explored looking for the point where the page replacing policies are implemented for the mapped files. To be able to compare mmap to the I/O stack a test suite will be created, consisting of various shell scripts combined with C programs to read and write with synthetic, known patterns. This will provide a deeper understanding of Linux memory concepts and will allow finding the files where policies are implemented. Further testing will be performed in pursuance of determining if mmap is a better alternative in every scenario. Once the implementation point is found, a synthetic data intensive program will be created and the memory subsystem will be modified to accommodate it to the application.

CHAPTER 3

Preliminaries

The Linux kernel is very intricate, but it can be understood after worming in into its source code. The whole project revolves around the behavior of the memory subsystem found in the Linux kernel for the x86-64 architecture in data intensive environments. As a consequence, all of the following research and most of the explained concepts will be centered on that specific version.

3.1 Introduction to kernels

The Kernel is the program (or set of programs) that governs the computer it runs on. It is responsible for the functioning of the system and acts as an intermediate layer between the operating system (and the user applications) and the machine's hardware, as depicted in Figure 3.1. A kernel is comprised of several processes running at a high privilege level that maintain the system running.

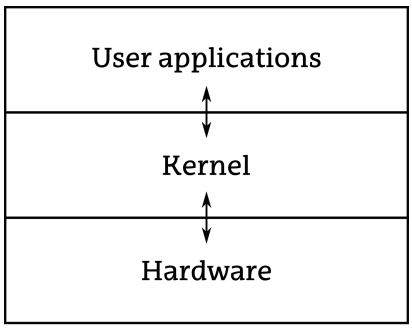


Figure 3.1: Logical location of the kernel in the system.

The kernel keeps track of every process, including the ones that shape it, in a complex structure called the ***Process Control Block*** (PCB). This structure includes the exact execution state for each process: processor register values, scheduling information, execution privileges, etc.

In multi-programming environments multiple tasks can be running at the same time, so the kernel stores a list (usually a doubly-linked list) containing all existing PCBs. The PCB list is stored inside protected structures inside the kernel memory in order to hide sensible data from user space.

All modern kernels support multi-programming which, as stated before, allows multiple processes to be run concurrently on the same processor core. This concurrency is but a mere illusion[1] created by the continuous context switching performed by the kernel. When a task is evicted from execution state for any reason different than it coming to its end, the kernel saves its information (context) in the corresponding PCB and then restores that of another ready-to-run task. This is called a context switch (or preemption) and involves a series of queues to store every process that is in each state at any given time. Each context switch is said to come at a cost: latency, as saving/restoring the information is not free (computationally speaking).

To assure security, CPUs include isolate, hard-coded privilege levels called ***rings***. The highest is the less privileged one and the lowest the most privileged. Rings differ in which CPU instructions can be executed by the calling program. If application issues an non-permitted instruction, a ***General Protection Exception*** will be raised by the CPU to stop its execution. In order to satisfy requests from non-privileged code that needs low-level access to protected resources, applications must ***ask*** the kernel to do it using an interface (i.e. system calls). When a program performs a system call, the kernel will receive the request and honor it if possible.

As it can be seen in Figure 3.2, in the x86-64 architecture user space applications may only be executed in ring 3, while the kernel is executed in ring 0. Even if rings 1-2 were originally created in order to be used by device drivers, they are widely unused, and modern operating systems such as Linux and Windows only work in two levels: supervisor/kernel (ring 0) and user modes(ring 3). An additional ring (-1), is used in virtualized environments. The VM hypervisor[2] executes at level -1 [3], and the hosted kernels at level 0.

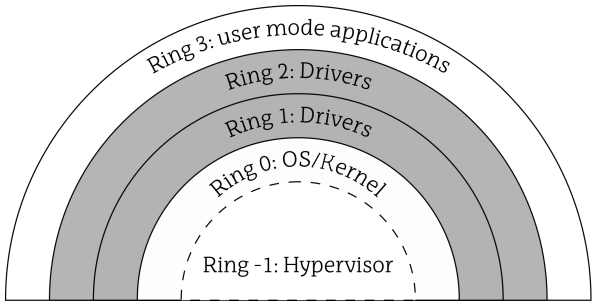


Figure 3.2: "Rings" found in the x86-64 architecture. Rings in gray are not used.

[1] Considering a single core processor.
[2] The hypervisor is an abstraction layer present in virtualized environments. It is located between the hosted kernels and the host kernel in order to isolate them one from another and from the hardware
[3] The ring -1 does not actually exist. It is a virtual layer created by the hypervisor.

3.2 Kernel types

Switches between privileged (kernel) and non-privileged (user) execution modes and vice-versa yield a time overhead to the execution. Various philosophies exist in regards of which parts of the system should be part of the kernel and thus be executed in kernel mode, as it can be seen in Figure 3.3. The main approaches are: **monolithic, microkernel and hybrid** kernel designs.

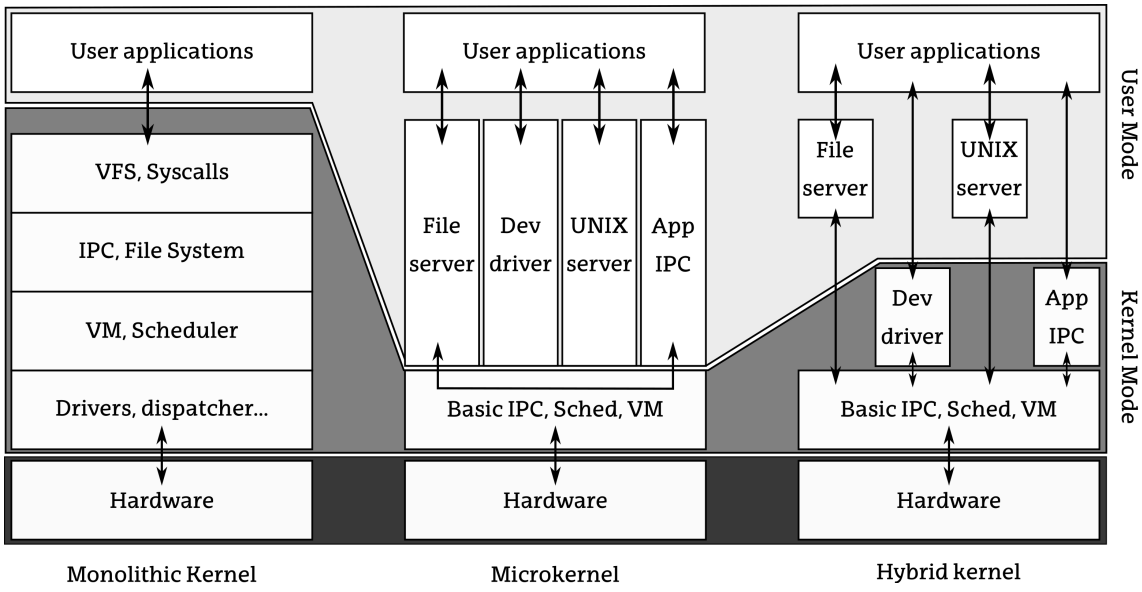


Figure 3.3: Differences between the three main kernel design patterns.

In a monolithic kernel based OS, all of the operating system runs in kernel mode, including device drivers, inter-process communication mechanisms, virtual memory, etc. User applications access resources using the kernel ***System Call Interface***. A call to this interface results in an privilege level switch, which triggers the functions that satisfies the call via kernel code and then returns to user privilege level.

As a major part of the whole system resides in the kernel in monolithic approaches, the number of mode transitions between kernel and user mode is reduced in comparison to other designs. As a downhill, this block-like design proves to be the most difficult to maintain, as the modularity level is very low and the code tends to get tangled. When a part of the kernel suffers a failure, the whole system may crash. Examples of monolithic kernels include BSD, Linux and Unix kernel families.

The opposite philosophy is implemented in **microkernels**, where only basic scheduling and IPC primitives are included and run in privileged mode. Every other application, such as device drivers, file managing and complex IPC systems (such as ***sockets***) is run in user mode. These tasks can communicate with the kernel and other applications in a client/server manner. On one hand, using this scheme may result in a slower execution, as these separate user space applications incur in more execution mode transitions when communicating than in

other kernel types, creating as a consequence, overheads. On the other hand, this loose coupling makes development, securing and maintenance easier, as modules can be updated and substituted without the need of modifying and re-compiling the whole system. Examples of microkernels are the L3 and L4 families.

The third option combines the former two. **The hybrid design** incorporates as kernel code some of the code that is considered unnecessary in microkernel approaches (such as device drivers or full IPC mechanisms). This approach aims to gather the benefits of both microkernel and monolithic designs, finding a balance. Examples of hybrid kernels are Windows NT, Apple XNU.

3.3 The Linux Kernel

The Linux kernel is a free and open-source, monolithic, Unix-like operating system kernel [4]. Since its launch in 1991, Linux has evolved by the hands of many different developers. This makes its source code enormous (27+ million lines of code [5]) and difficult to read and understand. Subsystems are usually scattered between different source files written and/or maintained by different programmers, each with their own thinking and coding style. Linux is a monolithic kernel, which makes tinkering through it an even more difficult task. Even when vastly simplified, as can be seen in Figure 3.4, the structure may seem intricate.

