Operating System Draft

CSCI 312

Instructor: Dr. Zubaidah AlHazza

26 March 2025

Fatima Farooq
2023006109

Rabaa Ismail Alshbib
2022005690

Lana Omar Zanneh
2022005620

Alya Alzaabi
2022005560

Roodh Alblooshi
2022005691

Dema Ammar Al-sos
2021004885

Department of Computer Science and Engineering
American University of Ras Al Khaimah
2024-25

Table of Contents

1 Excective summery

1.1 Introduction

Operating system concept is the interface between the software and the hardware systems. Zorin OS, a Linux distribution based on Ubuntu, has gained attention for its ease of use and user-friendly interface to be explored in this case study. It will discuss core functions like process and threads, synchronization, deadlocks, memory & file management using real implementation of the OS (Fatima). This analysis will examine the underlying mechanisms of these systems in practice, not just as theoretical concepts but as they are implemented in Zorin OS. The methodology involves monitoring tools such as ps aux, top, and strace. , the study will address the legal and ethical implications of using open-source operating systems

1.2 Process Creation and Management

Zorin OS 17.3, now with Linux kernel 6.8, enhances parallel I/O, power, and multi-core scheduling efficiency for improved process management. At the most fundamental level, every process on the system is structured using a detailed framework known as `task_struct`, which comprises peripherals such as memory and scheduling. Process creation utilizes standard Linux mechanisms with the system calls `fork()` and `clone()`. `fork()` replicates the parent process, while `clone()` grants finer control over resource-sharing through flags. In succession, new processes typically execute a fresh program using `exec()`. Scheduling is the responsibility of the Completely Fair Scheduler (CFS), which allocates CPU usage using a red-black tree to fairly distribute it through virtual runtime, thus maintaining responsive performance during desktop usage. Optimized context switching, which is a resource-intensive task, has to be tuned carefully to maintain overall system efficiency and directly impacts system performance. Processes can be monitored and managed using graphical interfaces such as GNOME System Monitor, and command-line based ones like `ps`, `top`, `htop`, and `kill`, with options to adjust process priorities to control how CPU time is allocated. This enables casual and power users to easily influence system behavior.

1.3 Threading Model and Implementation

Zorin OS implements threading with a 1:1 model mapping user-level NPTL threads to kernel threads via glibc's Native POSIX Thread Library. Scheduling and signal handling in this architecture are simplified while maintaining compliance with the standards of the multi-core systems. Threads are constructed with pthreadcreate() which, in turn, invokes clone() with specific resource sharing flags such as shared memory and file descriptors, but separate stacks and registers. Economical thread-local storage is granted and management control through threaded functions like pthreadjoin(), pthreaddetach(), and pthreadcancel() provides control shrouded by thread encapsulation. The encapsulation of threads relies on fast user space mutexes (futexes) for low overhead atomic user-space contended checks and kernel access during contention. Fewer resources are used to govern mutexes, condition variables, and semaphores with the encapsulation of threads through user space futexes that supersede kernel threads. Best outcomes of the 1:1 model are offered through CPU resource-bound exercises, however, constrained resources for performing tasks such as context switching accomplish pointers towards event driven computing or managing asynchronously input output heavily along with driven concurrent tasks puts the techniques of thread pools to good use.

1.4 System Strengths and Limitations

Key features of Zorin OS, which is based on the Linux kernel, include robust resource control with cgroups, responsive desktop performance with dynamic time slicing, and scalability with the CFS scheduler. While NPTL offers high-performance, POSIX-compliant threading with effective futex-based synchronization and true parallelism on multi-core systems, its process isolation improves stability and security. Nevertheless, there are drawbacks, such as complicated scheduling and context switching overhead, increased memory and resource consumption for threads and processes, and less control over scheduling granularity than with user-space models. Zorin OS is well-suited for most contemporary workloads due to these trade-offs, though applications with high concurrency or limited resources might need to be optimized.

1.5 Process Scheduling in Zorin OS

Using a red-black tree to ensure fair CPU time allocation based on process priority and virtual runtime, the Completely Fair Scheduler (CFS) is at the heart of Zorin OS's powerful, proactive multitasking and scheduling system, which is based on the Linux kernel. To handle a range of workloads, from standard user applications to crucial real-time tasks like multimedia and robotics, it supports several scheduling types, including default / normal (`SCHED_OTHER`), real-time

(SCHED_FIFO, SCHED_RR), and advanced (SCHED_DEADLINE). By preserving and restoring process states, context switching enables seamless multitasking even on single-core systems, and interrupt handling guarantees prompt reactions to hardware events. Improved responsiveness, better CPU utilization, and seamless multitasking are among its strengths; however, if real-time policies are handled incorrectly, lower-priority tasks may be starved. Niceness values are also supported by Zorin OS for user-level priority control.

1.6 Synchronization in Zorin OS

Inheriting Linux kernel features, Zorin OS manages concurrent access to shared resources using a thorough set of synchronization techniques at the kernel and user levels. Important methods are read/write locks for high-read, low-write situations like logging and configuration management, futexes for quick user space locking, spinlocks for busy-waiting in performance-critical code, semaphores for resource availability tracking, and mutexes for mutual exclusion. Synchronization in Zorin OS covers process sequencing, thread coordination, data consistency (e.g., file or cloud sync), and system time accuracy via NTP. By guaranteeing consistent, predictable behavior across apps and services, these systems help to avoid problems such race conditions and deadlocks. Proper use of locks, critical sections, and condition variables is essential for maintaining system integrity, especially in multithreaded and real-time environments.

1.7 Deadlock in Zorin OS

Deadlock in Zorin OS (built on Linux) happens when processes mutually block one another by holding resources while waiting for others, therefore generating a circular dependency. The four required criteria are: 1- Mutual Exclusion (non-sharable resources), 2- Hold and Wait (processes keep resources while asking for others), 3- No Preemption (resources cannot be forcibly taken), and 4- Circular Wait (cyclic dependency chain). Zorin OS offers tools for diagnosis which are : 1- ps aux (displays processes in "D" state), 2-strace (traces blocked system calls), and 3- top (finds stuck processes), but it lacks automatic deadlock detection. Scripts where processes lock resources A/B while waiting for B/A allow for simulated deadlocks. Resolution is timeouts, resource preemption (rare), or manual termination (kill -9). Lock ordering (set resource acquisition sequence), avoidance, and other prevention techniques are included.

1.8 Memory Management

Memory management in Zorin OS lets applications run even when they surpass physical RAM restrictions by using disk space (swap area) as an extension of memory, therefore running on Linux's virtual memory and paging. Managed by a page table, paging splits memory into fixed-size pages and frames; if data is not physically in RAM, a page fault happens and the needed page is loaded from disk. 1- Contiguous allocation (simple but prone to fragmentation), 2-paging (efficient and avoids external fragmentation), 3- segmentation (logical memory division), are among memory allocation techniques. With Zorin OS preferring paging plus Best Fit to strike performance and reduce fragmentation, dynamic allocation methods such as First Fit, Best Fit, and Worst Fit maximize memory use. Swapping transfers inactive processes to disk, freeing RAM.

1.9 File Management

File management in Zorin OS is centered on using files and directories to organize data. Each file has a name, extension, and metadata—aside from the filename—stored in inodes(index node). Regular files, directories, symbolic links, and special device files are examples of file types. Zorin OS uses the extra file system, which supports inode-based metadata, journaling, and handling large files. Permissions granted to the owner, group, and others (such as -rwxr-xr–) regulate file access, assisting in preventing system abuse and unwanted access. Symbolic links can span across file systems and point to file paths, whereas hard links directly refer to a file's inode. Zorin OS employs caching and buffering, batching write operations, and storing recent reads in RAM for efficiency. ACLs, file permissions, and other security measures.

1.10 Legal and Ethical Issues

In order to guarantee that its software remains free, open-source, and shareable, Zorin OS complies with the GNU General Public License v2.0, which also mandates that any modified versions remain open. The system is protected by LUKS full-disk encryption to safeguard data at rest, AppArmor , which restricts app behavior, and UFW, a basic firewall manager that prevents unwanted access. With a focus on privacy, Zorin OS provides easy-to-use settings for managing location, file history, and app permissions, particularly for Snap and Flatpak apps. It only gathers anonymous information when the user agrees to it, and it even keeps a warrant canary to ensure that no covert access or monitoring has been asked for. Zorin OS is morally sound due to its harmony of privacy, security, and openness.

1.11 Analyze Process and Thread Management

Thread analysis of Zorin OS reveals a Linux-standard model in which threads, also called Light Weight Processes (LWPs), share process identifiers and memory to facilitate effective multitasking. Real-time user and kernel thread inspection is aided by programs such as top, htop, ps, and the /proc filesystem. The gdm-wayland-ses process, for example, starts several threads that are normally sleeping and waiting for system calls or user input. The system's ability to scale in response to load is demonstrated by the dynamic thread count it maintains, which ranges from 192 to 443. Lightweight but necessary kernel threads like kworker and rcu use less power than user-space threads, such as those under gnome-shell and gnome-terminal. Thread hierarchy and resource usage are clearly displayed using tools like htop, demonstrating how GUI elements tend to.

2 Introduction

2.1 Background and Motivation

Operating systems (OS) are essential components of computer systems, acting as the interface between hardware and software. They manage system resources such as processing time, memory, input/output devices, and file systems, ensuring that various software programs run smoothly. The OS's role is crucial because it enables applications to interact with the hardware without needing to understand the complex hardware specifics.

Linux, a Unix-based operating system, has gained popularity in recent years due to its stability, security, and open-source nature. It is widely adopted in server environments and has a growing presence in personal computing. Zorin OS, a Linux distribution based on Ubuntu, has gained attention for its ease of use and user-friendly interface. Zorin OS makes it simple for users transitioning from other operating systems, especially Windows, to adapt to Linux while retaining all the advantages that come with it.

The primary motivation for this case study is to explore Zorin OS in depth, focusing on how it manages various OS functions such as process management, memory handling, file systems, and more. This analysis will examine the underlying mechanisms of these systems in practice, not just as theoretical concepts but as they are implemented in Zorin OS. The study will also address practical issues such as synchronization between processes, deadlock management, and how the system handles resources and memory.

2.2 Objective of the Case Study

The main aim of this study is to conduct an in-depth analysis of how Zorin OS handles fundamental operating system principles. We will examine specific areas such as:

- **Process and Threads:** Investigating how Zorin OS manages processes and threads, including creation, scheduling, and termination.
- **Process Scheduling:** Understanding the process scheduling mechanisms in Zorin OS and how CPU time is allocated to various tasks.
- **Synchronization:** Looking at how Zorin OS ensures proper synchronization between concurrent processes to avoid race conditions and deadlock.
- **Deadlock:** Analyzing how Zorin OS detects, prevents, and resolves deadlock situations.
- **Memory Management:** Exploring how Zorin OS handles memory allocation, paging, and virtual memory.
- **File Management:** Investigating how Zorin OS manages files, directories, and data storage.
- **Legal and Ethical Issues:** Discussing the legal and ethical considerations associated with using open-source operating systems like Zorin OS.

By focusing on Zorin OS, this case study aims to provide practical insights into how operating system principles are implemented in a modern Linux-based system. This hands-on approach allows us to analyze these OS functions as they are executed on a real system.

2.3 Scope and Methodology

This study is limited to Zorin OS, a specific Linux-based operating system, and will not extend to other distributions of Linux or different operating systems. The goal is to provide a detailed understanding of how Zorin OS implements key

OS functions, such as process management, scheduling, synchronization, memory handling, file systems, and deadlock resolution.

