WM144-OSCC-Report

Samin Khan

March 2025

Contents

1 Introduction

This report examines the execution of the bash shell command *cp -vi .bashrc bashrc.bak* inside a Linux-based operating system. It does so by firstly explaining the role of the OS and its interaction with the user and kernel when running the command. The process analysis section details the initiation, execution, and termination of the command. This is followed by a discussion of the OS security mechanisms for managing file permissions and protecting memory, in response to executing the command. The report serves as a case study to provide a better understanding of the OS fundamentals, especially how it manages files, memory and security, when using a terminal-based command.

2 Operating System Fundamentals

Operating systems refer to a collection of programs that provide an interface between the user and computer. They are responsible for providing various functions, including memory management, file management and security (GeekForGeeks, 2025). The command *cp -vi .bashrc bashrc.bak* in this instance is run on a Linux-based OS; in the case of this report it's going to be run on the Ubuntu 64-bit distribution.

This operating system is general purpose, meaning it's able to complete a diverse array of tasks without specialising in any single one. It's also classified as an open source operating system, meaning its source code is available for anyone to view and amend without holding a license. This OS is also terminal based, meaning the command *cp -vi .bashrc bashrc.bak* can only be executed when it's placed and run via the command line interface (CLI)(Uddin and Roy, 2023).

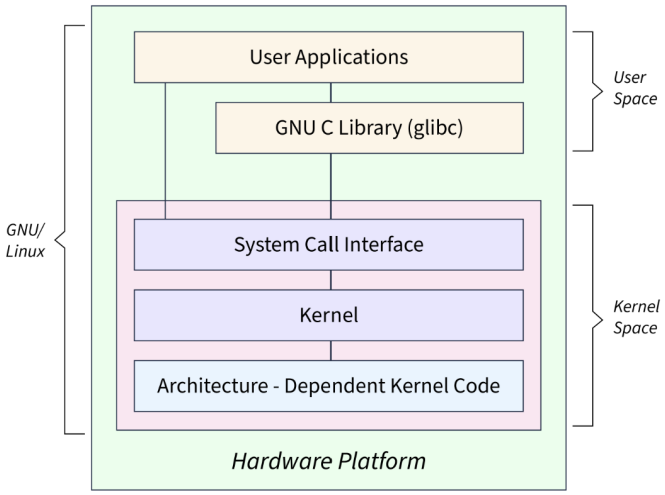

Figure 1: Fundamental Linux Architecture (Verma, 2023)

This carries advantages such as faster execution and needing less RAM, as the GUI component is removed, meaning less resources are allocated to graphics. However, its true efficacy resides in giving users precise control over the system. This is because the CLI also provides direct access to system-level utilities, making it efficient for issuing system instructions — everything from basic file operations to complex administrative tasks. In doing so, commands often trigger low-level operations that involve transitioning between user space (where application processes run with limited privileges) and kernel space (the protected core of the operating system), which can be seen in Figure 1 (GeeksforGeeks, 2024). This transition occurs via system calls (such as `open()`, `read()`, or `write()`) that safely shift execution to kernel mode to perform privileged tasks like file I/O or memory management. A distinction explored further in Section 3.1.

In terms of scheduling, Linux uses the Completely Fair Scheduler (CFS). It works by tracking each process's virtual runtime (vruntime). Processes with lower runtimes get higher priority over those with higher

run times. The selection of these processes is facilitated with the red-black tree (a self-balancing binary search tree), to quickly pick processes with lowest runtimes. In doing so Linux ensures all processes receive a fair share of CPU time. However, this does come with the cost of not being able to perform a specific operation within a guaranteed time frame. For example, tasks such as self-driving automobiles require a low-latency response, something the CFS algorithm will struggle compared to other solutions such as Real Time (Ubuntu Documentation, 2025).