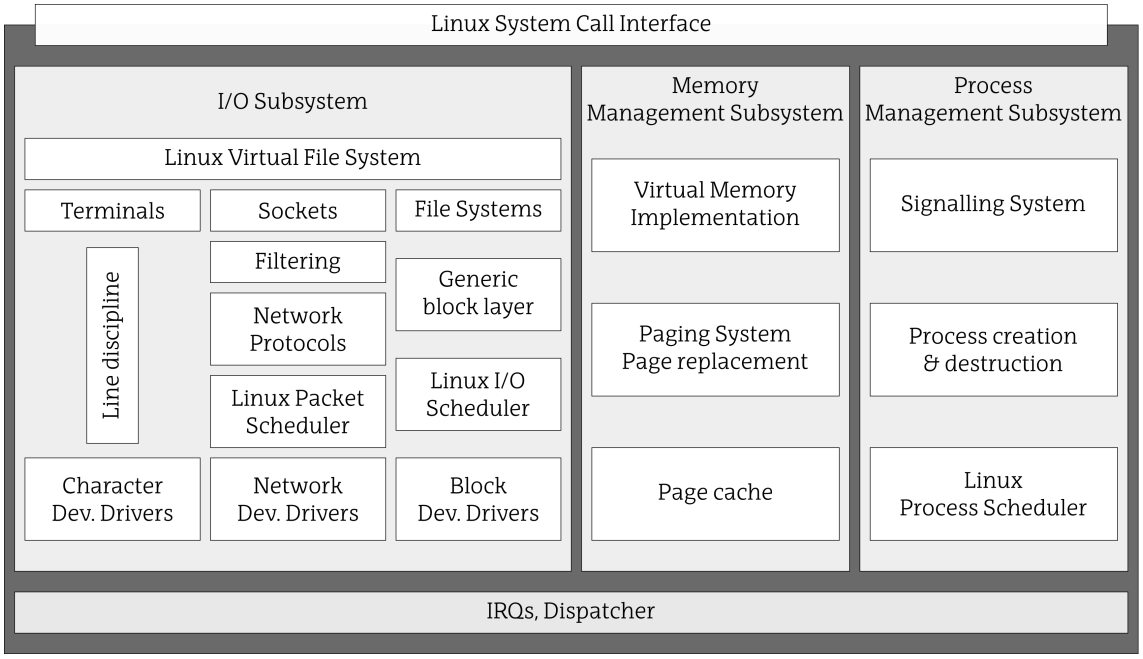


Figure 3.4: Basic structure of the Linux Kernel.

To correctly and securely run multiple processes at the same time, Linux implements a Virtual Memory system which presents a different working space to each process. Every pro-

[4] https://en.wikipedia.org/wiki/Linux_kernel
[5] https://www.linux.com/news/linux-in-2020-27-8-million-lines-of-code-in-the-kernel-1-3-million-in-systemd/

cess has a different and unique (virtual) memory address space and can be partially loaded into different, non-contiguous physical memory locations, effectively and efficiently allowing multiple processes to be run at the same time without clogging the machine and isolating one from another.

3.4 Processes

The term process is used to refer to a program during any of its execution states, from the moment it is created to its destruction. A program is said to be executing when it is read from disk into main memory (RAM) and instantiated. Various processes may exist as different instances of the same program. At any given time, a task may be in one of the states depicted in Figure 3.5.

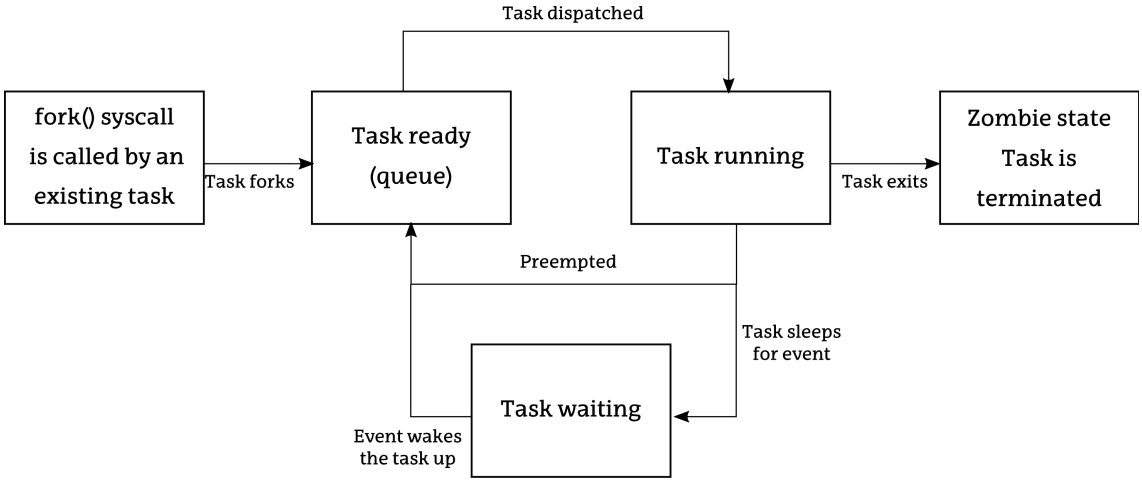


Figure 3.5: The simplified life cycle of a process.

In UNIX-like systems (such as Linux) the only way to spawn a new process is by duplicating an existing one. This is done by means of the ***fork*** system call[6], which is internally implemented by either the fork() function or a variant of the clone() function (it depends on the machine architecture).

In any case, when an existing process calls a ***fork***, the call tree will end up with a call to the ***_do_fork()*** function. This will duplicate the original operating structures of the parent process (e.g. the PCB) and its address space layout. After the information of the parent process has been completely copied to the child process, minor changes are done to the descriptor of the children process in order to differentiate it from its parent and is then put into the execution queue (if no image substitution is needed), waiting to be executed.

The process dispatcher will eventually put it to run in the CPU. This execution will carry on until one of the following happens:

[6] https://www.tutorialspoint.com/how-to-create-a-process-in-linux

- The execution time slice is exceeded and preemption[7] is provoked.
- A higher priority process enters the ready state and provokes preemption.
- The task blocks on an event, such as an I/O operation, and gets preempted.
- The process ends, calls the ***do_exit()*** function and terminates.

The highest time penalty of creating a new process is the one needed to copy the aforementioned structures of the parent process and modifying the unique fields. Pretty much everything will remain shared in memory among the parent and the child processes with a read-only flag. Only when a write operation is performed to a memory position there will be an actual information copy. This is known as copy on write (or CoW) or lazy copy philosophy.

This CoW is quite useful in many cases, as copying the entire address space content while creating new processes would be useless in the case of a process that changes its execution image after being created. For example, when executing a program from the terminal, the interpreter process (sh, zsh, bash...) forks itself and then changes the image being run at the children's side. Physically duplicating all of the information would be useless, as it would be discarded as soon as ***execve()*** is called to load the wanted program into memory.

The functions from the ***exec*** syscall family load the information of the desired program and prepare the process to be run (if its safe to do so). Such functions will clear the address space of the caller in order to accommodate the soon-to-be-born process, and similarly to what happens to the fork call, it will happen in a CoW manner. This means that only a part of the program will be initially loaded into main memory. The actual information transaction relies on the kernel's paging system, which will read the information from the secondary storage (disk) on demand.

As this project revolves around the behavior of processes in the memory scope, it is important to take a look at the kernel structures that help understand the interactions between the process and the main memory. Linux implements its PCB using the ***task_struct*** data type, a circular, doubly-linked list which manages all of the PCBs. This structure includes a ***mm_struct*** type field called **mm**. This field points to the memory descriptor of the task, which contains all sorts of information on how is it structured in the virtual memory system.

The memory descriptor contains multiple representations of the memory structure of the process. Only one of them is shown in the Figure 3.6 for the sake of simplicity, as the described areas are the same.

The **The Stack** is a region reserved for objects with automatic lifetime (see Figure 3.7), which include function calls and the variables generated in them. This region grows towards the MMAP segment. **The Memory Mapping region** contains information about the files that have been mapped into the memory as well as mappings of loaded shared libraries (.so files). The MMAP segment grows towards the Heap. **The Heap** stores the dynamic objects. These

[7] Wikipedia: Preemption is the act of temporarily interrupting a task being carried out by a computer system, without requiring its cooperation, and with the intention of resuming the task at a later time. This is also known as a context switch.

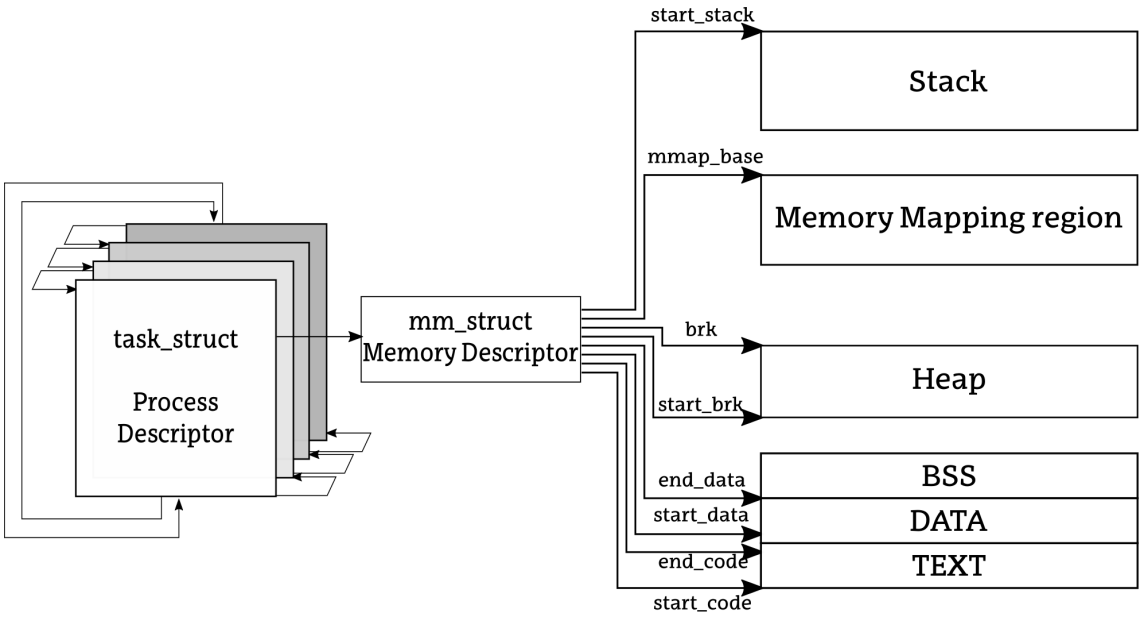


Figure 3.6: The PCB contains a memory descriptor which keeps track of the virtual address space usage.

are allocated using calls to i.e. ***malloc()*** and disappear when a destructor function is called, i.e. ***free()***. The Heap grows towards the MMAP segment. **The BSS region** holds all the non-initialized variables of the process. **The DATA region** contains initialised data such as assigned variables and constants, and **the TEXT region** contains the executable image of the program itself.

3.5 Virtual Memory in Linux

Virtual memory was developed during the decade of 1950. Its main goal is to allow multiple programs to be loaded into memory simultaneously and transparently in multiprogramming systems. Thanks to the Virtual Memory system, a program needs not to be fully loaded in memory in order to function. The parts of a process that are not being used can be evicted from main memory in order to accommodate new parts or even parts from another process. The parts of the process that are loaded into memory are said to be resident.