The methodology involves using various system monitoring tools available within the Linux environment, such as `ps aux`, `top`, and `strace`. These tools will help us observe system performance and behavior in real-time. By running relevant system commands, we will gather data about process states, memory usage, and resource allocation. Additionally, we will simulate scenarios such as process deadlock to see how the OS handles such issues.

To support the analysis, we will include screenshots, figures, and tables where necessary. These visuals will help clarify complex concepts and illustrate how the system behaves under different conditions.

2.4 Significance of the Study

The significance of this case study lies in its ability to bridge theoretical operating system concepts with practical application. Operating systems, while often discussed in the abstract, have concrete implementations that can vary significantly from one OS to another. By analyzing Zorin OS, we aim to provide a comprehensive understanding of how the operating system manages core functions in a real-world environment.

This analysis will also provide valuable insights for system administrators, developers, and users who interact with Linux-based systems. Understanding the mechanisms of process management, memory handling, and deadlock prevention is essential for anyone working with or developing for Linux systems. Additionally, the study will address the legal and ethical implications of using open-source operating systems, which is an increasingly relevant topic as open-source software becomes more widely adopted.

3 Process Creation & Management

3.1 Kernel Architecture Overview

Zorin OS 17.3 is built on Linux kernel 6.8, which provides the foundation for its process management capabilities. This kernel version introduces several improvements over previous iterations, including enhanced parallel I/O processing, better power management, and optimized scheduling for multi-core systems.

At the core of process management is the `task_struct` structure, which serves as the process descriptor. This structure, defined in `include/linux/sched.h` in the kernel source code, contains all the information the kernel needs to manage a process:

```
struct task_struct {
    /* Process identification */
    pid_t pid;
    pid_t tgid;

    /* Process state */
    volatile long state;    /* -1 unrunnable, 0 runnable, >0 stopped */

    /* Scheduling information */
    int prio, static_prio, normal_prio;
    unsigned int rt_priority;
    struct sched_entity se;
    struct sched_rt_entity rt;

    /* Memory management */
    struct mm_struct *mm, *active_mm;

    /* File system info */
    struct fs_struct *fs;
    struct files_struct *files;

    /* Signal handlers */
    struct signal_struct *signal;
    struct sighand_struct *sighand;
```

```
    /* ... many more fields ... */
};
```

Each process in Zorin OS has its own `task_struct`, allocated dynamically when a process is created. The kernel maintains a circular doubly-linked list of all `task_struct` structures, allowing it to traverse all processes in the system efficiently. This comprehensive data structure enables the kernel to track and manage every aspect of a process's execution, from its memory usage to its scheduling parameters.

3.2 Process Creation Mechanisms

Process creation in Zorin OS follows the standard Linux model, primarily using two system calls: `fork()` and `clone()`. These mechanisms provide the foundation for launching applications and services throughout the operating system.

The `fork()` system call creates a new process by duplicating the calling process. This creates a child process that is an exact copy of the parent process, including all memory segments, file descriptors, and other resources. After the fork, both processes continue execution from the point where `fork()` was called, with the only difference being the return value of the `fork()` call (zero for the child process and the child's process ID for the parent).

When `fork()` is called, the kernel performs several operations:

1. Allocates a new `task_struct` using `dup_task_struct()`
2. Sets up unique identifiers (PID)
3. Copies or shares resources based on flags
4. Initializes scheduling parameters
5. Adds the new task to the scheduler's runqueue

The `clone()` system call is a more flexible alternative to `fork()` that allows for more granular control over what resources are shared between the parent and child processes. It provides finer control through flags:

```
/* Example clone flags */
#define CLONE_VM       0x00000100  /* Share memory space */
#define CLONE_FS       0x00000200  /* Share filesystem info */
#define CLONE_FILES    0x00000400  /* Share file descriptors */
#define CLONE_SIGHAND  0x00000800  /* Share signal handlers */
#define CLONE_THREAD   0x00010000  /* Same thread group */
```

After a process is created via `fork()`, it typically executes the `exec()` system call to replace its memory image with a new program. This combination of `fork()` and `exec()` is the standard method for launching new applications in Zorin OS.

3.3 Process Scheduling

Zorin OS 17.3, with Linux kernel 6.8, uses the Completely Fair Scheduler (CFS) as its default process scheduler. CFS was designed to provide fair CPU time allocation to all processes based on their weights.

The CFS implementation in Linux Kernel 6.8 uses a red-black tree data structure to track runnable processes:

```
/* CFS rbtree node structure */
struct sched_entity {
    struct load_weight load;
    struct rb_node run_node;
    struct list_head group_node;
    unsigned int on_rq;
    u64 exec_start;
    u64 sum_exec_runtime;
    u64 vruntime;
    u64 prev_sum_exec_runtime;
    u64 nr_migrations;
    /* ... more fields ... */
};
```

Key technical aspects of CFS implementation include:

1. **Red-Black Tree:** CFS maintains a time-ordered red-black tree of runnable tasks, sorted by their virtual runtime (`vruntime`).
2. **Virtual Runtime Tracking:** Each process has a `p->se.vruntime` value that represents the amount of CPU time it has received, normalized by the process's weight. This allows processes with higher priority (lower nice value) to accumulate `vruntime` more slowly.
3. **Minimum vruntime Tracking:** The scheduler maintains `rq->cfs.min_vruntime`, which tracks the smallest `vruntime` among all tasks in the runqueue, ensuring that newly activated tasks are placed appropriately in the tree.
4. **Task Selection:** CFS always selects the "leftmost" task from the red-black tree (the one with the smallest `vruntime`) to run next, ensuring that the process that has received the least amount of CPU time relative to its priority gets to run.
5. **Preemption:** When a task's `vruntime` becomes sufficiently larger than the leftmost task's `vruntime`, preemption occurs, allowing the system to maintain fairness.

The CFS implementation in Zorin OS provides nanosecond granularity accounting and does not rely on fixed timeslices, making it more responsive and fair than older scheduler implementations. This is particularly important for desktop environments where responsiveness is a key user experience factor.

3.4 Context Switching

When the kernel decides to switch from one process to another, it performs a context switch. This involves saving the current state of the CPU (register values, program counter, etc.) for the process being switched out and loading the saved state of the process being switched in.

The context switch operation in Linux is highly optimized but still represents a non-trivial overhead. It involves:

1. Saving the current process's CPU registers
2. Updating memory management structures
3. Loading the new process's CPU registers
4. Potentially flushing the Translation Lookaside Buffer (TLB) if switching between different address spaces

Context switching is a critical operation that directly impacts system performance. Excessive context switching can lead to reduced throughput due to the overhead involved. Zorin OS's kernel is optimized to balance the need for responsiveness (which may require frequent context switches) with the need for efficiency (which benefits from fewer context switches).

3.5 Process Management Tools in Zorin OS

Zorin OS provides several tools for users to manage processes:

1. **GNOME System Monitor:** This is the graphical task manager in Zorin OS that allows users to view running processes, their resource usage, and to terminate processes if necessary. It provides a user-friendly interface for basic process management tasks, making system monitoring accessible to non-technical users.
2. **Command-line Tools:** Zorin OS includes standard Linux command-line tools for process management:
   - `ps`: Lists running processes
   - `top` and `htop`: Provide real-time views of process activity
   - `kill`, `pkill`, and `killall`: Send signals to processes, typically to terminate them
   - `nice` and `renice`: Adjust process priorities
3. **Process Priorities:** Zorin OS allows users to adjust process priorities to control how CPU time is allocated. This is done using the `nice` value, which ranges from -20 (highest priority) to 19 (lowest priority). Only privileged users can set negative `nice` values. The process priority can be set when launching a process or adjusted for running processes using the `renice` command or through the GNOME System Monitor.

These tools provide both novice and advanced users with the means to monitor and control process execution, helping to maintain system performance and stability.

4 Threading Model & Implementation

4.1 NPTL Architecture

Zorin OS implements threading through the Native POSIX Thread Library (NPTL), which is fully integrated into the GNU C Library (glibc). NPTL replaced the older LinuxThreads implementation to provide better POSIX compliance and performance.

NPTL was developed to address limitations in the older LinuxThreads implementation, particularly in the areas of signal handling, scheduling, and inter-process synchronization primitives. The development of NPTL involved collaboration between Red Hat and IBM, with the NPTL team incorporating the best features from both implementations.

Key technical aspects of NPTL in Zorin OS include:

1. **Full POSIX Compliance:** NPTL provides closer conformance to the POSIX threads standard than its predecessors.
2. **Performance Optimization:** NPTL is designed for high performance, with optimized thread creation, termination, and synchronization operations.
3. **Scalability:** The implementation scales well to systems with many threads, supporting modern multithreaded applications.
4. **Integration with glibc:** NPTL is fully integrated into the GNU C Library, providing seamless threading support for applications.

The NPTL architecture in Zorin OS provides a robust foundation for multithreaded applications, enabling efficient utilization of multi-core processors and responsive application behavior.

4.2 1:1 Threading Model

Zorin OS, through NPTL, implements a 1:1 threading model, where each user-level thread corresponds directly to a kernel-level thread. This means that when an application creates a thread using the `pthread_create()` function, a new kernel scheduling entity is created.

In the Linux kernel, there is no distinct concept of a "thread" versus a "process" at the kernel level. Instead, Linux implements all threads as standard processes that share certain resources. When a thread is created, the kernel uses the `clone()` system call with specific flags that indicate which resources should be shared between the parent and child processes.

All threads within a process are organized into a "thread group" and share the same thread group ID (which is equivalent to the process ID of the first thread). Each thread also has its own unique thread ID, which can be obtained using the `gettid()` system call.

The 1:1 threading model was chosen over the M:N model (where M user-space threads map to N kernel threads) because:

- It simplifies implementation
- With the CFS scheduler, there is minimal performance penalty
- It provides better integration with the kernel's scheduling and signal handling

Threads within a process share:

- Code segment (text)
- Data segment
- Heap
- File descriptors
- Signal handlers

- Process ID

Each thread has its own:

- Thread ID
- Stack
- Register values
- Thread-local storage
- Signal mask
- Scheduling properties

This model provides a good balance between simplicity, performance, and functionality for most applications running on Zorin OS.

4.3 Thread Creation and Management

Thread creation in Zorin OS occurs through the `pthread_create()` function, which internally uses the `clone()` system call with specific flags:

```
/* Simplified example of clone flags used for thread creation */
clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND | CLONE_THREAD |
      CLONE_SYSVSEM | CLONE_SETTLS | CLONE_PARENT_SETTID |
      CLONE_CHILD_CLEARTID | CLONE_DETACHED, ...);
```

These flags ensure that the new thread shares the appropriate resources with the parent thread while maintaining its own thread-specific data.

Thread-Local Storage (TLS) is a critical component of the threading implementation, allowing each thread to have its own private data area. NPTL implements efficient TLS through:

- Using the `set_thread_area()` system call on x86 systems
- Utilizing architecture-specific registers (e.g., `fs` register on x86)
- Providing efficient per-thread data storage

Thread management in Zorin OS includes:

1. **Thread Creation:** Using `pthread_create()` to spawn new threads.
2. **Thread Termination:** Threads can terminate by returning from their start function, calling `pthread_exit()`, or being canceled by another thread.
3. **Thread Joining:** The `pthread_join()` function allows one thread to wait for another to terminate and retrieve its return value.
4. **Thread Detachment:** Detached threads (`pthread_detach()`) automatically release their resources when they terminate.
5. **Thread Cancellation:** Threads can be canceled using `pthread_cancel()`, with cancellation points and cleanup handlers to ensure proper resource management.