3 Process Lifecycle Analysis

3.1 Process Creation

The user first checks the existence of the command inside the OS by first running the *ls -la | grep .bashrc* , as it's a hidden file (dotfiles in Unix) (Verma, 2022). The result of which is seen below in Figure 2:

```
samin-yasie-khan@samin-yasie-khan-VirtualBox:~$ ls -la | grep .bashrc
-rw-r--r--  1 samin-yasie-khan samin-yasie-khan 3771 Mar 31  2024 .bashrc
samin-yasie-khan@samin-yasie-khan-VirtualBox:~$ cp -vi .bashrc bashrc.bak
'.bashrc' -> 'bashrc.bak'
```

Figure 2: .bashrc being duplicated as bashrc.bak

This command begins with *cp*, which stands for copy and is used to duplicate files or directories in Unix-based systems. It is then followed by -v (verbose flag) ensuring that each file copied is displayed in the terminal, and the -i (interactive mode) which introduces a confirmation step before overwriting an existing file. The remaining two tokens are *.bashrc* and *.bashrc.bak*, where the former is the source file and the latter serves as its backup. Behind the screen, the shell parses the user's input and identifies the required files and options. It does so to begin the process creation stage by invoking the fork() system call, which creates a new process that enters the NEW state briefly before being moved to the READY queue as shown in Figure 3 (Silberschatz, Baer Galvin and Gagne, 2018, pp.107–109, Khan, 2025). Here it then waits to be scheduled by the CPU.

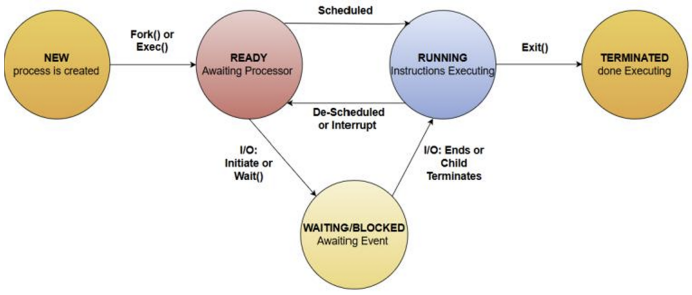

Figure 3: The Process States (Khan, 2025)

When selected by the scheduler, it transitions to RUNNING, and begins executing the copy operation in "user mode" , which by default gives low privileges. This transition from READY to RUNNING involves context switching, where the processor stores the state of the previous process and loads the state of *cp*, allowing the processor to resume its execution correctly. If higher privileges are required for a process, the process can use system calls (syscalls) to switch into "kernel mode" by generating an interrupt, enabling execution of higher process actions. For example, when a process wants to read a file, it has to use the read() system call to request access from the OS. When the process makes a system call, it shifts into kernel mode, allowing the OS to access the file directly on behalf of the process. After completion, the process switches back to user mode. This setup acts as a safeguard, making sure that any file operations follow the permission rules defined by the system.

3.2 Memory Management

After transitioning the process into RUNNING state, memory is allocated to the *cp* process by the OS virtual memory management. This technique involves making a section of the hard drive act as if it was RAM, giving each process the illusion of having its own contiguous block of memory, while actually mapping it to physical memory via address translation in order for the process to start executing in a secure and isolated way. Figure 4 demonstrates this by using the command to display the allocation of memory through the *pmap* command:

Figure 4: The cp process being given its own virtual memory space

When the *cp* process is created, the operating system allocates it a private virtual address space as shown in Figure 5 , where virtual memory is being used. In the case of Linux, it uses virtual memory to ensure each process only sees memory addresses relative to the start of their own space. This then ensures process isolation as it does not access the actual physical memory directly.

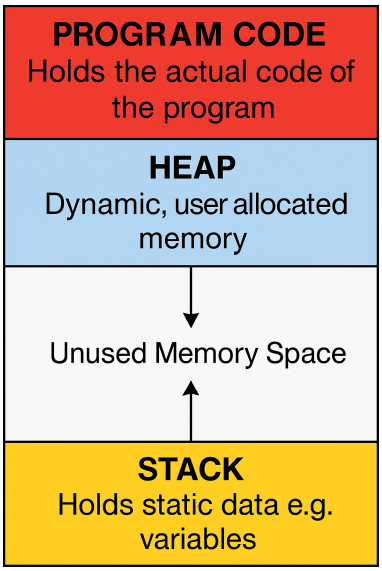

Figure 5: The address space layout (Khan, 2025)