When non-resident information is needed for the execution of a program, kernel's paging system in conjunction with the ***Memory Management Unit*** translates the virtual address to a physical one in order to bring it from persistent storage. This translation system relies on page tables to perform the correspondence.

3.5.1 Page tables

The page is the smallest allocation unit in terms of memory. The actual memory portion described by a page depends on the underlying architecture, but most of the actual ones (such

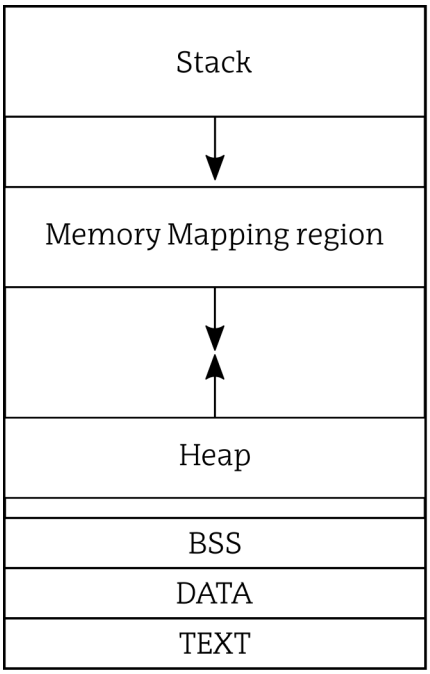


Figure 3.7: The structure of a process in memory.

as x86-64) implement multiple page sizes (4KiB is the most usual, but 2MiB and 1GiB pages can also be allocated). To keep everything uniform, the minimal allocation unit used for the secondary memory is also the page. The main memory is organized in **frames**, which may contain a page or be free.

In order to allocate as many programs as possible the virtual memory system (inside the kernel) keeps track of each process' page table, allowing the program to be not only partially loaded but also using (physically) non-contiguous memory regions, as the translation system allows it. As a consequence, contiguous linear addresses used by a process do not to correspond to physical contiguous addresses.

The page translation system is implemented in practice using a hierarchical set of address tables, as it can be seen in the Figure 3.8. Processes refer to "their" information using virtual addresses, which correspond to a certain physical memory address. In order to translate from virtual to physical and determine if a page is residing in memory, the kernel paging system must (usually) go through all of the levels of the aforementioned hierarchical table set.

Up until now, x86-64 used a 4-level paging system which uses 48 out of the 64 available hardware memory bus bits to address memory, limiting the maximum physical memory to 256TiB,. New advances in the matter present a 5-level translation system, which step up the actual physical addressable limit from 256TiB to 128PiB. In both 4-level and 5-level paging used in the x86-64 architecture, the uppermost table is addressed using the base address stored in the CR3 CPU register as the starting point of the translation.

From that point on, fragments of the linear address are used as offsets to find the exact entry on the current level table which, in turn, contains the base address of the next level table. This

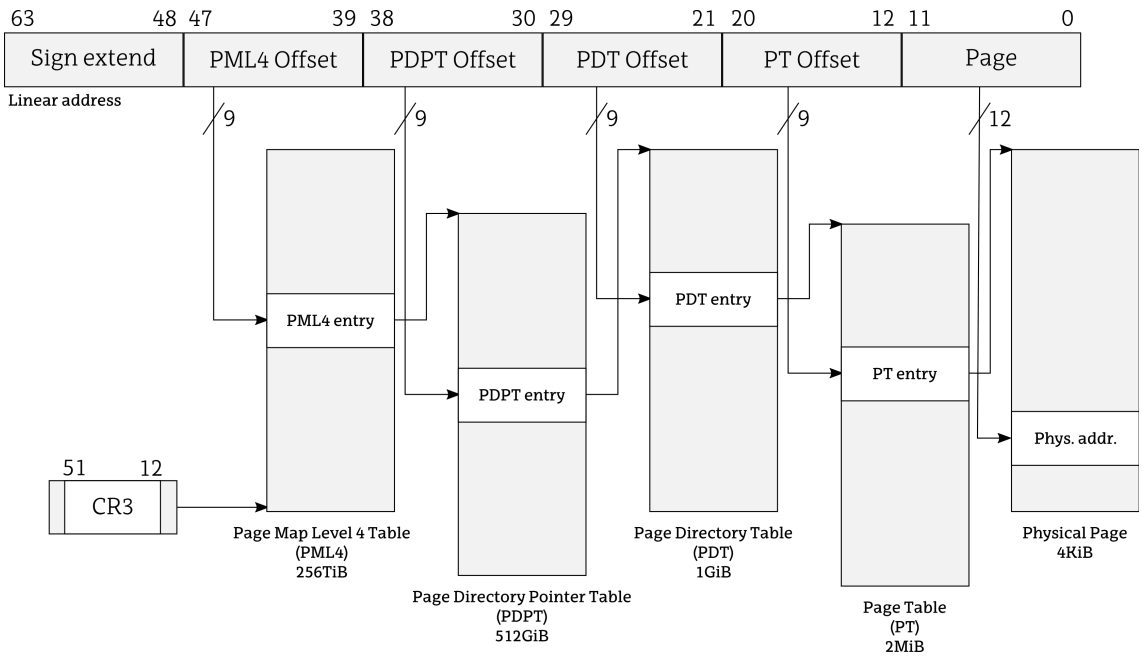


Figure 3.8: 4 level paging system found in x86-64.

is done until the physical page address is resolved and tested. If the page has not been loaded into memory (it is not resident), the paging system will raise a page fault in order to bring it to memory (if possible).

Bringing pages one by one can negatively affect the overall performance, as it will make the faulting process to continuously incur in costly I/O transactions. That is why pages are bought to memory in blocks rather than doing it one by one. This is useful in scenarios where the data is accessed sequentially, but not in the case of random reads or distant memory blocks, as the faults will require I/O operations more often.

3.5.2 Page faults

There are three main page fault types: soft, hard and invalid. When a program faults on a page, three situations may arise: (1) the page is residing in the main memory but is not mentioned in the page table of the process, (2) the page has not yet been loaded into main memory, so access to secondary memory is required and (3) the access is out of permitted bounds.

When a page fault occurs, the best case scenario is the minor fault (1), as it only requires minimum changes to the process descriptor and the Memory Management Unit. This is usually the case of references to shared resources in memory, where other processes have bought that page to memory but the MMU does not know that it has been loaded in the context of that process.

When a major page fault (2) happens the system's page fault handling system brings the page from the secondary storage. The process is then evicted from the running state and put into the waiting queue in order to proceed. The memory allocation code seeks a free page set big

enough to satisfy the petition or replaces victim, non-free pages based on the systems replacement policy. After the placing is resolved, the victim dirty data is written back to the disk (if needed) and the MMU and process descriptor are updated. This concludes the fault and puts the process back into the ready to run queue. The process will be awakened by the CPU scheduler when its time comes and it will proceed with the execution.

The invalid or illegal access case (3) just implies that the process has tried to access a memory location which does not belong to its address space. This is common when trying to access an uninitialized object which points to address 0 from kernel space (null pointer) and provokes a segmentation fault which stops the offending process.

3.5.3 Page fault handling in Linux kernel

The function designated as the default page fault interruption handler in Linux is named ***do_page_fault()***. Each architecture (x86, x86-64, ARM, ARM64, PPC, etc.) has a different version of this function, each designed to fit the hardware of each platform. The exact implementation is available under the ***arch/arch_name/mm/*** directory, in the ***fault.c*** file. In the case of the x86_64 architecture under the version 5.2.2, this function does as follows:

```
do_page_fault(struct pt_regs *regs, unsigned long error_code, unsigned long address)
{
  enum ctx_state prev_state;

  prev_state = exception_enter();
  trace_page_fault_entries(regs, error_code, address);
  __do_page_fault(regs, error_code, address);
  exception_exit(prev_state);
}
```

First, this function saves the current context into a variable using the ***exception_enter()*** function. This includes the processor register values, execution state, etc. mentioned earlier in this work. This is done to keep track of the execution state of the running process at the time of calling the function. Then the function activates page fault tracing (debugging) if enabled (***trace_page_fault_entries()*** and calls ***__do_page_fault()*** which simply carries on the fault routine and restores the execution state using ***exception_exit()***.

The next crucial function is the previously stated ***__do_page_fault()***. It is the page fault routine entry point in Linux. It will unchain the function series that will bring the information from storage into memory. It executes the following code:

```
static noinline void
__do_page_fault(struct pt_regs *regs, unsigned long hw_error_code,
  unsigned long address)
{
 prefetchw(&current->mm->mmap_sem);

```

```
7   if (unlikely(kmmio_fault(regs, address)))
8     return;
9
10  /* Was the fault on kernel-controlled part of the address space? */
11  if (unlikely(fault_in_kernel_space(address)))
12    do_kern_addr_fault(regs, hw_error_code, address);
13  else
14    do_user_addr_fault(regs, hw_error_code, address);
15 }
16 NOKPROBE_SYMBOL(__do_page_fault);
```

This function first locks on ***mmap_sem*** to protect the address space of the process from race conditions, then checks if the fault refers to a memory mapped I/O (MMIO) region. If so, it launches the kmmio handler to take care of it. After that, the function determines if the address belongs to the kernel portion or user portion of the address space. This check is done using the ***unlikely*** macro, which tells the compiler to generate assembly code which will favour the ***else*** branch, improving the accuracy of the CPUs branch predictor, as the majority of the page faults will be issued by user mode code.

The function then launches routines ***do_kern_addr_fault()*** or ***do_user_addr_fault()*** according to the origin. The first function is not relevant in the scope of this project, as DI applications are not run in kernel mode, and kernel space faults performed from user space will end up in ***segment faults***. The second function checks if the fault is valid, and if so, calls ***handle_mm_fault***. If no errors occur during the process, this function will also update the minor or major page fault counters.

As stated before, Linux is a big and complicated kernel. This means that for a certain operation (such as accessing a file), several methods exist. Having seen how frequent page faults and system calls may affect the performance and given the fact that reading a big file will result in lots of system calls, a question arises: Does Linux offer any other way of interacting with files apart from the common read and write system calls?