This comprehensive thread management system provides developers with the tools they need to create efficient multithreaded applications on Zorin OS.

4.4 Thread Synchronization

A critical component of NPTL's efficiency is its use of futexes (Fast Userspace Mutexes) for thread synchronization. Futexes were introduced to minimize the overhead of system calls during synchronization operations.

```
/* Basic futex operations */
#define FUTEX_WAIT        0
#define FUTEX_WAKE        1
#define FUTEX_FD          2
```

```
#define FUTEX_REQUEUE     3
#define FUTEX_CMP_REQUEUE 4
```

Futex operation combines user-space and kernel-space components:

1. **User-space:** Atomic operations attempt to acquire/release locks without system calls
2. **Kernel-space:** When contention occurs, the `futex()` system call manages thread blocking and waking

Example of a simplified mutex implementation using futexes:

```
/* Simplified mutex implementation using futexes */
int mutex_lock(int *mutex) {
    int c;
    /* Try to acquire lock without system call */
    if (atomic_cmpxchg(mutex, 0, 1) == 0)
        return 0;

    /* Contention: use futex system call */
    while (1) {
        c = atomic_xchg(mutex, 2);
        if (c == 0)
            return 0;
        futex(mutex, FUTEX_WAIT, 2, NULL, NULL, 0);
    }
}
```

NPTL provides various synchronization primitives to coordinate between threads:

- Mutexes: Provide mutual exclusion, allowing only one thread to access a protected resource at a time.
- Condition Variables: Allow threads to wait for a specific condition to become true.
- Read-Write Locks: Allow multiple readers or a single writer to access a resource.
- Barriers: Synchronize a group of threads at a specific point in execution.
- Semaphores: Control access to a resource with a counter.

The futex mechanism enables NPTL to implement these POSIX synchronization primitives with minimal overhead, significantly improving the performance of multithreaded applications in Zorin OS.

4.5 Thread Performance Analysis

The 1:1 threading model used in Zorin OS provides good performance for CPU-bound tasks on multi-core systems, as threads can be scheduled on different CPU cores for true parallel execution. However, creating and managing a large number of threads can be resource-intensive.

Performance considerations for threading in Zorin OS include:

1. **Thread Creation Overhead:** Creating a thread involves allocating kernel resources and setting up thread-specific data structures. This operation is more expensive than creating a lightweight user-space thread in an M:N model.
2. **Context Switching Overhead:** Each thread context switch involves kernel intervention, which adds overhead compared to user-space thread switching.
3. **Memory Usage:** Each thread requires its own stack and kernel resources, which can limit the number of threads that can be created.
4. **Cache Locality:** Threads running on different CPU cores may experience cache coherency overhead when accessing shared data.

For applications that require a large number of concurrent tasks, alternative approaches are often recommended:

- **Thread Pools:** Reuse a fixed number of threads to execute multiple tasks
- **Asynchronous I/O:** Use non-blocking I/O operations to handle multiple connections with fewer threads

- **Event-driven Programming:** Use event loops to handle multiple tasks within a single thread

These approaches can help applications achieve better scalability and performance on Zorin OS, particularly for I/O-bound workloads or when dealing with a large number of concurrent connections.

5 System Strengths and Limitations

5.1 Process Management Strengths and Limitations

The process management implementation in Zorin OS demonstrates several technical strengths:

1. **Scalability:** The CFS scheduler scales efficiently to systems with many cores and processes, making Zorin OS suitable for both desktop and server workloads. The red-black tree data structure used by CFS has $O(\log n)$ complexity for insertion, deletion, and lookup operations, ensuring good performance even with a large number of processes.
2. **Responsiveness:** The use of virtual runtime and dynamic time slicing provides good interactive performance, essential for a desktop-oriented distribution. By prioritizing processes that have received less CPU time relative to their weight, CFS ensures that interactive processes remain responsive even under system load.
3. **Resource Control:** Integration with `cgroups` allows for fine-grained resource control, enabling better system stability under load. Administrators can set limits on CPU, memory, and I/O usage for groups of processes, preventing any single application from monopolizing system resources.
4. **Process Isolation:** Strong process isolation mechanisms prevent processes from interfering with each other, enhancing system stability and security. Each process has its own virtual address space, file descriptors, and other resources, limiting the impact of process failures.

However, there are also some limitations:

1. **Overhead:** The comprehensive process management system introduces some overhead, particularly for process creation and context switching. This can impact performance in scenarios with frequent process creation or high context switching rates.
2. **Complexity:** The sophisticated scheduling algorithms and process management mechanisms can be complex to understand and tune for specific workloads. Default settings may not be optimal for all use cases.
3. **Memory Usage:** Each process requires its own memory structures, which can limit the number of processes that can be created on systems with limited memory.

5.2 Threading Model Strengths and Limitations

The NPTL threading implementation in Zorin OS provides several advantages:

1. **Performance:** The 1:1 threading model combined with futex-based synchronization delivers excellent performance for multithreaded applications. By mapping each user-space thread directly to a kernel thread, NPTL enables true parallel execution on multi-core systems.
2. **POSIX Compliance:** Full POSIX threading API compliance ensures compatibility with a wide range of applications. Developers can rely on standard threading interfaces without worrying about implementation-specific quirks.
3. **Scalability:** The implementation scales well to systems with many threads, supporting modern multithreaded applications. The efficient futex-based synchronization minimizes overhead for thread coordination.
4. **Robustness:** Features like robust mutexes and thread cancellation handling improve application stability. Robust mutexes can detect and recover from situations where a thread holding a mutex terminates unexpectedly.

However, the 1:1 threading model also has some limitations:

1. **Resource Usage:** Each thread requires its own kernel resources, which can limit the number of threads that can be created. This can be a limitation for applications that need to create thousands of threads.
2. **Context Switching Overhead:** Thread context switches involve kernel intervention, which adds overhead compared to user-space thread switching in an M:N model. This can impact performance in scenarios with frequent thread switching.

3. **Scheduling Granularity:** The kernel scheduler may not have application-specific knowledge that would enable optimal thread scheduling decisions. User-space schedulers in an M:N model can potentially make better scheduling decisions based on application-specific requirements.

6 Process Scheduling in Zorin OS

6.1 Overview

In Zorin OS, process scheduling is essential for controlling the distribution of system resources across active activities, ensuring responsiveness and effective multitasking. Based on the Linux kernel, Zorin OS inherits a strong and well-optimized scheduling engine that can efficiently manage workloads on both desktops and servers. In order to balance system performance and fairness, the kernel uses a process scheduler to decide which process is running at any given time. Both preemptive and non-preemptive scheduling are supported by Zorin OS, which enables the system to either allow processes to finish voluntarily or force task switching when required.The Completely Fair Scheduler (CFS), the default scheduler found in the majority of Linux-based systems, including Zorin, seeks to allocate CPU time to each running process equitably according to its priority and weight. It effectively manages operations and schedules them to resemble fair queuing using a red-black tree data structure. Furthermore, Zorin OS supports traditional scheduling methods such as Round Robin, Priority Scheduling, Shortest Job First (SJF), and First-Come-First-Served (FCFS), however these are more theoretical and only applied in particular situations or for teaching.With its emphasis on usability and performance, Zorin OS makes sure that background services, user applications, and system daemons receive fair and timely access to CPU resources, which contributes to a smoother and more responsive user experience, especially on older hardware. Real-time scheduling policies like FIFO (First In, First Out) and RR (Round Robin) are available for time-critical applications, providing deterministic behavior essential for certain multimedia or industrial tasks. Zorin OS also considers process niceness values, which control priority without requiring root access, allowing users to control how much CPU time processes receive.

6.2 Types of Scheduling

1. **SCHED OTHER** (Normal/Default Scheduling) Type: Time-sharing (used for routine chores that don't require immediate attention)

Completely Fair Scheduler (CFS) was the scheduler used.

Use Case: Most user processes, desktop apps, and background activities.

It operates by dynamically allocating CPU time to processes according to "virtual runtime" in order to maintain equity.

2. **SCHED FIFO** (Real-Time First-In, First-Out) Real-time

Use Case: Robotics and audio processing are examples of critical operations.

How it operates: Unless a higher-priority real-time task is received, processes execute in the order they are scheduled and are not preempted.

Note: If not handled correctly, it may starve regular tasks.

3. **SCHED RR** (Real-Time Round Robin) Real-time

Use Case: Real-time applications that require timeslice fairness and predictable behavior.

It functions similarly to FIFO, except it adds a set time slice. Then, the subsequent process with the same priority level is executed.

More Equitable: Guarantees a turn for every real-time process.

4. Advanced Real-Time Scheduling **(SCHED DEADLINE)** Real-time, with the earliest deadline first

Use Case: Complex real-time applications, such as industrial control, robotics, and multimedia.

How it works: Each task's runtime, deadline, and period can be specified.

CPU Scheduling Hierarchy

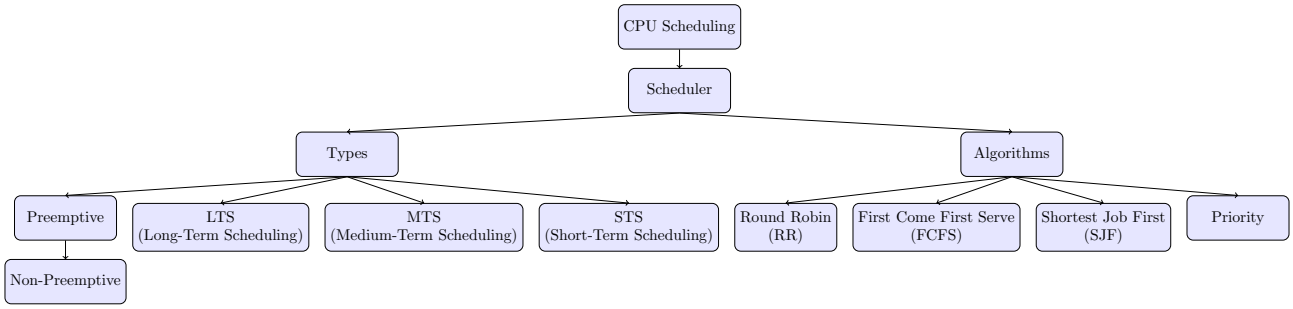


Figure 1: CPU Scheduling Hierarchy in Zorin OS

6.3 Context Switching and Interrupt Handling in Zorin OS

Like the majority of Linux-based distributions, Zorin OS uses a preemptive multitasking model, which gives the kernel authority over whether to switch and interrupt processes. Even on a single processor, the operating system can smoothly switch between threads and processes thanks to a technology called context switching, which gives the appearance of simultaneous operation. To ensure effective multitasking, the kernel saves the state of the current process (including registers and stack) during a context transition, which is brought on by events like interrupts or system calls. It then loads the state of the next scheduled process. Because they enable communication between the kernel and hardware elements like I/O devices, interrupts are essential to this process. For instance, an interrupt is created when a network packet is received or a USB device is plugged in, causing the kernel to react appropriately. The Linux kernel's standard method for handling these interruptions is used by Zorin OS. interruptions can be either mask-able, which permits temporary deactivation, or non-mask-able, which necessitates rapid attention, as in the event of serious hardware failures.

6.4 Advantages of Scheduling in Zorin OS

- Better Multitasking: Zorin OS can effectively manage several tasks at once thanks to scheduling, giving the appearance of simultaneous execution. Context switching between tasks guarantees seamless multitasking without observable lags, even on single-core computers.