This isolation is maintained through the use of the Memory Management Unit (MMU), which checks for a small cache called the Translation Lookaside Buffer (TLB) to store recent mappings. If the address being accessed is not found in the TLB, the MMU performs a page table walk, by digging through the page table to find where that memory lives in physical RAM (The Linux Kernel Documentation, n.d.). Each page in memory has permissions tied to it(read, write, or execute). If a process attempts to violate these permissions by trying to access memory that has not loaded in RAM, a page fault is triggered. This signals the kernel to either load the missing page into memory or, if the access is invalid, shut the process down with a SIGSEGV (segmentation fault). These protection mechanisms help ensure that memory cannot be accessed in unsafe or unauthorised ways.

3.3 File Management

The *cp* command carries out a series of low level file operations, which are handled by the Linux file management system. In the case of *cp -vi .bashrc bashrc.bak*, the system opens the file in read-only mode for the OS to verify whether the calling process has the appropriate read permissions for the file. If granted, the contents and file permissions of the source file (*.bashrc*) are first read into memory through the use of the read() command, before copying these to the destination file (*bashrc.bak*). This can be seen below in Figure 6, where the *ls -l* command highlights both files having " rw-r–r–" permissions. This means only the owner " samin-yasie-khan" has the ability to read and write, while the group and others can only read the files:

```
samin-yasie-khan@samin-yasie-khan-VirtualBox:~$ ls -l .bashrc bashrc.bak
-rw-r--r-- 1 samin-yasie-khan samin-yasie-khan 3771 Mar 31  2024 .bashrc
-rw-r--r-- 1 samin-yasie-khan samin-yasie-khan 3771 Mar 15 13:25 bashrc.bak
```

Figure 6: The file permissions of the command

To further understand how file access is managed with the *cp* token inside the *cp -vi .bashrc bashrc.bak* command , the inode values (unique identifiers that reference the file's metadata and disk location) of the files involved were examined. The results are shown in Figure 7, which displays inode values of 1444117 for *.bashrc* and 1501992 for *bashrc.bak*.

```
samin-yasie-khan@samin-yasie-khan-VirtualBox:~$ ls -i .bashrc bashrc.bak
1444117 .bashrc  1501992 bashrc.bak
```

Figure 7: The inode values of the command

Once these inode values are resolved, the OS proceeds to build out the internal file management structures. This includes assigning file descriptors within the process, mapping to entries in the file table. This file table contains both file access permissions and the offset values, which for this case were assumed to be 0 as per standard OS behaviour for newly opened files. These file table entries, in turn, point to their relative inode table entries, completing the chain of access as shown in Figure 8.