CHAPTER 4

The mmap inteface

MMAP is a functionality present in modern kernels such as Linux and BSD. Its functions allow processes to map files from secondary storage, devices and even other processes into their virtual address space and access them. This feature allows seamless and direct access to resources, as an access to an array position is directly translated into a memory space access. In the case of devices, MMAP will connect the device memory with the address space, so references to a certain address range will result in accesses to the device itself. If at the time of addressing a position its content is not residing in main memory, the system brings it through a special kind of page faults. This access goes through less layers than the classic reading method, which involves intermediate buffers and using system calls.

4.1 Interfacing mmap from user-space

The two main system calls used to map and unmap files into the calling processes address space are ***mmap()*** and ***munmap()***. Their headers are:

```
1 #include <sys/mman.h>
2 void  *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
3 int   munmap(void *start, size_t length);
```

The first function, ***mmap()***, takes as an input parameter the desired ***start*** address and tries to map ***length*** bytes at an ***offset*** value offset, using the file pointed by the ***fd*** file descriptor as a source. If succeeded, the function will map the file into the processes address space with the given ***prot*** mode and ***flags*** flags and return a pointer to the initial address.

The parameter ***prot*** can either be ***PROT_NONE***, which won't allow access to the pages or a bitwise ***or*** of any of ***PROT_EXEC***, ***PROT_READ*** and ***PROT_WRITE*** macros. As their names imply, they mark the ability of the mapped pages to be executed, read and/or written respectively.

The ***flags*** parameter determines the characteristics of the mapped object. A mapping can be fixed, shared or private. Fixed maps will overwrite existing mappings in case of overlapping and will try to force the specified address to be the beginning of the mapping. Shared maps can be shared among processes and writing operations are equivalent to writing directly to the mapped file. Private maps create a copy of the mapped file which does not reflect changes in the file until flushed. The Linux kernel's MMAP implementation also describes 11 more flags which are not POSIX compliant.

When a file mapping is created using the ***mmap()*** system call, an entry is created in the calling process' PCB mm_struct, assigning part of the virtual address space to the backing file.

Pointers to start and end of memory mapped files are stored in several ways inside the memory descriptor of the process (see Figure 3.6), and the data is allocated in the Memory Mapping Region segment of the process, growing towards the heap segment. As it is represented in the Figure 4.1 (all other vm_area_structs have been omitted for the sake of simplicity), mapping a file using mmap will proyect it as contiguous linear addresses in the process control block.

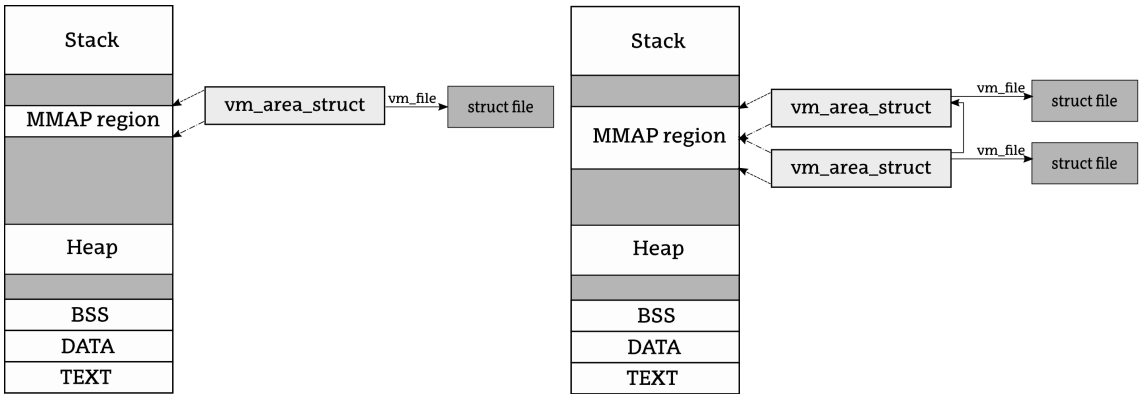


Figure 4.1: Effect of mmap-ing a file in the process memory descriptor.

When the mmap system call is performed successfully, the memory-related structures are updated (ie. a Virtual Memory Area -or VMA- structure which points to the mapping and the file descriptor is created and it is then inserted into the VMA list between the stack and heap VMAs). If no hints are given to the system or ***flags*** does not contain MAP_POPULATE, no information further than the first block is copied to main memory at first. If such flag is present, the virtual-physical address correspondences will be resolved before returning control to the calling code.

This happens because of the "lazy" implementation style of MMAP in Linux, which relies on ***mapped page faults*** to actually perform memory transactions. As stated before, the process may press the system to populate the memory address translation table at the beginning using the MAP_POPULATE flag. The same goes with the actual data copying, which can be "forced" by using ***madvise*** over the address range.

4.2 The page cache and mapped file faults

The previous Linux chapter found in this work mentions a page faulting system used to transfer data from the disk to the main memory. This routine and the code that composes it are in charge of bringing data blocks from storage devices into main memory in the process called ***page faulting***. When taking a look at the statistics shown by processes that employ MMAP as the main mean of dealing with files, it becomes clear that there is an inconsistency in the ***expected faults vs observed faults*** binomial. Preliminary experiments show that there are no more than a few major page faults when dealing with sequential reads even in huge file. Logic would tell that after (access-size * number-of-accesses › block-size) a major fault would occur, as the program would be reading a non-resident block. The results show that only minor faults occur during a sequential read. This happens because the ***"regular"*** page faulting system does not deal with such kind of faults. That is to say, there is a different system that handles page faults generated by mapped files.

The function mainly in charge of bringing pages correspondent to memory-mapped files is filemap_fault(), which resides in /mm/filemap.c. This background-running process makes use of a ***readahead*** mechanism and the ***page cache*** in order to diminish the amount of directly-generated hard faults. In Linux, the mapped faulting mechanisms can be given hints about the memory access behavior by means of calling madvise(), posix_fadvise() and related functions. Calling these functions will result in a ***VM_TYPE*** flag being activated, where ***TYPE*** is the access pattern hint.

As of Linux version 5.2.2, the mapped fault system uses two VM hints implemented as bitmasks: VM_SEQ_READ for sequential read patterns and VM_RND_READ for random or unknown patterns. The system also implements a simple heuristic to detect linear and random read patterns, so even if a hint is given at first it may be internally changed later.

Operating Systems use a portion of the free main memory space to accommodate what is called the ***page cache*** [Duarte, 2019]. This cache contains pages from secondary storage in order to speed-up data accesses. The page cache also has eviction mechanisms in order to free up space when needed, so pages not used in a long time are marked ***cold*** and pages used often are marked ***warm***. When the system is under memory pressure, the cold pages are more likely to be evicted from main memory than the ones that are warm.

The method used for filling in the page cache is a readahead system [Services, 2017]. This includes what is called a ***readahead window***, which is a region next to the faulted page which will be asynchronously loaded into memory. When the accesses reach the end of the window, it is moved forward in order to bring more pages. As this is done in background, the process itself does not incur in major faults, but just minor faults which will just assign the pages from the page cache to the process address space.

When first accessing a mapped file or the access pattern stops being sequential (or if its not sequential at all) a major page fault will occur. When that happens, the mapped file fault system brings an initial block to memory. This can result beneficial for programs that access

data following principles such as time, spatial and sequential reference locality, as the related pages will be warm in the page cache and the number of major page faults will be reduced. This same behaviour harms applications with random access patterns, as references to distant blocks will force the system to fault pages more often. As a result, if a random access pattern is hinted or if the heuristic determines so, the readahead mechanism is disabled.

4.3 Comparing mmap and read/write I/O functions

From the many ways that Linux offers to access files, the classic read/write(2) combination and the mmap(2) interface are the most straightforward to use and thus the more common ones. The main difference between ***classic*** and ***memory mapped*** file operations resides in the way of interacting with the file. MMAP directly maps the file into the address space, and the process can access it as if it was in memory, counting on ***mapped*** page faults to load the data into a page cache. The classic method, on the other hand, copies the file into separate buffers and then interacts with them instead.

MMAP can be seen as a simpler alternative of interacting with files comparing to the traditional read and write functions due to the evasion of a significant part of the I/O stack and the lack of frequent, direct system calls (as it can be seen on Figure 4.2).

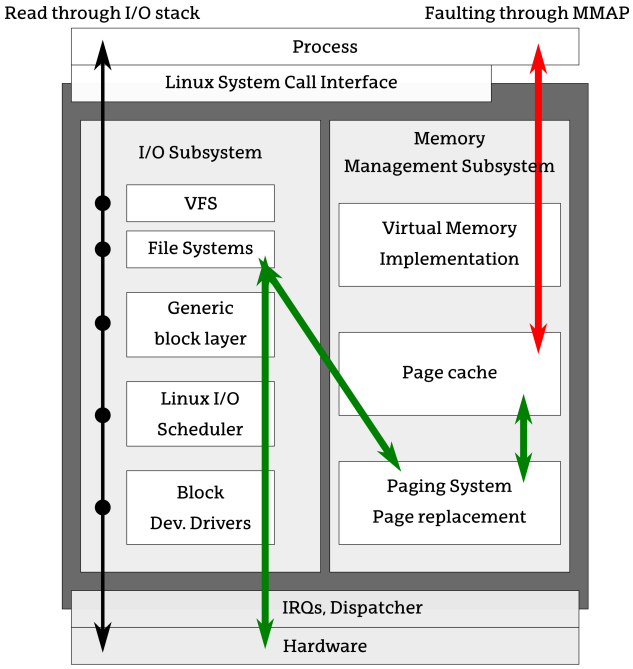
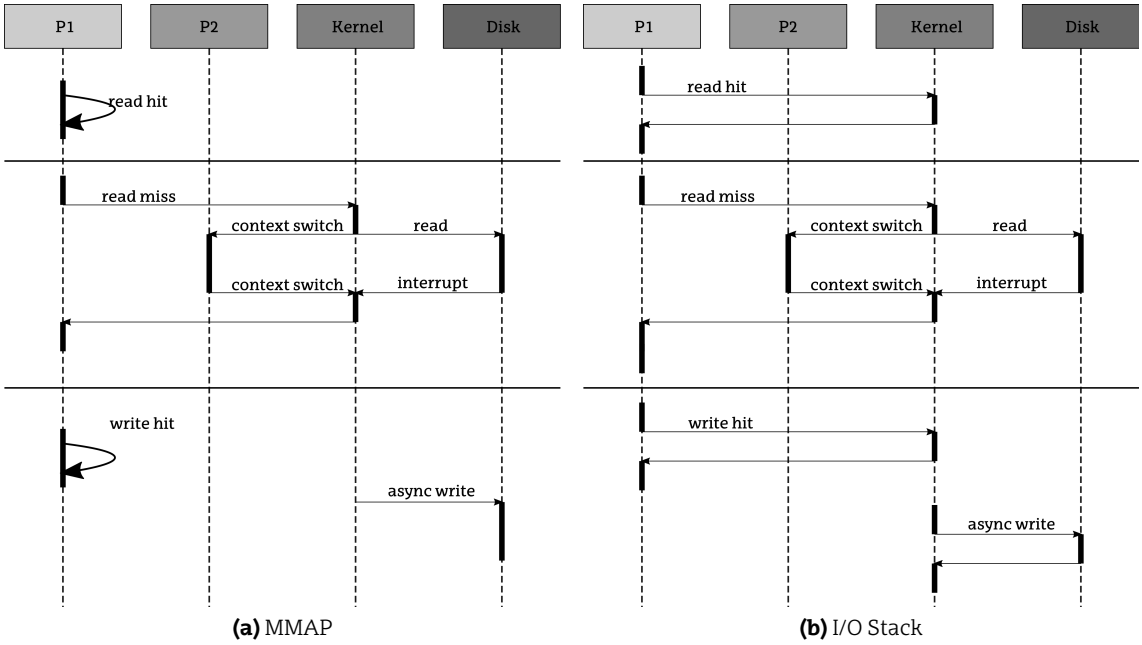