- Improved CPU Utilization: Zorin OS guarantees that the CPU is always used efficiently through efficient scheduling, avoiding idle periods and optimizing performance. As a result, the system becomes more responsive and tasks are completed more quickly.

- Enhanced Responsiveness: Zorin OS's preemptive scheduling methodology makes sure that high-priority tasks receive CPU time as soon as possible. This makes the system more responsive to background processes, user inputs, and outside events, guaranteeing a seamless user experience.

7 Synchronization in Zorin OS

7.1 How Zorin OS synchronization works?

Zorin OS inherits the entire set of synchronization methods offered by the Linux kernel since it is based on Ubuntu. Both the kernel level (for drivers and system activities) and the user level (for apps and services) make use of these techniques. Typical methods include of:

Mutexes (Mutual Exclusion): The vital area can only be accessed by one thread or process at a time. Access to shared resources is managed using semaphores, which are counters. Spinlocks: In a spinlock, a thread waits in a loop to see if it can get to the crucial part.

Threads can wait for specific criteria to be met by using condition variables.

Fast Userspace Mutexes, or futexes, are a quicker method that doesn't go into kernel mode unless absolutely necessary. Both kernel-level code (device drivers, kernel modules) and user-level applications (such as system daemons, apps) use them.

7.2 Synchronization types include:

1) synchronizing the Process: regulates the sequence in which processes are executed (for example, when one operation needs to wait for another to finish). utilized in pipelines, shared memory, or inter-process communication (IPC).

2) synchronizing Threads: controls how threads within the same process are executed. stops several threads from concurrently accessing the same memory.

3) Synchronization of Data: maintains data consistency across several apps or devices. Examples include file synchronization across devices and cloud sync (e.g., Dropbox on Zorin or Synching).

4) Time/Clock Synchronization: uses the Network Time Protocol (NTP) to make sure the system time is accurate. crucial for network operations, activities, and logs.

How multiple processes or threads access shared resources?

- **Mutexes and Semaphores:** Mutexes (mutual exclusions) and semaphores are employed in Zorin OS to control access to shared resources. A mutex only allows one thread at a time to access a vital area, ensuring strict exclusivity. Semaphores, on the other hand, can handle both exclusive and limited shared access by keeping a count of resource availability. They are commonly utilized in resource management for both the user- and kernel-space. Proper handling of these synchronization tools is critical, particularly for dealing with edge cases and avoiding issues like deadlocks and resource leaks.
- **Spinlocks:** Spinlocks are another synchronization mechanism that protects critical code areas from concurrent access. When a thread attempts to acquire a spinlock that is already held, it continuously checks ("spins") until it becomes available. Spinlocks are commonly used in low-level kernel operations or interrupt handlers and are also useful in uniprocessor systems due to potential preemption. Unlike semaphores, spinlocks do not allow sleeping while waiting, making them ideal for performance-critical and real-time activities within the kernel.
- **Read/Write Locks:** Read-write locks (reader-writer semaphores) improve concurrency when many threads need read access but only a few require write access. This mechanism allows multiple threads to read from a shared resource at once while ensuring exclusive access for writing. In Zorin OS, read-write locks are especially useful in tasks such as logging, configuration management, and caching systems, where reads are frequent, and writes are infrequent but essential.

7.3 Python Thread Synchronization Example (with and without Lock)

```
import threading

# Shared variable
counter = 0

# Create a Lock object
lock = threading.Lock()

# Function to increment counter WITHOUT lock (unsafe)
def unsafe_increment():
    global counter
    for _ in range(100000):
        counter += 1

# Function to increment counter WITH lock (safe)
def safe_increment():
    global counter
    for _ in range(100000):
        with lock:
            counter += 1

# Test WITHOUT synchronization
def run_without_lock():
    global counter
    counter = 0
    t1 = threading.Thread(target=unsafe_increment)
    t2 = threading.Thread(target=unsafe_increment)
    t1.start()
    t2.start()
    t1.join()
    t2.join()
    print("Without Lock, counter =", counter)

# Test WITH synchronization
def run_with_lock():
    global counter
    counter = 0
    t1 = threading.Thread(target=safe_increment)
    t2 = threading.Thread(target=safe_increment)
    t1.start()
    t2.start()
    t1.join()
    t2.join()
    print("With Lock, counter =", counter)

# Run both tests
run_without_lock()
run_with_lock()
```

7.4 Brief Summary of synchronization

Synchronization in computing is the coordination of processes or threads to ensure that they operate in a controlled and predictable manner, particularly when accessing shared resources. This is critical to avoiding conflicts, such as race conditions, in which the outcome of an operation is determined by the timing or sequence of occurrences.

The key methods for synchronization are:

- **Locks**: Prevent several processes from accessing a shared resource concurrently. Mutexes and semaphores provide as examples.
- **Critical Portions**: Code portions that can only be executed by one thread at a time to ensure data integrity.
- **Monitors and Condition Variables**: Higher-level abstractions for managing shared data and signaling between threads.

Effective synchronization is critical in multi-threaded contexts to maintain dependability and consistency in concurrent processes.

7.5 Synchronization Diagram

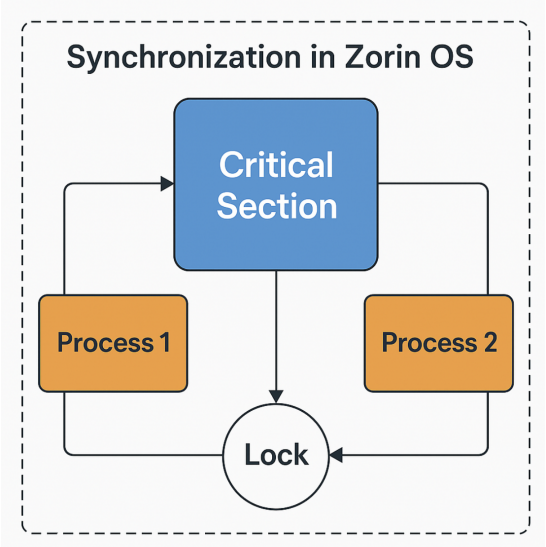


Figure 2: Synchronization within Zorin OS, focusing on processes, critical sections, and locks

8 Deadlock in Zorin OS

Deadlock is a critical issue in operating system design and can significantly affect system stability and performance. In Zorin OS, which is based on the Linux kernel, deadlock can occur when two or more processes are unable to proceed because each is waiting for a resource held by another. This section focuses on the actual implementation and observation of deadlock in Zorin OS, exploring how it can be detected, its impact, and methods of resolution.

8.1 What is Deadlock?

Deadlock occurs in systems that use multiple resources and processes. It is a situation where two or more processes are each waiting for a resource held by the other, creating a cycle of dependencies that prevents any of them from proceeding. The four necessary conditions for deadlock to occur are:

1. **Mutual Exclusion:** At least one resource must be held in a non-shareable mode, meaning only one process can use the resource at a time.
2. **Hold and Wait:** A process holding at least one resource is waiting for other resources held by other processes.
3. **No Preemption:** Resources cannot be preempted. A process must release resources voluntarily.

4. **Circular Wait:** A set of processes exists where each process is waiting for a resource held by the next process, forming a circular chain.

These four conditions must be met simultaneously for a deadlock to occur.

8.2 Deadlock Detection in Zorin OS

In Zorin OS (and other Linux-based systems), deadlock is not automatically detected by the kernel, but there are system monitoring tools that can help identify and diagnose deadlock conditions. These tools include:

- **ps aux:** The `ps aux` command lists all running processes along with their status. Processes in a "D" (uninterruptible sleep) state may be waiting for a resource and could indicate deadlock.
- `strace`: `strace` is used to trace system calls made by a process. This can help in identifying which system resources (such as files or memory) a process is waiting for.
- `top`: The `top` command provides real-time statistics about system processes, memory, and CPU usage. It helps in identifying processes that are stuck or consuming excessive resources.

By using these tools, we can observe the behavior of processes involved in a deadlock and identify the exact resources they are waiting for.

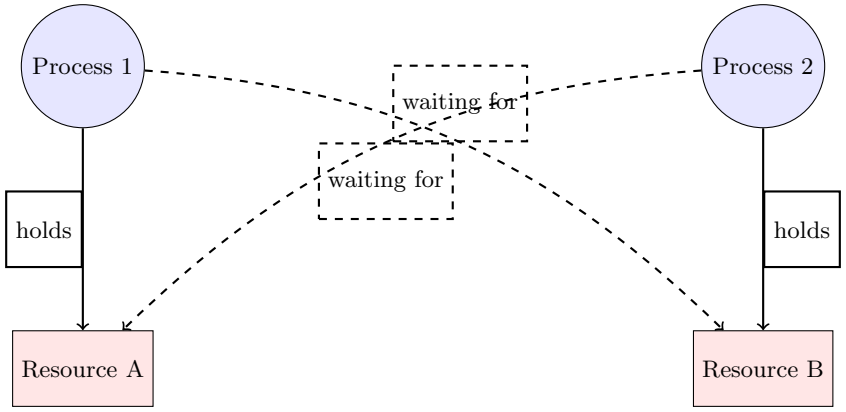


Figure 3: Deadlock situation in Zorin OS: Processes waiting on each otherś resource.

8.3 Simulating Deadlock in Zorin OS

To simulate a deadlock situation in Zorin OS, we can create two processes that each hold one resource and wait for the resource held by the other process. Hereś how we can do it:

1. Process 1 locks Resource A and waits for Resource B.
2. Process 2 locks Resource B and waits for Resource A.

This scenario creates a circular dependency and results in deadlock.

To simulate this deadlock using shell commands, we can use the following script:

```
#!/bin/bash

# Simulating Process 1
(
    echo "Process 1 holding Resource A"
    sleep 5  # Simulate holding Resource A
    echo "Process 1 waiting for Resource B"
    sleep 10  # Simulate waiting for Resource B
) &

# Simulating Process 2
(
```

```
    echo "Process 2 holding Resource B"
    sleep 5  # Simulate holding Resource B
    echo "Process 2 waiting for Resource A"
    sleep 10  # Simulate waiting for Resource A
) &
```

8.4 Detecting Deadlock Using `ps aux`

After running the script, we can use the `ps aux` command to monitor the processesśtates. If deadlock occurs, the output will show that both processes are in a "D" (uninterruptible sleep) state. Below is an example of what the output might look like:

USER	PID	STAT	COMMAND
user1	1234	D	bash script1.sh
user2	5678	D	bash script2.sh

Table 1: Deadlock Detected Using `ps aux`: Processes in "D" State (Uninterruptible Sleep).

8.5 Resolving Deadlock in Zorin OS

Once deadlock is detected, we need to resolve it. In Zorin OS, deadlock can be addressed using the following methods:

- **Manual Termination:** The simplest method to resolve deadlock is to manually terminate one or more processes. This can be done using the `kill` or `killall` commands.

  ```
  kill -9 <PID>
  ```

  This command forcibly terminates the process with the specified Process ID (PID), allowing resources to be released and resolving the deadlock.

- **Resource Preemption:** In more complex scenarios, the kernel may use resource preemption to take resources from one process and allocate them to another. This method is not commonly used in user-space programs but may occur in certain system-level resource management tasks.

- **Timeouts:** Implementing timeouts for processes waiting on resources can also help in avoiding deadlock. If a process waits too long for a resource, it can be forced to release its current resources or be terminated.