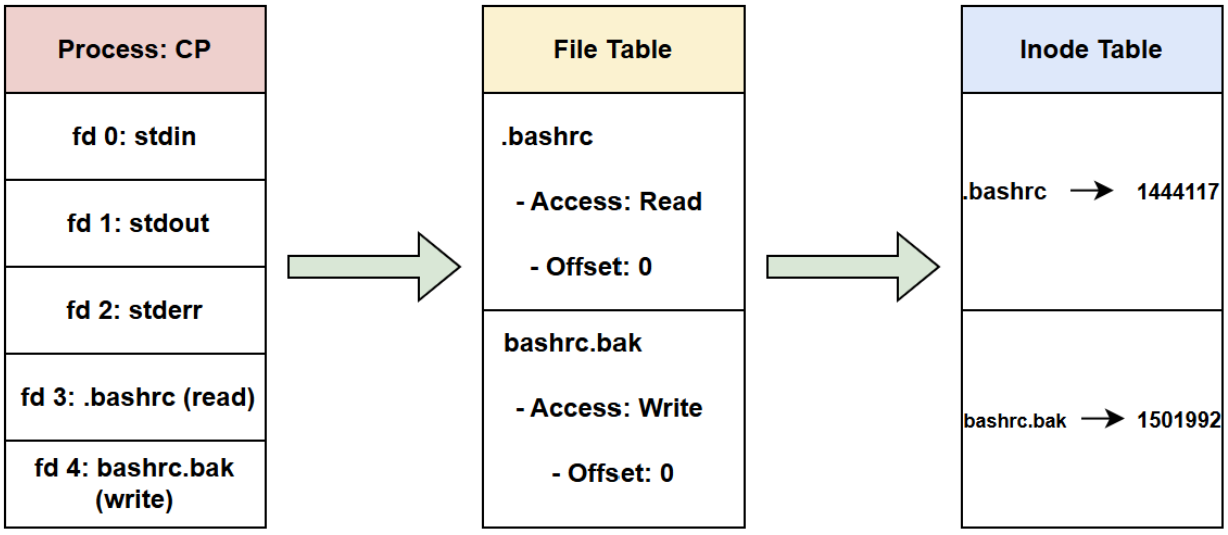

Figure 8: The File Descriptor Table (Adapted by Khan, 2025, from Linux TLDR, 2024)

3.4 User Interaction

The -v (verbose) flag outputs each copy operation to the terminal, while the -i (interactive) flag prompts the user for confirmation before overwriting existing files. In the case of trying to overwrite *bashrc.bak*, the system prompts with *cp*: overwrite *'bashrc.bak'?*, waiting for user input.

At this point, the process pauses by executing a system call like read() in the background; during this time the process is temporarily suspended and held in memory, meaning the processor does not fully release it, but it won't let the process proceed until user input has been entered.

This allows for the user to type "yes" to confirm the continuation of the operation as seen in Figure 9. This is because the handling is done via standard input (stdin) and standard output (stdout), which explains the interactive behaviour of Linux CLI tools.

```
samin-yasie-khan@samin-yasie-khan-VirtualBox:~$ strace -o cp_trace.txt cp -vi .bashrc bashrc.bak
cp: overwrite 'bashrc.bak'? yes
'.bashrc' -> 'bashrc.bak'
```

Figure 9: The terminal prompt showing user interaction during the cp-vi command

This user input is then passed to the shell, which has already forked and executed the *cp -vi* process. The management of these input and output streams are then shown in Figure 10:

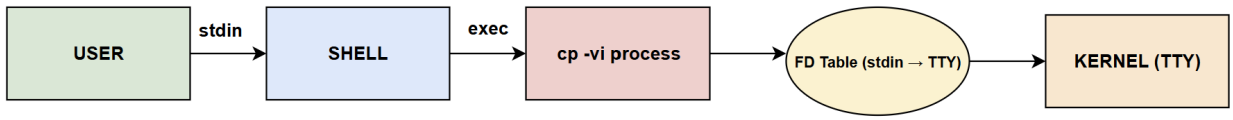

Figure 10: Flow of user input and output from user to kernel (Khan, 2025)

Figure 10 starts with representing the USER typing on the keyboard. The input is then sent via stdin to the command interpreter(SHELL), causing it to use the exec() system call to launch the *cp -vi* process. Once launched, the process takes over the execution responsibility, while still depending on the shell to handle user inputs.

The *cp -vi* process then relies on the file descriptor table to map stdin, stdout, and stderr to the terminal (TTY). This links these descriptors to kernel-level I/O interfaces, since they are handled by the operating system. This in turn completes the communication loop between the user, shell, process and system.

During this process, the shell operates in canonical mode(Arcege, 2011), also known as "cooked". This mode is the default in most Linux terminals, and it works by allowing certain characters like Ctrl+D and Ctrl+U to be interpreted by terminal drivers, which are line-based systems (Arcege, 2011). This behaviour explains why `cp -vi` remains paused until the user submits their response, as the shell holds the entire line in a buffer until the user presses Enter, at which point the input is delivered to the process all at once.