Figure 4.2: Differences between accessing a file through the I/O stack and MMAP.

While using the full stack via read/write, each interaction will result in a direct system call that must traverse through all of the I/O stack to satisfy the request, which as stated in previous chapters adds context-switch and buffer interaction overheads. With a memory-mapped region, the content is copied to a page cache in main memory in the background rather than

a process-specific buffer and in foreground, so there is no explicit context switching to kernel mode. As the page cache is global, if a thread closes a file and then re-opens it (or another thread opens it), chances are the file will still be cached as opposed with the case of using read, where the file are flushed from user-space buffer memory quickly after the file is closed.

As it can be seen in Figure 4.3, an operation on a mapped file will never directly perform a system call (apart from the initial mmap, the final munmap and possible mremap calls). This



(a) MMAP **(b)** I/O Stack

Figure 4.3: Differences between read/write operations using both mmap and I/O stack.

means that the direct context switches are vastly reduced, as mmap is run in user-space with no system call being used apart from the map creating function call. In any write case, the Virtual Memory system inside the kernel will take care of syncing the dirty pages to disk from the page cache in the background according to its configuration.

As stated before, mmap runs in user mode. This also allows the system to take advantage of the underlying vector architecture (SSE and AVX instruction set) to manage data faster. When the access pattern favors MMAP, a significant speed-up can be achieved when compared to the read/write functions. As a downside, if the application reads memory in a way that is unsuitable for MMAP, a high amount of pages will be constantly be read to page cache via mapped page fault interruptions. This may add an excessive overhead when invoked too often or with a pattern that does not suit the predefined behaviors.

In a big data environment where the file does not fully fit in the RAM, and thus can not be entirely read into it, mmap should perform better than the classic read/write functions.

4.4 Comparing mmap and read/write

The mmap interface would seem to be faster than the typical I/O stack functions to access files in certain situations. But, where does this difference really reside in terms of execution? In order to discover the real differences between the memory access via mmap and the standard read/write functions, an experiment is going to be carried.

4.4.1 Sources and credits

Given the needs of the project, an existing benchmark has been modified instead of creating one from scratch. The base is ***pm*** [Fedorova, 2019], which can be found in her GitHub repository[1]. The C benchmarking code has been altered both to modify the file access and to output the desired metrics in the desired format. A new Bash script has also been created to launch the experiment and collect the metrics in separate csv files. A Python script has also been created in order to process and plot the csv files outputted by the experiments. The generated code along with the results can be found in the DADI repository under the ***comparison*** directory[2].

4.4.2 Experimental setup

The premise is the next: ***mapped file accesses should be faster than the read/write file accesses.*** The experiment consists on reading and writing information using the same patterns and techniques over the same data, and extract the elapsed time and the obtained bandwidth. The parameters and their variations are described in the table 4.1.

Parameter	Values
Mode	mmap, syscall
Test	read, write
File size	256MB, 1GB, 4GB
Cache	cold, warm
Access	sequential, random
Block size	8KB

Table 4.1: Parameters used in the experiment and their possible values.

A previous analysis has been performed to check the impact of the working block size in the resulting bandwidth. The tested sizes were 4, 8 and 16 kilobytes. After 720 different tests, 8KB size was deemed the best size for this experiment.

Further trial-and-error experiments have been done to determine the impact of different benchmarking techniques. After the tests, the conclusion is that both methods should be reading with the same level of handicap in order to obtain reliable results. Thus, the original

[1] https://github.com/fedorova/pm
[2] https://gitlab.com/dadi-tfg/dadicode/-/tree/master/comparison

code was modified so both mmap and read/write experiments process the whole files character by character. For each combination, 5 runs will be done, each run using a seed from 1 to 5 to assure reproducibility. A mean will be calculated (plus standard deviation). This adds up for 240 unique experiments. The tests have been performed on a machine with the following specifications:

- **CPU:** Intel Xeon Gold 6130, 16 cores (32 threads) @ 3.7GHz
- **RAM:** 32GiB, DDR4-2666MHz
- **Storage:** SATA III SSD
- **OS:** Rocks 7.0 ***Manzanita*** (CentOS 7.4 kernel 3.10)

The experiment could have been run on a machine with a mechanical HDD, SATA SSD, NVMe and NV-DIMM, in order to further see which method suits better different physical device types. This was not possible due to external causes explained in a later chapter[3].

4.4.3 Analysis of the results and conclusions

The raw results (time and bandwidth) from the experiments can be found in the respective ***csv*** files in the repository[4]. The mean bandwidth along with its standard deviation can be seen in the table 4.2.

Execution	Test	256MB	dev	1GB	dev	4GB	dev
cold seq	MMrd	0.269	0.000	0.227	0.001	0.434	0.002
	IOrd	0.220	0.005	0.085	0.005	0.121	0.002
	MMwr	2.251	0.151	2.085	0.151	0.706	0.592
	IOwr	2.023	0.196	2.009	0.196	0.812	0.435
warm seq	MMrd	0.269	0.000	0.256	0.008	0.500	0.018
	IOrd	0.227	0.001	0.229	0.000	0.229	0.001
	MMwr	2.026	0.116	2.186	0.192	0.896	0.508
	IOwr	1.835	0.209	1.917	0.172	0.958	0.501
cold rnd	MMrd	0.267	0.000	0.237	0.011	0.465	0.019
	IOrd	0.153	0.004	0.155	0.001	0.106	0.003
	MMwr	2.749	0.303	2.568	0.259	2.039	0.177
	IOwr	1.855	0.172	1.711	0.129	1.271	0.177
warm rnd	MMrd	0.267	0.000	0.247	0.007	0.494	0.005
	IOrd	0.229	0.000	0.228	0.001	0.228	0.000
	MMwr	2.753	0.143	2.605	0.203	2.231	0.221
	IOwr	1.879	0.202	1.757	0.157	1.448	0.255

Table 4.2: Mean bandwidth (in GB/s) obtained in each test and its standard deviation (Xeon server).

The measurements show that mmap yields higher data transfer rates for every tests. The only exception occurs in the 4GB file sequential writes, where the write function presents a better

[3] COVID-19 disease and related confinement measures.
[4] https://gitlab.com/Ratolon/dadicode/-/tree/master/comparison/

mean bandwidth. This discrepancy can be ditched as the standard deviation is at least 50% of the mean value. As the size of the file increase, the fraction of time spent in populating the mmap translation tables becomes smaller in comparison to the actual data access time.

Writing differs from reading both in magnitude and standard deviation. Writes are in general faster than reads, and have a notable variation in terms of time and bandwidth. This could be due to the machine on which the tests were performed or its SATA controller/bus, or due to other processes running in the system.

Tests have also been performed with a 16GB file to test the behavior of both methods on scenarios where the file is larger than the system memory. Results are not shown as the hardware of the testing machine could not process the classic read/write tests in a moderate time.

Additional tests have been performed on a laptop with following specs:

- **CPU**: Intel Core i7-8550U, 4 cores (8 threads) @ 4GHz
- **RAM**: 8GiB, DDR4-2400MHz
- **Storage**: Samsung PM961 256GiB NVMe, read-write 3GiB/s - 1.7GiB/s
- **OS**: Void Linux with kernel 5.5.2 (Minimal installation with no GUI)

Results are gathered in the table 4.3 (standard deviation omitted as it is similar to the one seen in 4.2).

Execution	Test	256MB	1GB	4GB
cold seq	MMrd	0.507	0.509	1.028
	IOrd	0.375	0.359	0.352
	MMwr	2.791	0.917	0.363
	IOwr	5.04	0.301	0.269
warm seq	MMrd	0.506	0.507	1.01
	IOrd	0.39	0.084	0.071
	MMwr	3.431	2.578	0.49
	IOwr	4.237	1.031	0.316
cold rnd	MMrd	0.507	0.511	1.027
	IOrd	0.418	0.415	0.417
	MMwr	2.862	0.78	0.368
	IOwr	5.504	0.332	0.316
warm rnd	MMrd	0.504	0.504	1.009
	IOrd	0.415	0.413	0.413
	MMwr	3.765	2.068	0.498
	IOwr	4.802	1.088	0.3906

Table 4.3: Mean bandwidth (in GB/s) obtained in each test (i7 laptop).

Results show that different kernels, architectures and storage device types show different bottlenecks. Nonetheless, the conclusion stays the same: mmap performs better except in cases with high variance. The final conclusion is that mmap is a viable solution for big files, for both regular and NVMe SSD's.

CHAPTER 5

A real-life case study: BWA

The mmap interface found in Linux has been proven to be a competitor to the normally used read/write functions. It shows advantage over the classical data accessing methods in synthetic programs that were specifically created for the tests. But is it competitive in a real-world application? Is it worth the effort modifying a program to interact with the files via mmap? BWA is a data-intensive, memory-bound application used in biology-related work. The data access logic found in the application will be modified to add mmap capabilities and will then be compared to the original version in terms of performance.