8.6 Deadlock Prevention Techniques in Zorin OS

While Linux-based systems like Zorin OS do not automatically prevent deadlock, developers can take certain precautions to reduce the likelihood of deadlock:

- **Lock Ordering:** Ensure that all processes acquire resources in a fixed order, preventing circular waits.
- **Avoid Hold and Wait:** Design systems where processes request all the resources they need at once, rather than holding some resources while waiting for others.
- **Timeout Mechanisms:** Implement timeouts that prevent processes from waiting indefinitely for resources, reducing the chances of deadlock.

By applying these practices, developers can minimize the risk of deadlock occurring in Zorin OS.

8.7 Overall Concept

Deadlock is a critical issue in operating systems like Zorin OS. By understanding the conditions that lead to deadlock, how it can be detected, and the methods for resolution, we can better manage system resources and ensure system stability. Using tools like `ps aux`, `strace`, and `top`, we can monitor processes and identify deadlock situations. Once detected, deadlock can be resolved through manual termination, resource preemption, or timeouts. Furthermore, developers can use deadlock prevention techniques to reduce the likelihood of deadlock in their applications.

9 Legal and Ethical Issues

9.1 Open-Source Licensing

9.1.1 GNU General Public License v2.0

Zorin OS components are primarily licensed under the GNU General Public License version 2.0 (GPL v2.0), a strong copyleft license that ensures the software remains free and open. The technical and legal implications of this licensing choice include:

1. **Freedom to Use:** Users have the freedom to run the program for any purpose.
2. **Freedom to Study and Modify:** Users have access to the source code and can study how the program works and adapt it to their needs.
3. **Freedom to Redistribute:** Users can redistribute copies of the software.
4. **Freedom to Improve and Share Improvements:** Users can improve the program and release their improvements to the public.
5. **Copyleft Requirement:** Any derivative works must also be distributed under the same license terms, ensuring that modifications remain open source.

9.1.2 Licensing Compliance Mechanisms

Zorin OS ensures GPL compliance through several technical mechanisms:

1. **Source Code Availability:** The GPL v2.0 requires that source code be made available to anyone who receives the binary form of the software. Zorin OS complies with this requirement by providing access to its source code through multiple repositories:
   - `https://github.com/ZorinOS/`
   - `https://launchpad.net/žorinos/+archive/ubuntu/stable`
   - `https://zorin.com/about/#source-code`
2. **License Notices in Source Files:**

```
/*
 * Copyright (C) [year] Zorin OS Technologies Ltd.
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * ... additional GPL text ...
 */
```

3. **Build System:** The build system is designed to track license information and ensure that appropriate notices are included in the distribution.
4. **Source Code Organization:** The source code is organized into several repositories:
   - Zorin OS Stable: Contains the core components of the stable Zorin OS releases.
   - Zorin OS Patches: Includes patches and updates applied to the base system.
   - Zorin OS Drivers: Contains hardware drivers specific to Zorin OS.
   - Zorin OS Apps: Includes Zorin-specific applications and utilities.
   - Ubuntu (upstream): Links to the upstream Ubuntu repositories that form the base of Zorin OS.
   - Zorin Connect Android app: Source code for the Android companion app.

9.1.3 Paid vs. Free Versions

Zorin OS offers both free and paid versions (Zorin OS Ultimate). The paid version includes additional applications and features but still complies with GPL requirements by making the source code available.

This dual-model approach is permitted under the GPL, which allows for charging a fee for distributing the software or providing additional services. The GPL does not prohibit commercial use or selling the software; it only requires that the source code be made available to recipients of the software.

The Zorin OS Ultimate edition helps support the development of Zorin OS while still adhering to open-source principles. Users who purchase the Ultimate edition receive additional software, support, and features, but the core operating system remains open source and compliant with the GPL.

9.2 Security Architecture

9.2.1 AppArmor Mandatory Access Control

AppArmor is a kernel security module that restricts programs' capabilities through per-program profiles. In Zorin OS:

- AppArmor is preinstalled and enabled by default
- It is specifically configured to protect WINE applications (Zorin App Support) from potential Windows malware
- Profiles define what files a program can access and what operations it can perform
- Technical implementation involves path-based access controls enforced at the kernel level
- Status can be checked using `sudo apparmor_status`

AppArmor in Zorin OS 17.3 implements Mandatory Access Control through:

1. Profiles stored in `/etc/apparmor.d/`
2. Path-based access controls enforced at the kernel level
3. Mediation of file operations, network access, and capabilities

Example AppArmor profile snippet:

```
# Example AppArmor profile for WINE
/usr/bin/wine {
  #include <abstractions/base>
  #include <abstractions/user-tmp>

  /lib/** r,
  /usr/lib/** r,

  owner @{HOME}/.wine/** rwk,

  deny /etc/shadow r,
  deny /etc/passwd w,

  # ... more rules ...
}
```

9.2.2 Uncomplicated Firewall (UFW)

UFW provides a simplified interface to the netfilter packet filtering system:

- Implemented as a front-end to iptables/nftables
- Allows configuration of inbound and outbound traffic rules
- Uses a default-deny policy for incoming connections
- Includes application profiles for common services

The Uncomplicated Firewall in Zorin OS provides a frontend to netfilter/iptables:

```
# Example UFW rules that demonstrate the underlying iptables implementation
sudo ufw allow 22/tcp
# Translates to iptables rules like:
# iptables -A INPUT -p tcp --dport 22 -j ACCEPT
# iptables -A INPUT -p tcp --dport 22 -m state --state NEW -j ACCEPT
```

9.2.3 Full-Disk Encryption

Zorin OS supports LUKS (Linux Unified Key Setup) encryption:

- Implemented through `dm-crypt` at the kernel level
- Uses AES-XTS encryption with 512-bit keys by default
- Protects data at rest from unauthorized physical access
- Can be configured during installation or post-installation

Technical Specifications:

- Uses dm-crypt at the kernel level
- AES-XTS encryption with 512-bit keys
- SHA-256 for key derivation
- LUKS2 format with Argon2id key derivation function

9.2.4 Secure Boot Support

Zorin OS supports UEFI Secure Boot:

- Verifies that boot components are signed with trusted keys
- Prevents unauthorized boot code execution
- Uses shim bootloader to bridge Microsoft and Linux signing keys

9.3 Privacy Protection Mechanisms

9.3.1 Privacy Controls in Settings

The Settings application includes a dedicated Privacy section with controls for:

- Location Services: Enables/disables system-wide location access
- File History: Controls storage of recently used files
- Screen Lock: Configures automatic screen locking
- Usage & History: Manages collection of usage statistics

These controls provide users with granular control over their privacy settings, allowing them to balance convenience with privacy.

9.3.2 Application Permissions

For Flatpak and Snap applications, Zorin OS provides granular permission controls:

- Sandboxing configuration
- File system access restrictions
- Device access permissions
- Network access controls
- This permission system is similar to those found in mobile operating systems, allowing users to control what resources applications can access.

9.3.3 Data Collection Policy

Zorin OS implements a minimal data collection policy:

- No personal data is collected from user devices
- Anonymous system info may be collected during installation if the user opts in
- All data collection is opt-in

9.3.4 Transparency Measures

Zorin OS maintains a warrant canary in its privacy policy:

- Indicates no backdoors have been installed
- Confirms no FISA or court orders have been received
- Promotes transparency regarding government requests

9.4 Windows Compatibility Security Considerations

9.4.1 WINE Security Architecture

To mitigate risks from Windows applications:

1. AppArmor Confinement: WINE applications are confined using AppArmor profiles that restrict their access to system resources.
2. Filesystem Isolation: Windows applications run through WINE have limited access to the host filesystem, reducing the potential impact of malware.
3. No System Integration: Windows applications cannot modify system files or settings outside their confined environment.

9.4.2 Malware Protection Mechanisms

Zorin OS provides several tools to protect against malware in Windows applications::

1. ClamAV Integration: While Linux systems are generally less susceptible to malware than Windows, Zorin OS can use ClamAV, an open-source antivirus engine, to scan for and eliminate malware and viruses.
2. Regular Updates: Security updates for WINE and related components help address vulnerabilities that could be exploited by malware.
3. Alternative Native Applications: Zorin OS encourages the use of native Linux applications as alternatives to Windows software, which generally provides better security and integration with the system.
4. These measures help mitigate the security risks associated with running Windows applications while still providing the convenience of Windows compatibility.

9.5 Security Implementation Analysis

The security implementation in Zorin OS balances protection with usability:

1. **Defense in Depth:** Multiple security layers (AppArmor, UFW, encryption) provide defense in depth against various threats. This layered approach ensures that a breach of one security mechanism does not necessarily compromise the entire system.
2. **Usability Focus:** Security features are implemented with usability in mind, making them accessible to non-technical users. Tools like UFW provide simple interfaces for complex security configurations, lowering the barrier to implementing good security practices.
3. **Regular Updates:** Integration with Ubuntu's security update system ensures timely patching of vulnerabilities. Security updates are delivered through the same package management system used for regular updates, simplifying the maintenance process.

4. **Minimal Attack Surface:** The default installation minimizes the attack surface by enabling only necessary services. This reduces the potential entry points for attackers.

9.6 some Limitations:

- Performance Impact: Security mechanisms like AppArmor and full-disk encryption can introduce performance overhead. This impact is generally minimal but may be noticeable on older or resource-constrained hardware.
- Complexity vs. Security Tradeoffs: Some security features are simplified to improve usability, which may reduce their effectiveness in certain scenarios. Advanced users may need to configure additional security measures for high-security environments.
- Dependency on Upstream: As a derivative of Ubuntu, Zorin OS relies on upstream security patches. While this generally ensures good security coverage, it also means that Zorin OS is dependent on Ubuntu's security response times.

10 Memory Management

10.1 Paging and Virtual Memory

10.1.1 What is Virtual Memory?

Virtual memory allows an operating system (OS) to present each program with the illusion of having a large, continuous block of memory, even if the physical RAM is small. The OS uses disk space as an extension of RAM, moving inactive programs to a swap area on the disk.

10.1.2 What is Paging?

Paging divides both physical and virtual memory into fixed-size blocks:

- **Pages** - blocks of virtual memory
- **Frames** - blocks of physical memory

When data required by a program is not in RAM (a page fault), the OS:

1. Pauses the program
2. Loads the required page from disk
3. Updates the page table
4. Restarts the program

Paging allows programs larger than physical RAM to run efficiently.

10.2 Memory Allocation Techniques

Contiguous Allocation Allocates memory in one large block:

- Simple but leads to external fragmentation
- Inflexible for smaller memory areas

Paging

- Memory is divided into fixed-size pages
- Eliminates external fragmentation
- Slightly complex but effective

Partitioning Divides memory logically into code, data, and stack segments:

- Each segment can grow or shrink independently
- Easier to understand and debug

Dynamic Allocation

- **First Fit:** Allocates the first block large enough
- **Best Fit:** Allocates the smallest suitable block
- **Worst Fit:** Allocates the largest block (rarely used)

Zorin OS Strategy Zorin OS uses:

- Paging
- Best Fit dynamic allocation
- Demand-paging to reduce memory usage

10.3 Swapping

Swapping involves temporarily moving processes from RAM to disk to free up space. In Zorin OS, this is achieved using a swap partition or file, especially under heavy multitasking or limited RAM.

10.4 Page Replacement Algorithms

When RAM is full, these algorithms decide which page to replace:

- **FIFO:** Removes the oldest page
- **LRU:** Removes the least recently used page
- **Optimal:** Removes the page not needed for the longest time (theoretical)

Linux-based systems like Zorin OS often use LRU variations.