3.5 Process Termination

All I/O tasks, including flushing any buffer data or closing tabs, are marked completed in the system logs. The process then proceeds to terminate cleanly by invoking the exit group(0) system call, signaling a successful termination. Part of this transition involves triggering the deallocation of the process control block (PCB), which releases the OS level meta data such as the process ID, scheduling priority, and execution state. Additionally, the kernel now performs comprehensive resource cleanup: reclaiming any unfreed heap memory, detaching shared memory segments and IPC channels, and closing all open file descriptors including standard I/O streams.

```
samin-yasie-khan@samin-yasie-khan-VirtualBox:~$ ls
bashrc.bak     Documents            gef    peda-arm  pwndbg     Videos
cp_trace.txt   Downloads            Music  Pictures  snap
Desktop        gdb-peda-pwndbg-gef  peda   Public    Templates
samin-yasie-khan@samin-yasie-khan-VirtualBox:~$ tail cp_trace.txt
copy_file_range(3, NULL, 4, NULL, 9223372035781033984, 0) = 3771
copy_file_range(3, NULL, 4, NULL, 9223372035781033984, 0) = 0
close(4)                                = 0
close(3)                                = 0
lseek(0, 0, SEEK_CUR)                   = -1 ESPIPE (Illegal seek)
close(0)                                = 0
close(1)                                = 0
close(2)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
samin-yasie-khan@samin-yasie-khan-VirtualBox:~$
```

Figure 11: End of strace showing file descriptor closure and process termination

This system call notifies the kernel that the process is ready to shut down and has exited without errors, as seen in Figure 11, which is a continuation from Figure 9. During this stage, if a process had established shared memory or inter process communication channels, they are either detached or marked for removal. This signals the kernel to take over and begin its cleanup job of closing all open file descriptors and standard input and output. Once finished, the kernel sends a SIGCHLD signal to inform the completion of the parent's child process, which in this case is the shell that initially spawned the *cp* command.

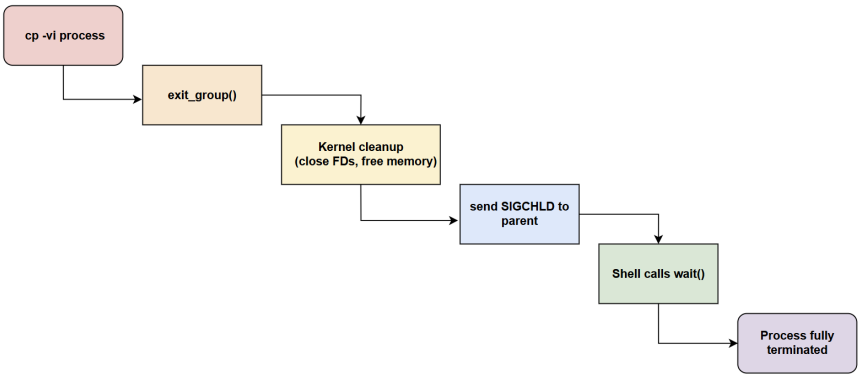

Figure 12: Termination flow from process exit to shell via exit group(0) and SIGCHLD (Khan, 2025)

Upon receiving this signal, the shell executes the wait() system call to retrieve the child process's termination status, preventing the creation of a zombie process. Figure 12 maps out this logical flow, starting from when the user signals the process termination, to then how the kernel performs cleanup on the resources, ending with control returning back at the shell once everything's been properly shut down.