5.1 Introduction

BWA (Burrows-Wheeler Aligner) is a software package for mapping low-divergent sequences against a large reference genome, such as the human genome. It consists of three algorithms: BWA-backtrack, BWA-sw and BWA-mem. As the ***mem*** algorithm is the latest incorporation to the BWA application, it is the one that is going to be looked at. The mem algorithm requires the input genome to be previously indexed into several auxiliary files. The ***index*** algorithm found in bwa is responsible for this indexing and it will also be modified.

As stated before, modifications to the code will only affect index and mem programs. The changes are only related to memory management and do not alter the results of the output files or the algorithm itself. The original program can be found in its official repository[1], while the modified version can be found in the DADI repository[2].

[1] https://github.com/lh3/bwa
[2] https://gitlab.com/dadi-tfg/bwa-1-mmap

5.2 Bwa-index

The index function generates several files parting from the original FASTA file. For a given ***.fasta*** genome the program outputs 5 files with the extensions .fasta.amb, .fasta.ann, .fasta.bwt, .fasta.pac, .fasta.sa.

The program creates a buffer to store the information of each file in memory using ***calloc(3)***. When fully processed the program writes the information to the storage in the form of the aforementioned files and finally frees the allocated memory using free(3). This algorithm occupies a big amount of physical memory, as each buffer needs to be fully allocated from start to end. If the machine is low on memory or simply does not have the required amount of RAM, the execution will fail or incur in heavy thrashing as the information will be continuously being swapped from and to main storage.

5.3 Bwa-mem

The inputs of the ***mem*** program are a "big" genome in the FASTA format, and one or optionally two "small" reads that are going to be aligned with the big one. In order to process the FASTA file, it has to be indexed using the ***index*** program first. The mem function loads the fasta file, its associated indexed files and the short reads that are to be compared with it. Similarly to what happens with index, mem loads all of the information at once into the main memory in regions allocated using ***calloc(3)***. If the available memory is not sufficient, heavy thrashing will occur and the application will run indefinitely at maximum CPU use, thus rendering the system unusable.

5.4 Motivation

In order to be successful, the modified version must both allow files that could not be processed originally to be treated in a tolerable time and not harm the execution time in cases where the files could originally fit in the system memory. That is to say, the time is not important for when the files were not originally supported, but it is crucial not to exceed by much the original time for the cases that were supported. The table 5.1 shows the output file sizes obtained by indexing two sample fasta format genome assembly files.

As a general rule, the program will need 2.72 times the fasta file size in RAM in order to index it. This requirement is greater while executing the algorithm, where all of the aforementioned files plus the (one or optionally two) readings are loaded in main memory. The total amount of necessary RAM in that case parts from 2.72 times the fasta file and its maximum value depends on the size of the readings that are to be aligned with it. If taken into account, auxiliary buffers also allocated via calloc(3) aggravate the high memory requirement.

File	SARS-CoV-2	%	ucsc.hg19	%
fasta	72.6MB	1	2.98GB	1
amb	59KB	7.9E-4	8.39KB	2.7E-6
ann	336KB	4.5E-3	3.94KB	1.2E-6
bwt	71.1MB	0.98	2.92GB	0.98
pac	17.8MB	0.24	748MB	0.24
sa	35.6MB	0.49	1.46GB	0.49
TOTAL	197.5MB	2.72	8.1GB	2.72

Table 5.1: List of files generated by the bwa index algorithm for two sample input files and their size. The percentage denotes the fraction of the original FASTA file size.

5.5 Possible solutions

Two solutions that satisfy the problem arise. The first one implies dividing the process into chunks and generating the files in smaller sections using read/write. The second consists on mapping the files using mmap, so they are loaded on-demand but fully "present" at all times for the process.

Applying the first approach (which is partially done in the original code) may seem impractical as the code has to be examined in depth in order to check if it can even be implemented in every memory-heavy point. If at any given point the system is under a big memory pressure and the chunks are not of an appropriate size, thrashing may occur as it happens in the original version. This thrashing is likely to occur due to the nature of calloc/malloc memory regions, where an evicted page ***must go*** to swap.

The second approach maps the needed files, so reads will bring information into the page cache via mapped file faults. This alternative is more attractive, as the memory does not get as clogged as with the first approach. When the memory system is under pressure, the pages in the page cache can be just evicted (uncommitted write operations would be performed before doing so) without swapping.

Both approaches also differ in terms of resource usage. The original approach may generate a big bottleneck when writing the whole file at the end. The first alternative would divide that "big" write operation into smaller writes at the end of the processing of each chunk. The second one would distribute the usage through all of the execution evenly, as the I/O operations are performed at the moment of addressing the data rather than in big chunks. This can degrade the performance by some seconds but reduce the overall stress and effect generated by bwa on the memory system and other running processes.

5.6 Modifications

As stated previously, the changes consist on mapping the big files. When creating them, the process will directly write to disk. When reading them, the process will keep only the necessary information at each time. The main difference comes after mapping the ***.bwt*** file, which

has a size of 0.98 times the size of the input fasta. The ***.sa*** file is also relatively large, having a size of 0.49 times the input fasta, so it is also mapped. Even though the ***.pac*** file is also considerably large, it is not going to be mapped to boost the speed, as it has not been deemed big enough.

5.7 Experimentation and results

The mem algorithm and its requirements vastly vary depending on the fastq read count and size, so executing it and comparing results among files in a consistent way is unfeasible. That is why the benchmarking will consist on running the index algorithm -which only depends on the input file size- over different files. The experiments will be carried on a machine with the following specs:

- **CPU:** Intel Xeon Gold 6130, 16 cores (32 threads) @ 3.7GHz
- **RAM:** 32GiB, DDR4-2666MHz
- **Storage:** SATA III SSD
- **OS:** Rocks 7.0 ***Manzanita*** (CentOS 7.4 kernel 3.10)

Table 5.2 gathers the resulting execution time of the index algorithm over different-sized files. The results show that both the original and the mapped versions perform almost equally for

Name	Fasta size	Index (orig)	Index (mmap)	Speed-up
Candida Albicans	13.8MB	6.68s	6.14s	**1.087x**
SARS-CoV-2	72.6MB	68.85s	65.91s	**1.044x**
Electrophorus Electricus	533.MB	561.3s	510.1s	**1.100x**
Asparagus Officinalis	1.1GB	1263s	1173s	**1.076x**
Homo Sapiens Sapiens	3.0GB	3942s	3611s	**1.091x**

Table 5.2: Time difference between the original bwa and the mapped bwa index algorithms.

every input file, regardless of the input fasta file size. In fact, the mapped version is strictly faster than the original. An experiment on a machine with 8GB RAM - which could not originally process the Homo Sapiens Sapiens genome because of memory requirements- shows that the indexing can be performed in approximately 3000 seconds when the processor is at its ***Intel Turbo Boost*** mode. There is an infinite speed-up as the computer could originally not process the file.

5.8 Conclusions

When using bwa ***index*** and ***mem***, if the system memory is enough to store everything, the performance of the original version is expected to be on a par (if not better) with the modified

version, as it reads everything in the beginning. The latency of reading the whole file at once from memory will be equal to that derived from faulting from disk during execution. In the case of files that exceed the memory of the machine, the time does not really matter as the original version will simply get stuck, while the modified version will be able to process the files.

This experiment shows that not only MMAP can perform faster than the classic ways of reading files, but it also allows applications to process big files with little modifications.

5.9 BWA2

A second version of the BWA algorithm has been developed, bringing significant changes. The second version is written in C++ in its majority as opposed to the full-C implementation of the first one. It also includes vast improvements on the architectural side, using AVX vector instructions when available and reorganizing the program logic to exploit the characteristics of the underlying machine (e.g. multiple threads). BWA2 was presented in the work of [Vasimuddin et al., 2019] and can be found on its repository [3].

An attempt has been done to give mmap capabilities to this second version of the program[4], but the nature of the algorithm it implements will not allow a full mmap-ing. It turns out that second version gains speed at the cost of consuming ***much*** more memory, as it decompresses everything into memory. All huge, intermediate buffers could be mapped into storage, but bwa2 would then flood the disk storage. Due to the obscurity of C++ code and time constraints, the attempt has been abandoned.

[3] https://github.com/bwa-mem2/bwa-mem2
[4] https://gitlab.com/dadi-tfg/bwa-mem2-mmap

CHAPTER 6

Solution for a synthetic application

As stated in the previous chapter, the Linux kernel and the way it manages mapped file faults can be a burden for certain applications. The performance could be improved if a read-ahead policy that satisfies the reading pattern is designed and implemented at kernel level. Throughout this chapter a synthetic application will be designed with a given access pattern, and a way to modify Linux will be explored.

6.1 Background on DBMS

The created synthetic application is inspired on the principles of a relational data base management system (RDBMS)[1]. Relational databases (RD) use projections from the real world domain in the form of ***tables***, following the relational model[2]. The data is presented as tuples (rows) of data, which contain a unique key, information and usually references to other tables. The tables are entwined and grouped using cross-references or relations (hence the name). Table 6.1 shows a fictional list of people who are to attend a congress.

Table: RITSI congress trip, people information

ID	Name	Surname	DNI	Phone No	LDAP	Bus	Hotel
001	Mikel	Iceta	72546844G	600000001	miceta003	N	N
002	Jose	Pascual	12344321E	600000002	scpasaj	Y	Y
003	Inaki	Morlan	54328461I	600000003	acpmosai	Y	Y
004	Jose M	Irizar	73816492S	600000004	jirizar009	N	N
...	...	...	...	...	...	...	...

Table 6.1: Example table from a fictional relational database.

The organization staff could use this table in conjunction with other tables in order to i.e. get the e-mail address associated to each person, or to check if they have paid the inscription.

[1] https://en.wikipedia.org/wiki/Relational_database#RDBMS
[2] https://en.wikipedia.org/wiki/Relational_model

Queries performed against a certain table may access the memory it is stored in in many different ways. For example, an access to a whole row will impact the memory in a different way than an access to a whole column[3]. This is due to the way in which the information is stored in the hard disk.

6.1.1 Storage logic and the problem with big databases

Similarly to what happens with matrices in programming languages[4], RDBMS's may store data in two different ways. While ***row-oriented*** systems serialize each tuple (or row), ***column-oriented*** systems will serialize the columns. This means that in the first model all elements from the same row are adjacent in memory, while in the second model the elements of the same column are the ones being adjacent, as depicted in Figure 6.1.