11 File Management

11.1 File System Organization

The OS manages file creation, storage, access, and security, much like a virtual library. It tracks:

- File locations
- User access
- Metadata (creation time, type, etc.)

11.1.1 File System

Defines how data is stored and organized on devices. It includes:

- File and directory names
- Directory structure
- Disk data locations
- Metadata storage

Files Contain data (text, images, executables, etc.), and metadata like:

- Name and extension (e.g., `report.txt`)
- Size, timestamps

Directories Contain files and other directories.

Inodes (Linux systems) Store metadata excluding filename:

- File size, ownership
- Permissions, timestamps
- Data block pointers

11.1.2 Types of Files

- Ordinary Files: text, video, executables
- Directories
- Symbolic Links: references to other files
- Special Files: represent devices (e.g., `/dev/sda`)

Example: Zorin OS Uses the `ext4` file system supporting:

- Journaling
- Permissions and Ownership
- Large partitions and files
- Inode-based metadata

11.2 Permissions and Access Control

User Roles

- **User** (owner)
- **Group**
- **Others**

Permission Format Example: `-rwxr-xr--`

- `rwx` - read, write, execute (owner)
- `r-x` - read, execute (group)
- `r--` - read only (others)

Importance of Permissions Prevents:

- Unauthorized deletion of system files
- Malware modifications
- Unauthorized data access

11.3 Links

Hard Links Point directly to the inode. Deleting one does not remove the file if others exist.

Symbolic Links (Symlinks) Point to the path of another file. Break if the original file is deleted.

11.4 Caching and Buffering

- **Disk Caching:** Recently accessed data is stored in RAM
- **Write Buffering:** Delays writes to batch operations

11.5 Security and Encryption

Linux offers:

- File-level permissions
- ACLs (Access Control Lists)
- Full-disk encryption (e.g., LUKS)
- Per-file encryption (e.g., GnuPG, eCryptFS)

12 Operating System Implementation

12.1 Video Implementation Link

https://drive.google.com/file/d/1iv_1_I3X3xi3AcG0P3xdV-26vHa2r6KO/view?usp=sharing

12.2 Set Up a Virtual Machine

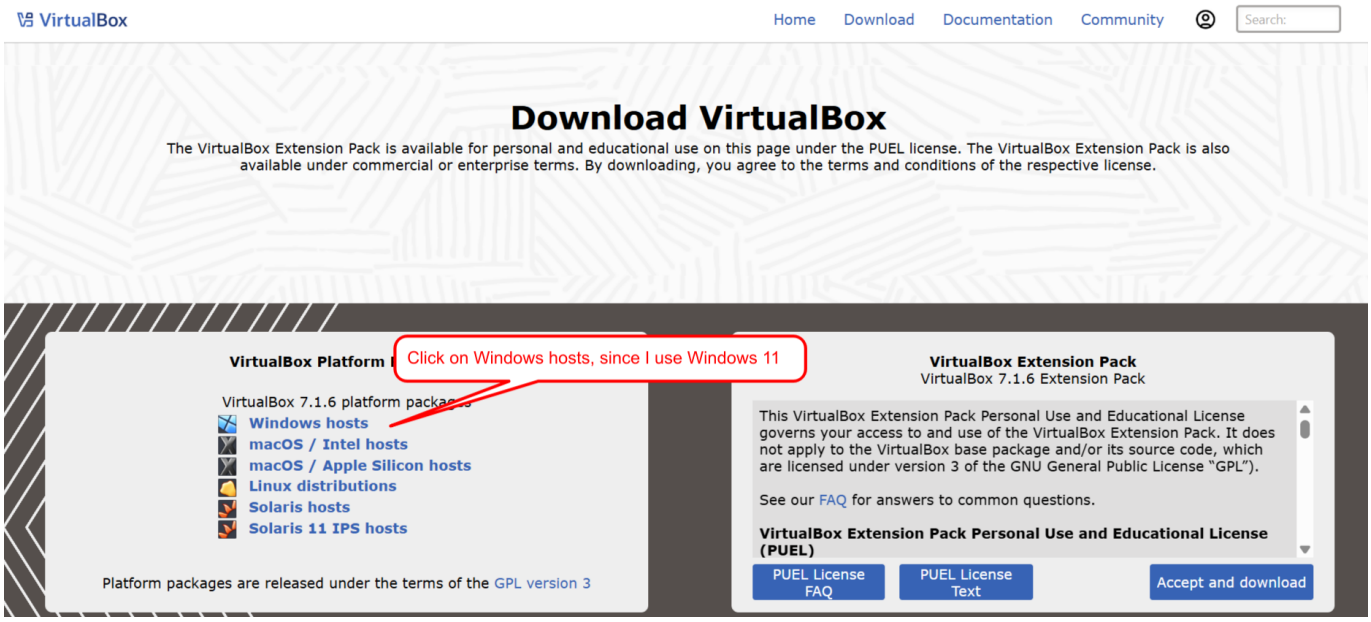


Figure 4: Choose the download package and version availability (7.1.6) based on your OS