3.6 Process Lifecycle Table

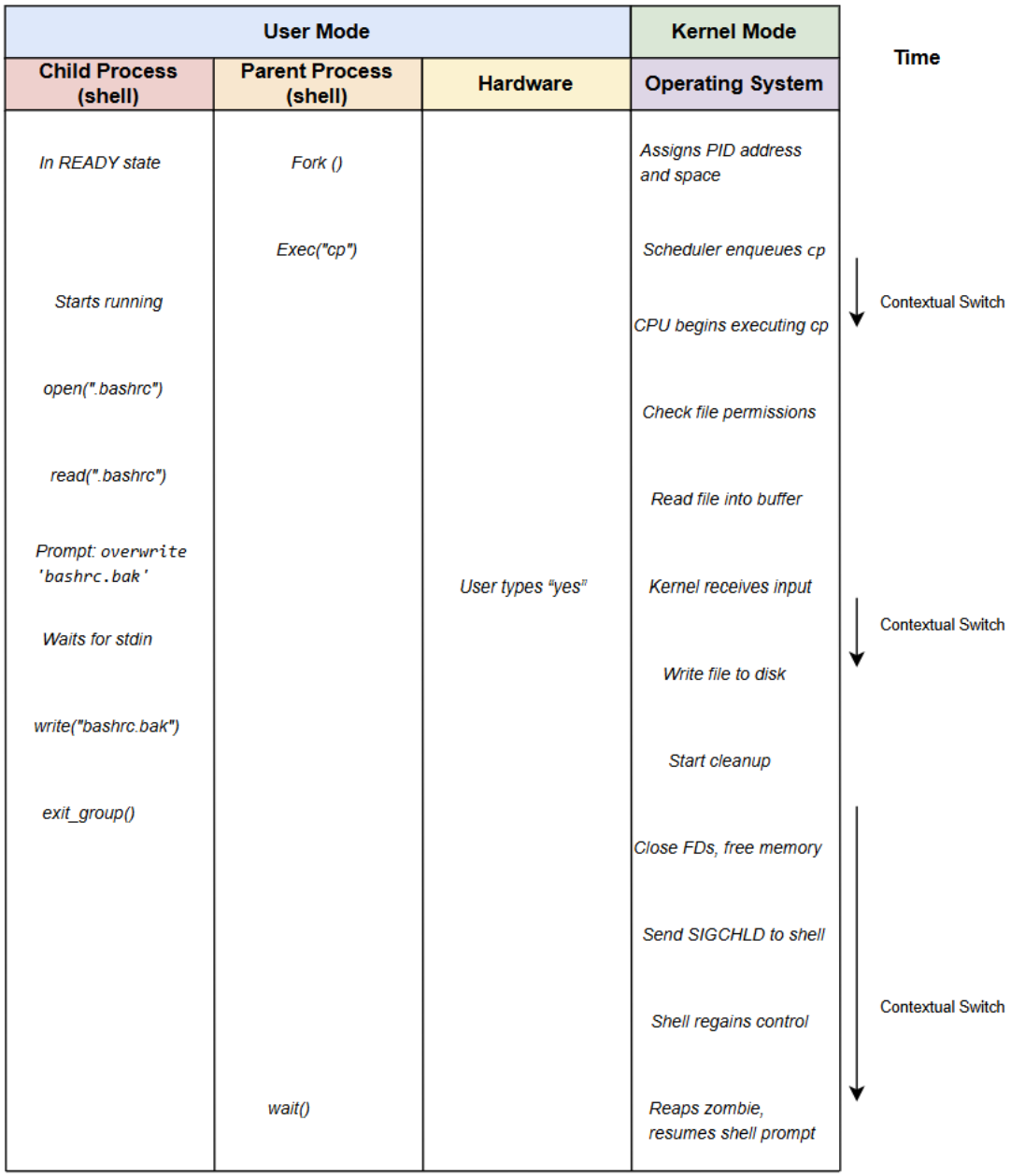

Figure 13: The lifecycle of *cp -vi .bashrc bashrc.bak* (Khan, 2025)

Figure 13 summarises and highlights the user/kernel, memory management and signal handling, with arrows showing what time contextual switching occurs.

4 Security Considerations

4.1 File Permissions

In addition to just displaying permissions via *ls -l*, Linux enforces them at runtime through system-level checks. In the case of Figure 6, only the owner (samin-yasie-khan) had read and write permissions, while the group and others only had the read permission. This however changed when the user removes all access rights from *.bashrc* using *chmod 000*. The command works by ensuring that no user, not even the file's owner, can open it unless permissions are restored. This means the next time the user tries to execute the *cp* command, it will fail since this new lack of permissions will trigger a permission check failure in the kernel. This can be seen in Figure 14 , where the system prevents the *cp* process from accessing *.bashrc* and returns a "Permission denied" error. This enforcement is implemented via the kernel's VFS(Virtual File System) layer, which checks access permissions against the file's inode metadata before allowing operations such as open(), read(), or write().

```
samin-yasie-khan@samin-yasie-khan-VirtualBox:~$ chmod 000 .bashrc
samin-yasie-khan@samin-yasie-khan-VirtualBox:~$ cp -vi .bashrc bashrc.bak
cp: overwrite 'bashrc.bak'? yes
'.bashrc' -> 'bashrc.bak'
cp: cannot open '.bashrc' for reading: Permission denied
samin-yasie-khan@samin-yasie-khan-VirtualBox:~$
```

Figure 14: Failed attempt to access .bashrc after removing all file permissions

4.2 Memory Protection

Once the *cp* process enters the RUNNING state, the virtual memory management system of the operating system allocates a slice of memory to it. This is known as a virtual address space which is completely isolated, meaning the process can't read another program's memory or write to any area that wasn't specifically assigned to it. This has already been shown in Figure 5, where the virtual space is segmented into 4 distinct regions: program code, heap, stack, and free memory.