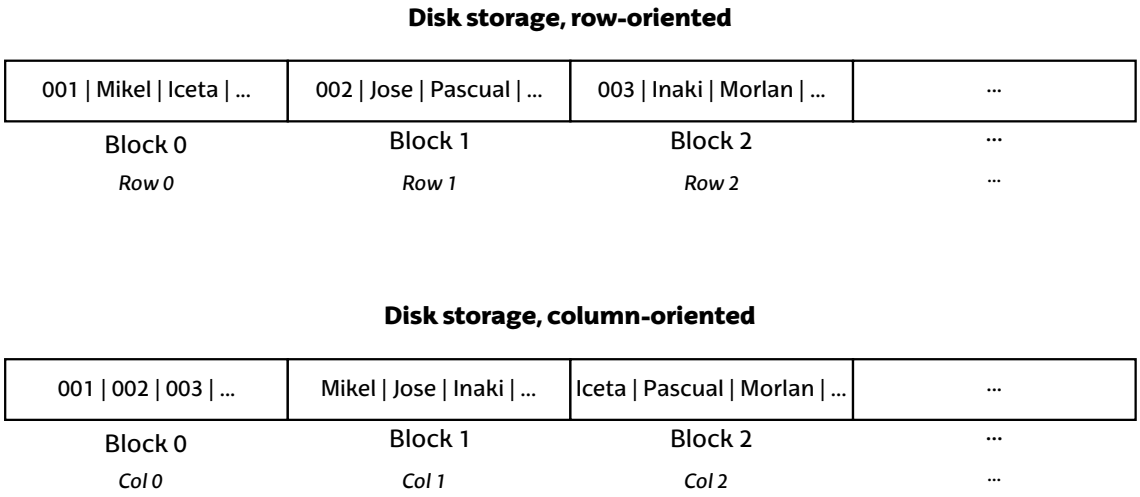


Figure 6.1: The fictional database in a hypothetical memory, with row and column orientations.

When the whole table set fits in memory (and once its fully loaded), accesses to all table positions will have a constant time cost. The same does not happen when dealing with ***huge*** tables that do not fit in the RAM, where a contrasting behavior occurs. As the tables do not completely fit in memory, I/O operations must be performed every now and then via page faults. In row-oriented systems, accesses to a whole column will result in far much more page faults than in column-oriented ones (and vice-versa), as elements will be separated one from each other by a stride instead of being adjacent. This may degrade the performance of the whole database system and impact the service time.

In previous chapters mmap has been analyzed as an alternative to the standard read/write functions when dealing with data-intensive problems. It has been proven that it allows a machine that could not process a file to be able to do so, and to do the same thing faster than read/write in machines that could handle the file. Databases suppose a big part of the data-intensive application ecosystem.

[3] This only applies to when the database does not fit in the main memory and it has to be read from disk.
[4] C stores matrices in row-major order while Fortran stores them in column-major order.

6.1.2 Designing a database-like synthetic application

A synthetic benchmark application has been written bearing in mind the nature of databases. The program opens a file, maps it into its memory space using mmap and then just reads some data, simulating a query. The elapsed time is then printed. The reading pattern is designed to be contrary to the storage logic to reproduce a situation where the page cache readahead policies will be a bottleneck. That is to say, the simulated file is considered to be row-oriented and the program reads a whole column to generate page faults that can not be satisfied by the actual sequential policy.

6.2 Modifying the Linux kernel

In order to implement a new flag for the application to use, it has first to be implemented at kernel level and compiled. The system must not only detect when the application gives hints on its memory usage and act, but it also must not disrupt the normal functioning of the system.

It is important to remark the lofty hardship of worming through a whole kernel ***composed of 27+ million lines*** with the only help being a page containing the source code[5]. The process of modifying the kernel has included installing a bare-bone Void Linux, setting up a compilation environment, endless wire-pulling from file to file to determine the execution flow of function calls and trial-and-error kernel debugging. In addition, the memory subsystem and some of the internal kernel structures happen to be quite undocumented, so a lot of blind walking was involved in the process of determining the context and exact actions performed in each function call. The asynchronous and entwined nature of the kernel make everything even more difficult to understand.

That being said, all of the modified files can be found on the project repository, in the ***kernel*** folder[6], along with some useful scripts to move the information from the editing folder to the kernel source code folders.

6.2.1 Needed modifications

The first thing needed for the kernel to adapt to a new data access pattern is the application itself to announce or give a hint on how it is going to access the memory. This is done by means of madvise(), which has some predefined flags declared in its code. Creating a new ***hint*** needs the corresponding flag to be declared in both /include/uapi/asm-generic/mman-common.h and /tools/include/uapi/asm-generic/mman-common.h. The function madvise_behavior() in the file madvise.c must also be modified in order to add a case in its internal switch to add the VM flag to the VMA structure. This also requires the corresponding VM flag to exist, which can be declared in mm.h. If the userspace GLIBC headers are not modified to add the

[5] https://elixir.bootlin.com/linux/latest/ident
[6] https://gitlab.com/dadi-tfg/dadicode/-/tree/master/kernel

new mode, the MADV flag will not be declared for userspace and the calling user code will fail to compile[7].

6.2.2 Overview of mapped file faults

The entry point of a mapped file page fault is the function ***filemap_fault()***, which is invoked via the vma operations vector from the faulting mapped memory region, and is given the information of the virtual memory fault by the VM system. The call chain is simplified and summarized in the Figure 6.2. Depending on if the page is already in memory or if it is not present, the function will call do_async_mmap_readahead() or do_sync_mmap_readahead(), respectively.

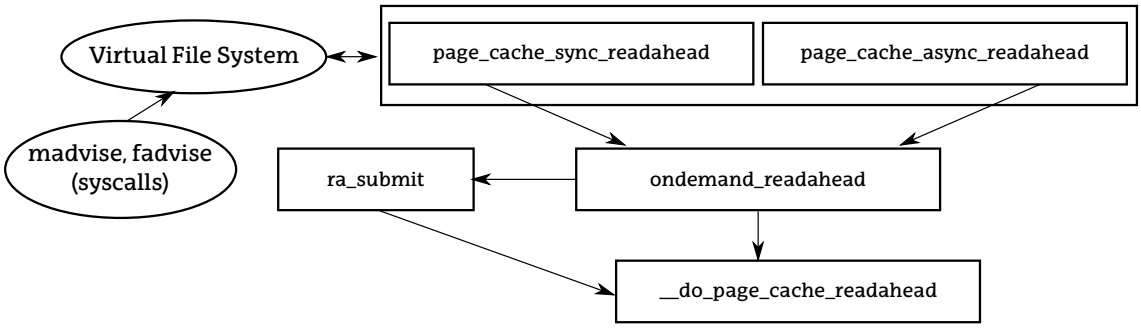


Figure 6.2: Simplified interactions and calls between the VM system and the functions.

In the case of do_sync_mmap_readahead(), the algorithm will determine via the VM flags if the access is random or sequential. If the access is random, the algorithm does nothing. If it is a sequential access (VM_SEQ_READ is present in the flags) the program calls the function page_cache_sync_readahead(). [8]

The function page_cache_sync_readahead() resides on the file readahead.c and will just call ondemand_readahead(), which in turn will also only call the function ra_submit(), which calls __do_page_cache_readahead(), returning the execution flow to the file filemap.c. This function, after several checks, will call read_pages(), which will perform the actual I/O operation and read the page.

6.3 Applying the modifications

Apart from the mentioned modifications, the new readahead should be implemented inside the call tree. This could be implemented by adding counters, or address markers to the existing programs and structures. This part has been classified as ***future work*** by force majeure, as the (later explained) circumstances have not allowed it.

[7] Actually, the user code will be able use the flag referring to it by the associated number, but not with the name

[8] If enough misses occur during a sequential read-ahead, the system automatically changes the pattern flag to VM_RAND_READ.

CHAPTER 7

Project Management

Many information is gathered and many documents are created during a project's lifetime. Project management plays a vital role in maintaining a working order and assuring the correct advancement of the project itself.

7.1 WBS Diagram

The ***Work Breakdown Structure*** diagram expresses the work that needs to be performed in a tree, with the project itself being the root. The second level of the tree contains the phases of the project and the next two levels contain the work packages and the tasks that define them. This project's WBS diagram can be seen in Figure 7.1.

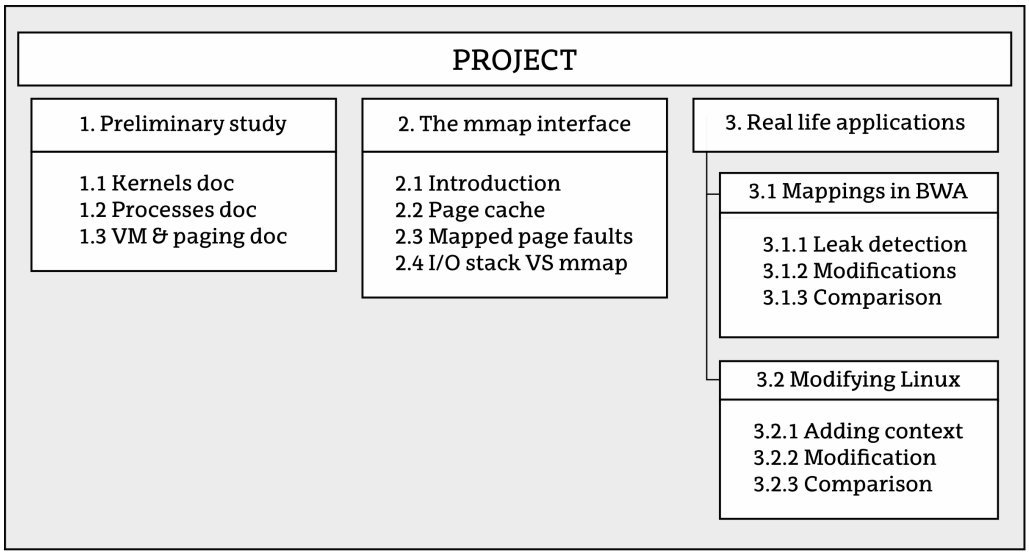


Figure 7.1: WBS Diagram of the project.