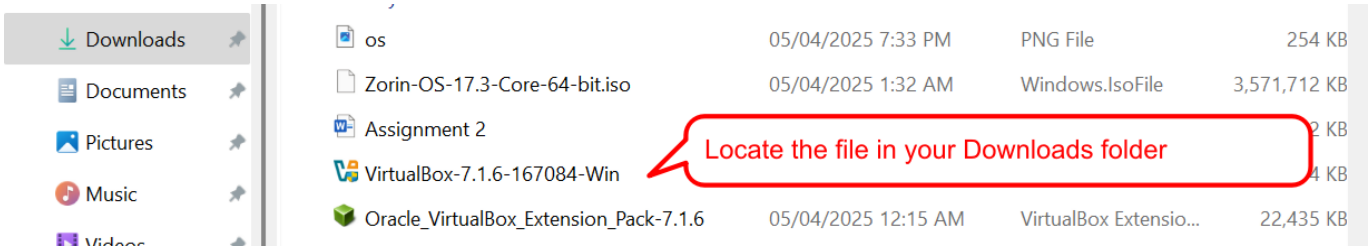


Figure 5: Click on the file

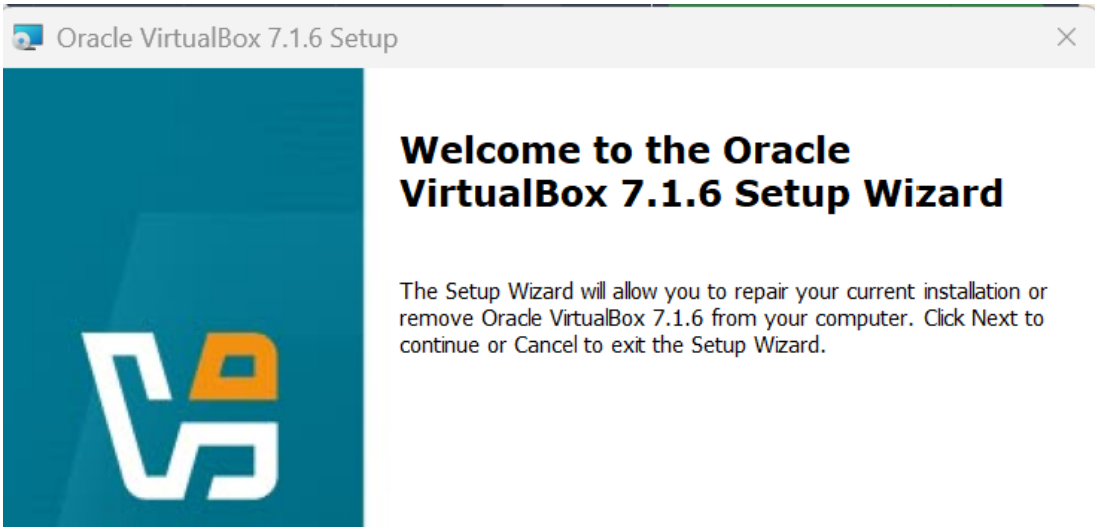


Figure 6: Set up the Oracle VirtualBox

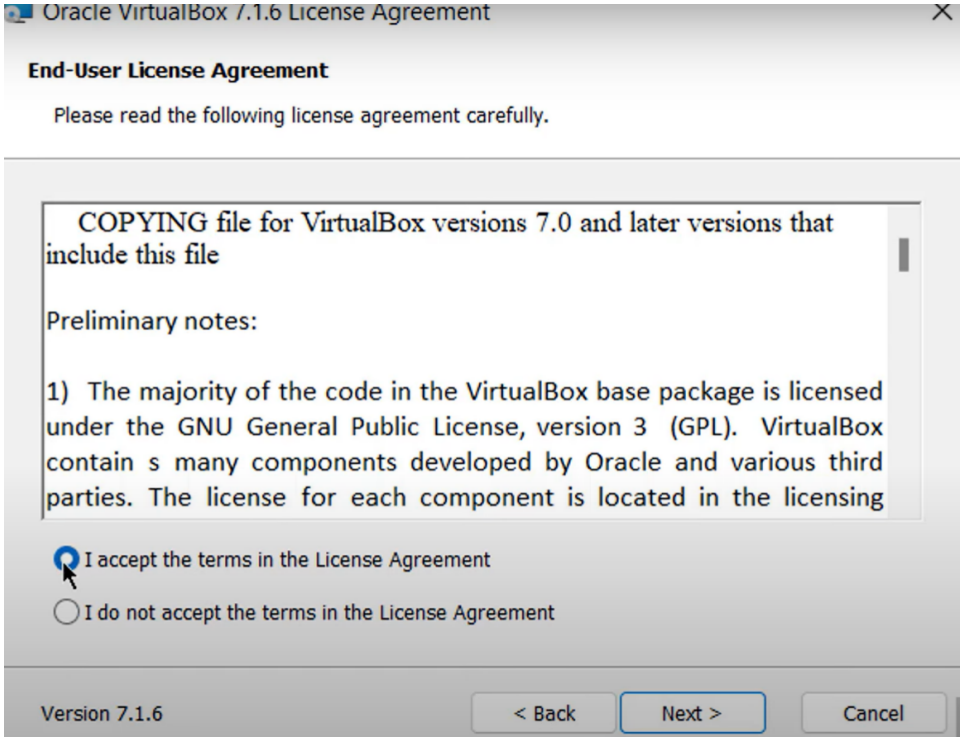


Figure 7: Accept the License Agreement

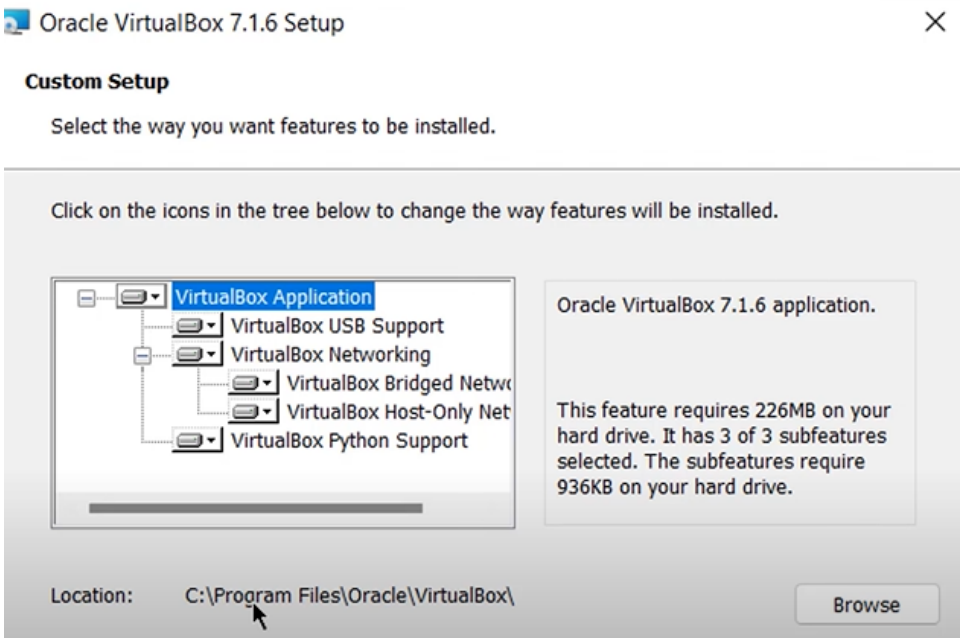


Figure 8: Keep Default Settings

Figure 9: Proceed with Installation

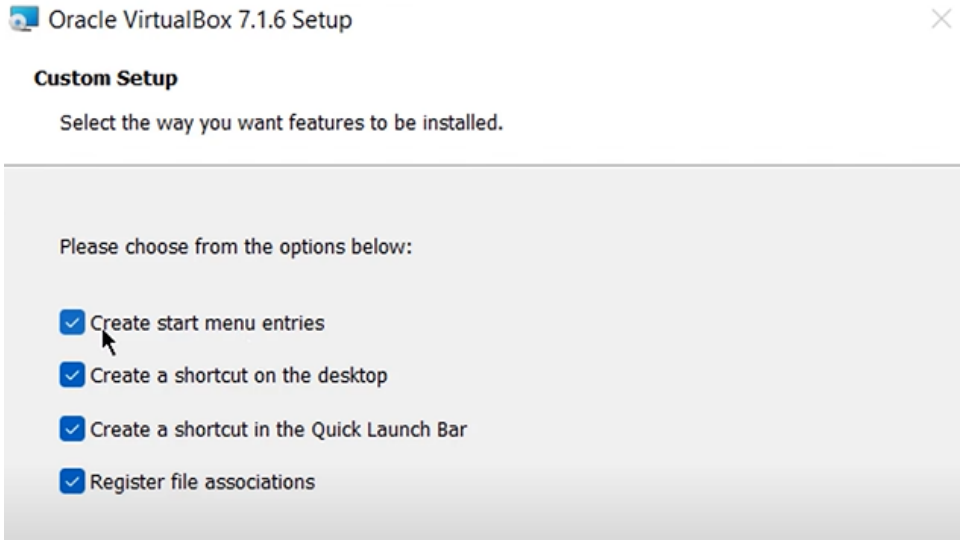


Figure 10: Choose based on your preference

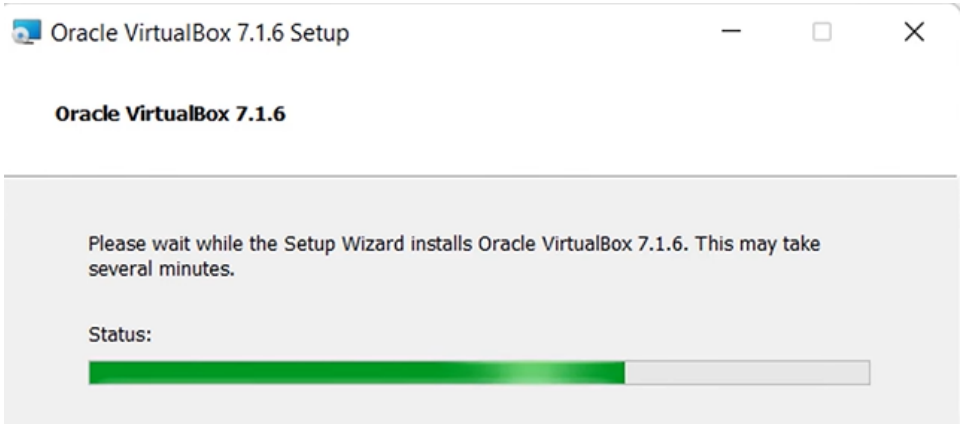


Figure 11: Wait for it to install

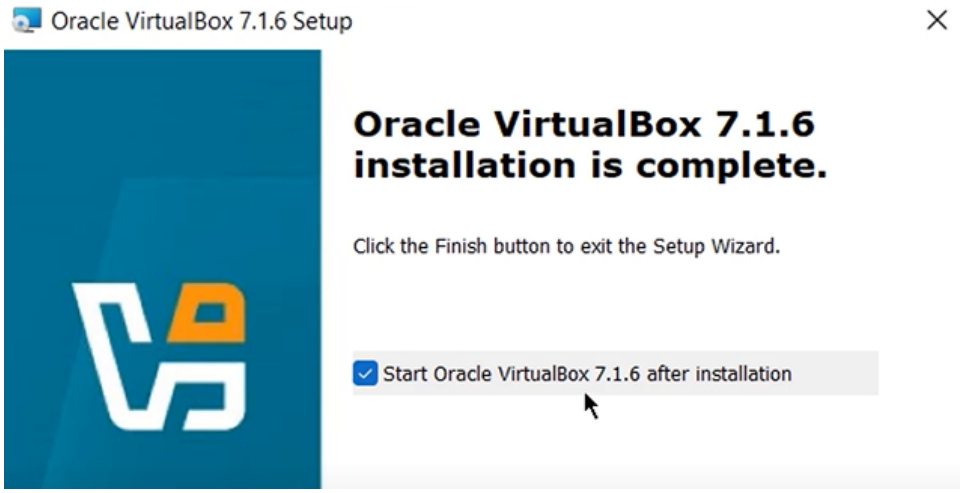


Figure 12: Once it is finished, run it

12.3 Install the OS (Zorin)

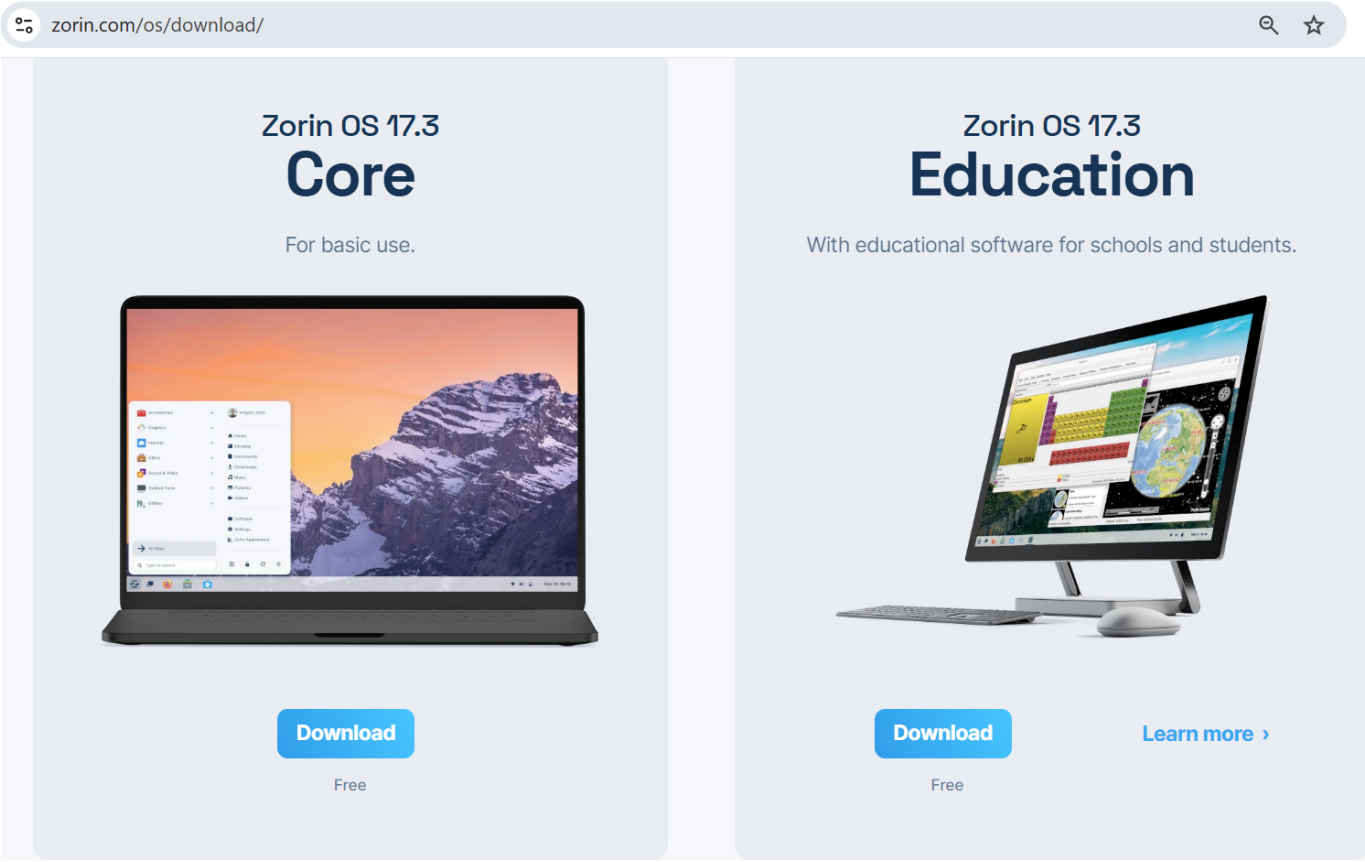


Figure 13: Install the core version from the official Zorin website

The latest Core version (17.3) instead of Education was used because the latter does not offer significant benefits for processes & thread scheduling, memory management or system calls.