This isolation is enforced through the Memory Management Unit (MMU), also known as paged memory management unit (PMMU). This is a hardware unit responsible for translating virtual memory addresses into physical addresses. It does so by checking for a small cache called the Translation Lookaside Buffer (TLB) to store recent mappings. If the address being accessed is not found in the TLB, the MMU performs a page table walk, by searching through the page table to find where memory lives in the RAM (The Linux Kernel Documentation, n.d.).

If a program tries to use memory that isn't currently in RAM, a page fault occurs. This alert tells the operating system's kernel to either load the missing memory page or, if the memory access is not allowed, terminate the program using a segmentation fault (SIGSEGV). These mechanisms are used to help the system prevent unauthorised access, ensuring that only intended programs are able to use memory.

4.3 Conclusion

Through the lens of the *cp -vi .bashrc bashrc.bak* command, this report illustrated how a Linux-based operating system manages process execution from creation to termination. The research displayed how the Linux based operating system effectively manages user/kernel interactions while utilising virtual memory, file permissions, and security. Understanding these help to see how contemporary operating systems handle memory, security, and instruction processing while preserving functionality and performance.

5 References

- Arcege. (2011). *What's the difference between a raw and a cooked device driver?* [online] StackExchange. Answered 30 September. Available at: `https://unix.stackexchange.com/questions/21752/what-s-the-difference-between-a-raw-and-a-cooked-device-driver` [Accessed: 20 April 2025].
- GeeksforGeeks. (2024). *Difference between User Mode and Kernel Mode.* [online] Last updated: 28 December. Available at: `https://www.geeksforgeeks.org/difference-between-user-mode-and-kernel-mode/` [Accessed: 19 April 2025].
- Khan, S.Y. (2025). *Address space layout diagram.* Created by the author. (Unpublished)
- Khan, S.Y. (2025). *Flow of user input and output diagram.* Created by the author. (Unpublished)
- Khan, S.Y. (2025). *Process lifecycle diagram for cp -vi .bashrc bashrc.bak.* Created by the author. (Unpublished)
- Khan, S.Y. (2025). *Termination flow diagram.* Created by the author. (Unpublished)
- Linux TLDR. (2024). *What are File Descriptors in Linux?* [online] Published 9 January. Available at: `https://linuxtldr.com/file-descriptors-linux/` [Accessed: 19 April 2025].
- Silberschatz, A., Galvin, P.B. and Gagne, G. (2012). *Operating System Concepts.* 9th edn. Hoboken, NJ: John Wiley & Sons.
- The Linux Kernel Documentation. (n.d.). *Page Tables.* [online] Available at: `https://docs.kernel.org/mm/page_tables.html` [Accessed: 19 April 2025].
- Ubuntu Documentation. (2025). *Schedulers explained.* [online] Published 28 January. Available at: `https://documentation.ubuntu.com/real-time/en/latest/explanation/schedulers/` [Accessed: 19 April 2025].
- Uddin, B. and Roy, M. (2023). *What is Command in Linux? [A Complete Overview].* [online] LinuxSimply. Last updated: 9 November. Available at: `https://linuxsimply.com/command-in-linux/` [Accessed: 19 April 2025].
- Verma, J. (2022). *What is .bashrc file in Linux?* [online] DigitalOcean. Published 3 August. Available at: `https://www.digitalocean.com/community/tutorials/bashrc-file-in-linux` [Accessed: 19 April 2025].
- Verma, S. (2023). *Linux Kernel Architecture.* [online] Scaler. Last updated: 27 July. Available at: `https://www.scaler.com/topics/linux-kernel-architecture/` [Accessed: 19 April 2025].