The work must be correctly separated into sequential phases. Each phase will then be decomposed into smaller and easier to process work packages. This way, the work that needs to be done can be expressed in a hierarchical manner, making it easier to understand and to perform.

7.2 Work packages and their tasks

This section is devoted to elaborating on the tasks needed to achieve the main goals of the project. A brief text is provided with each item, describing the work package or task it refers to.

7.2.1 Preliminary study

The theoretical base of the project includes the definition of some key concepts. The preliminary study will determine the base of the process of running an application.

- **Kernels**: Determine the main types of kernels and compare them.
- **Processes**: Determine how a process is created and how it interacts with the memory, other processes and the kernel itself.
- **Virtual Memory and paging**: Further study of the interactions of processes and physical memory.

7.2.2 The mmap interface

Once the basic functioning of the system is understood, a deeper dive can be done in terms of file accesses. The mmap interface seems like a good alternative, so a study of its viability has to be carried.

- **Introduction**: Summarize the usage and functioning of the mmap implementation found in Linux.
- **Page cache and mapped file faults**: Determine the approximate functioning of the page faulting system in charge of mapped file page faults.
- **Compare mmap and read/write**: Check whether mmap is a viable alternative to standard read/write system calls.

7.2.3 File mappings in BWA

A real-world application that suffers from memory leaks can be modified so it can treat bigger files without actually having the necessary amount of RAM. BWA is an existing data-intensive program that suffers from memory leaks.

- **Identify the leaks**: Determine the points where the memory is allocated.
- **Modify bwa**: Substitute the memory allocations with file mappings.
- **Comparison**: Compare the speed of both methods, old and new.

7.2.4 Modifying the Linux kernel

Modifications can be performed at kernel-level for applications that already interact with files using mmap to execute faster.

- **Context**: Find a situation where the actual page faulting system does not satisfy (enough) the running data-intensive application.
- **Modification**: Modify the kernel so it suits the application.
- **Comparison**: Check if it the modification was of any good.

7.2.5 Meta-management

This is the transversal part of the project. It includes all the tasks needed to define and control the advancements on the matter of the project. The goal of this part is to assure all of the work packets and tasks are completed. As it is meta-management, it is not considered needed including it in the WBS diagram.

- **Documentation:** This task includes all of the written documents and code documentation done throughout the project.
- **Code/information repository:** This task regards the creation and maintenance of the online GitLab repo. It includes all of the code (kernel modifications, test applications, etc.) and the information (this document, the presentation, etc.) created throughout the project.

7.3 Time estimation and deviations

Once all of the tasks needed to achieve the goals of the project are defined, a time estimation can be done in order to evaluate the scope of the project. This estimation is represented in the Table 7.1. The tasks have been extracted from the Figure 7.1. Each line contains the estimated time required for the task ***(a priori)*** and the real time that has been needed to complete it ***(a posteriori)***.

The project starts in December, 2019. The milestones for each part of the project are depicted in the Figure 7.2.

	Estimation (h)	Real (h)
PRELIMINARY STUDY		
Generic		
Kernel documentation	10	10
Linux-specific		
Process documentation	20	20
VM & page faults documentation	20	40
Experiments	20	30
THE MMAP INTERFACE		
Introduction	20	30
Page cache study	40	50
Comparison	50	50
A REAL LIFE CASE STUDY: BWA		
Identify the leaks	10	10
BWA modification	20	30
Comparison	20	10
MODIFYING THE LINUX KERNEL		
Context	10	10
Modifications	50	20
Comparison	20	-
PROJECT MANAGEMENT		
Documentation	20	20
Code/information repository	3	2
Total	**333**	**332**

Table 7.1: "Estimated vs real time" comparison for each work package group.

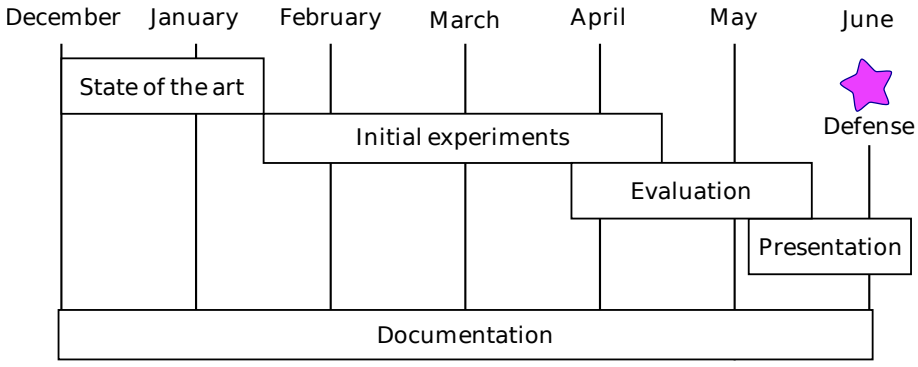


Figure 7.2: Simplified Gantt diagram of the project.

7.4 Deviation analysis

7.4.1 Lack of documentation

The Table 7.1 shows a notable time difference in the documentation parts of both Linux, its memory and the mmap interface. This deviation has been caused primarily by the lack of actual documentation, research and articles on the topics of the project. That is to say, the documentation has been mainly created rather than collected and summarized.

7.4.2 COVID-19 related back-offs

The COVID-19 pandemic is an ongoing pandemic originated from the severe acute respiratory syndrome coronavirus (SARS-CoV-2). At the time of publication, it is a global pandemic. The Spanish government has implemented a ***state of alarm*** in order to apply some restrictions and measures to contain the virus. The measures applied to prevent further spreading of the virus include self-isolation and social distancing. Businesses, public services such as the university and buildings shut themselves down in order to save lives. As a side effect, the confinement has left people stranded at home, far from their workstations and study material.

In what concerns to the project two consequences have appeared due to the state of alarm caused by the COVID-19, a technical one and a logistic one. As there was no way to access other machines, no further tests could be done in chapter 4 with the mentioned storage configurations. Also, moving back home consumes time and in this particular case removes the only quiet study spaces available. This has left the modification of the kernel out of the time scope of this thesis.

CHAPTER 8

Conclusions and future work

The end of the this work comes with some interesting conclusions:

- ***mmap*** is actually a viable alternative to read/write
  - when dealing with big files.
  - when accessing the files several times or in a one-shot fashion.
  - when performing both sequential and random accesses.
- Applications
  - can be converted to a mmap logic without modifying the underlying algorithms, getting a performance boost.
  - can be initially designed to handle files with the mmap interface.
  - could benefit from a readahead policy that better suits them than "sequential" or "random".
- The Linux kernel
  - is hard to understand, read and modify.
  - can be modified ad-hoc to satisfy the needs of a given problem, but not without a hassle.
  - can be modified to add new mapped file fault page readahead policies.

The progress of the project has been vastly slowed down by the lack of documentation on the matter. Few documents exist explaining the internal functioning of the Linux memory subsystem. In fact, few information (including papers, articles, websites, wiki pages, etc.) exists documenting the insides of the Linux kernel in general. Some books such as [Gorman, 2004], [Bach, 1989] and [Love, 2010] give an insight of the internal mechanisms found in Linux, but are either too shallow in the field of memory management and mmap or too old (kernel 2.X)

to reflect the functioning of the new Linux kernels. Profound diving into the internet and the Linux kernel source files had to be done in order to determine entry points and to define some operative structures of the kernel.

This project as a whole has supposed a personal evolution. From the day the subject and direction were defined, this has brought many challenges in terms of time administration, workload estimations and even personal management. It has also brought the opportunity to improve as a person and as a researcher, as there was not much work to rely on at the beginning.

Two main lines of action are defined for future work in this project. The first one implies **finishing** the pending tasks described in chapters 4 and 6. The second one is a little bit more complicated. It supposes **studying** data intensive applications and developing data access strategies, not only at kernel level but also at application level. This will enhance data access and thus reduce the compute time and allow processing data at a larger scale without the need of actually upgrading the hardware. Likewise, the impact of technologies (such as Intel Optane[1]) will be studied and, as a consequence, new kernel-level strategies will have to be explored for the memory subsystem to take advantage of these new kind of devices.

A third (optional) future task includes repeating the tests in a ***updated*** kernel. The provided computer had a distribution using Linux kernel 3.10, which at time of redacting the work is 7 years old (latest LTS version is 5.4.47). This means that some features present in modern hardware are not fully supported, and also means that the memory system is outdated as does not fully reflect the functioning of actual systems.

[1] https://www.intel.es/content/www/es/es/architecture-and-technology/optane-memory.html

Bibliography

[Bach, 1989] Bach, M. J. (1989). ***The design of the Unix Operating System***. Prentice/Hall, Inc., Englewood Cliffs, New Jersey.

[Duarte, 2019] Duarte, G. (2019). Webpage: Page cache, the affair between memory and files. https://manybutfinite.com/post/page-cache-the-affair-between-memory-and-files/. Accessed: 2020-06-18.

[Essen et al., 2012] Essen, B. V., Hsieh, H., Ames, S., and Gokhale, M. (2012). Di-mmap: A high performance memory-map runtime for data-intensive applications. In ***2012 SC Companion: High Performance Computing, Networking Storage and Analysis***, pages 731–735.

[Fedorova, 2019] Fedorova, A. (2019). Webpage: Why is mmap faster than system calls. https://medium.com/@sasha_f/why-mmap-is-faster-than-system-calls-24718e75ab37. Accessed: 2020-06-18.

[Gorman, 2004] Gorman, M. (2004). ***Understanding the Linux Virtual Memory Manager***. Pearson Education, Inc., Upper Saddle River, New Jersey.

[Love, 2010] Love, R. (2010). ***Linux Kernel Development, 3rd ed.*** Pearson Education, Inc., Upper Saddle River, New Jersey.

[Makrani et al., 2018] Makrani, H. M., Rafatirad, S., Houmansadr, A., and Homayoun, H. (2018). Main-memory requirements of big data applications on commodity server platform. In ***Proceedings of the 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing***, CCGrid '18, page 653–660. IEEE Press.

[Services, 2017] Services, H. L. (2017). Webpage: Page cache, the affair between memory and files. https://www.halolinux.us/kernel-architecture/page-cache-readahead.html. Accessed: 2020-06-18.

[Vasimuddin et al., 2019] Vasimuddin, M., Misra, S., Li, H., and Aluru, S. (2019). Efficient architecture-aware acceleration of bwa-mem for multicore systems. In ***2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)***, pages 314–324.