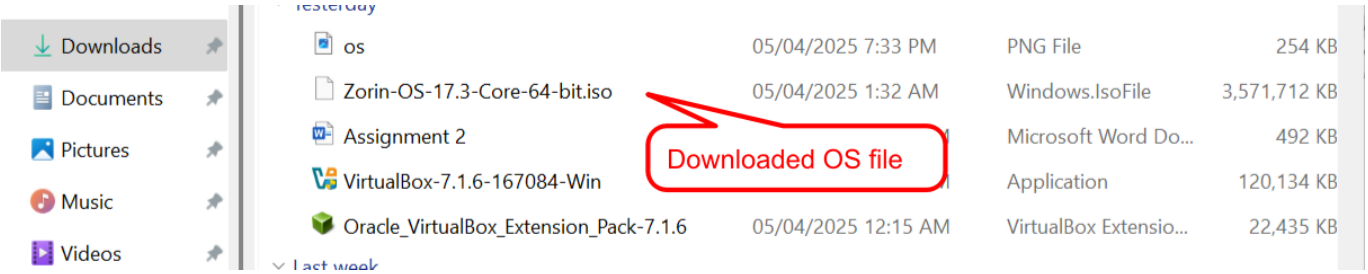


Figure 14: Locate the `.iso` file in your downloads folder

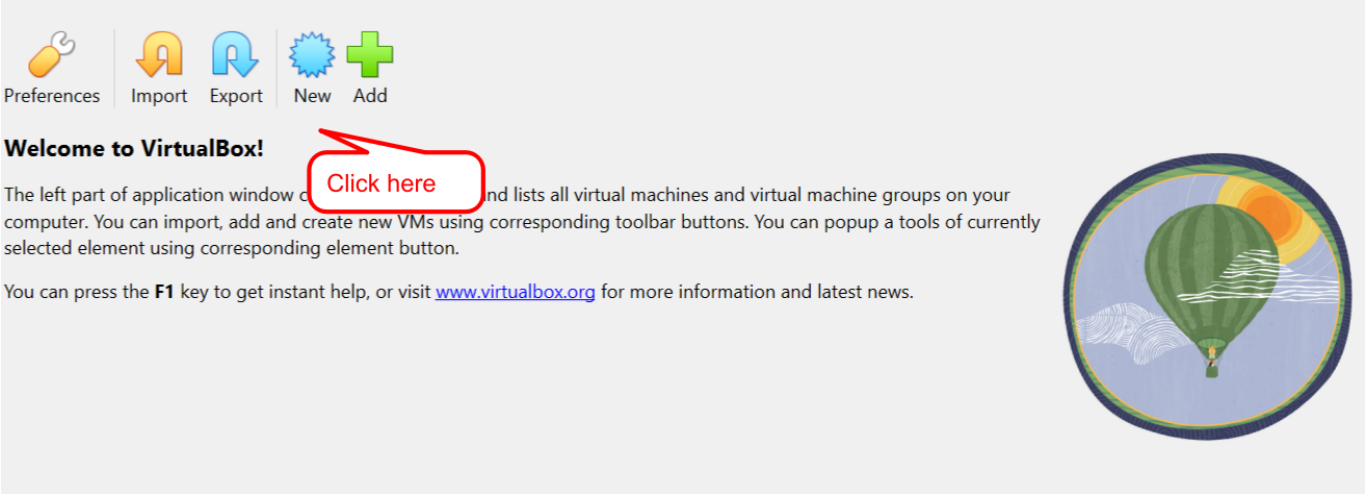


Figure 15: Return to Oracle VirtualBox Manager, create a new Virtual Machine (for Zorin OS) by clicking on new

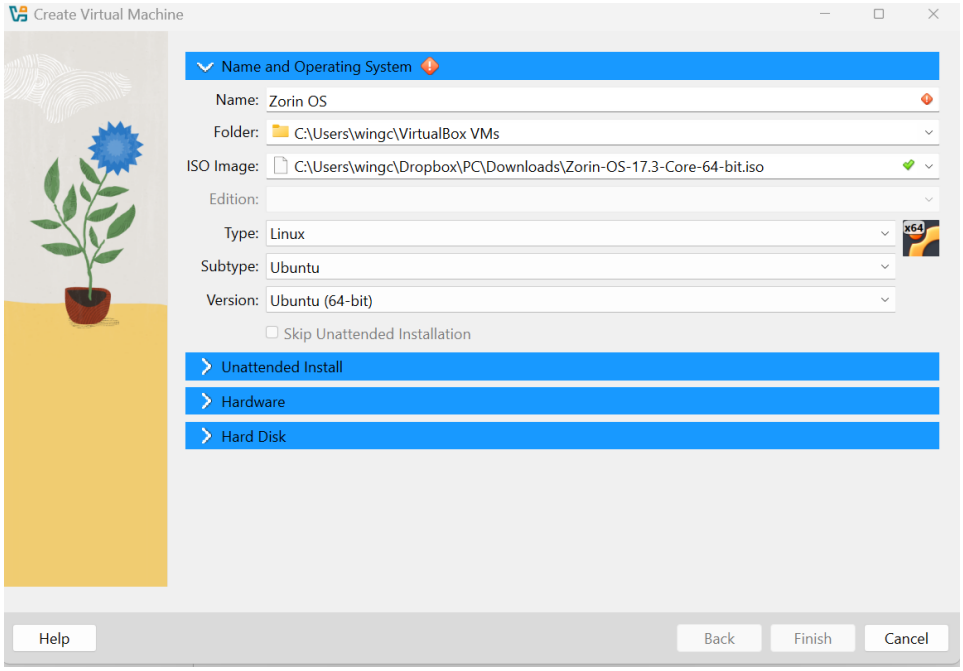


Figure 16: Enter the name as *Zorin OS* and select the `iso` image from the downloads folder, will automatically detect it as Ubuntu (64-bit)

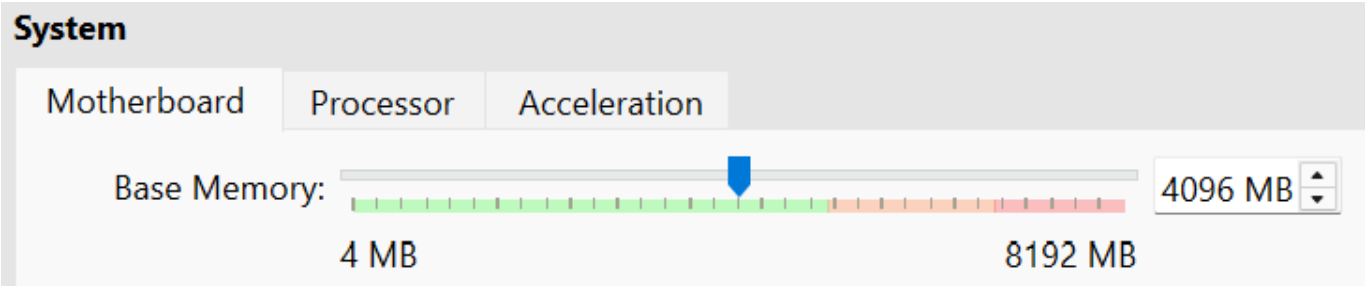


Figure 17: Set the Base Memory to 4GB, since my Laptop has only 8GB RAM, so ensures smooth performance for both VM and laptop.

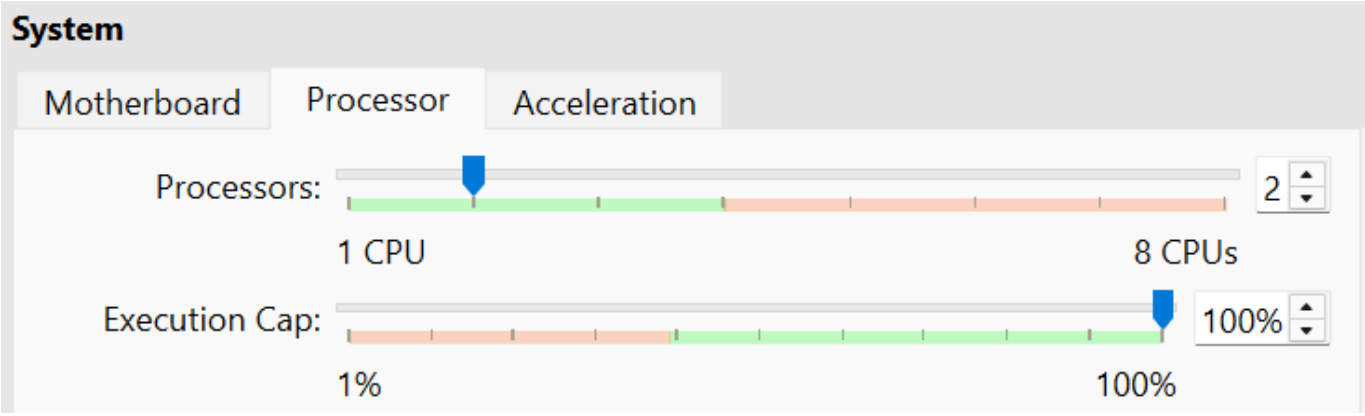


Figure 18: Set the processors to two, so it balances the performance without slowing down the laptop.

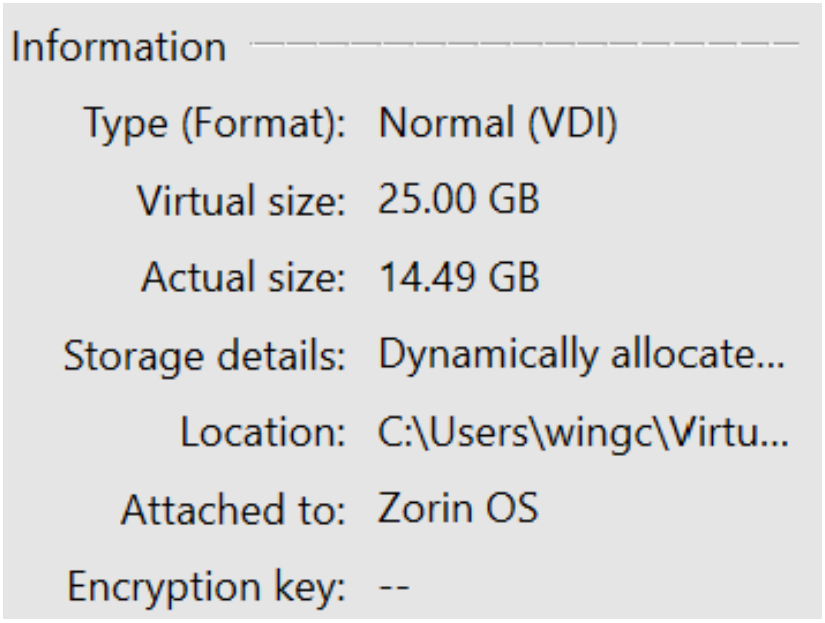


Figure 19: Storage set as 25GB, because my available storage is 40GB, resulting in free 20GB so I won't run out of space. Zorin requires minimum of 20GB, for flexibility, choose 25GB.

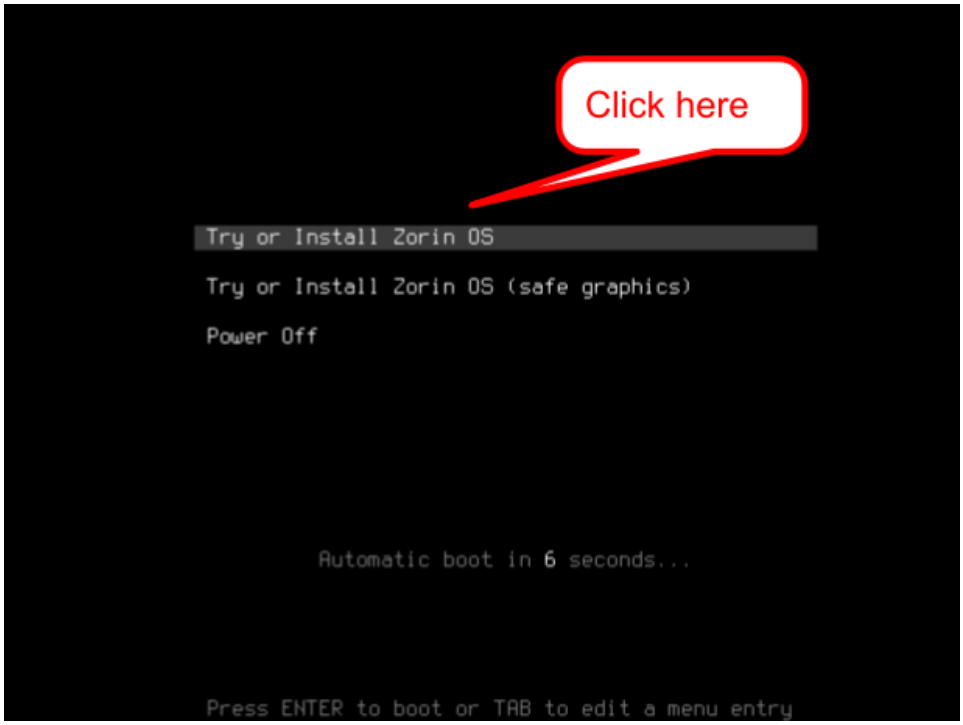


Figure 20: After you have finished your hardware specifications, you are led to this menu

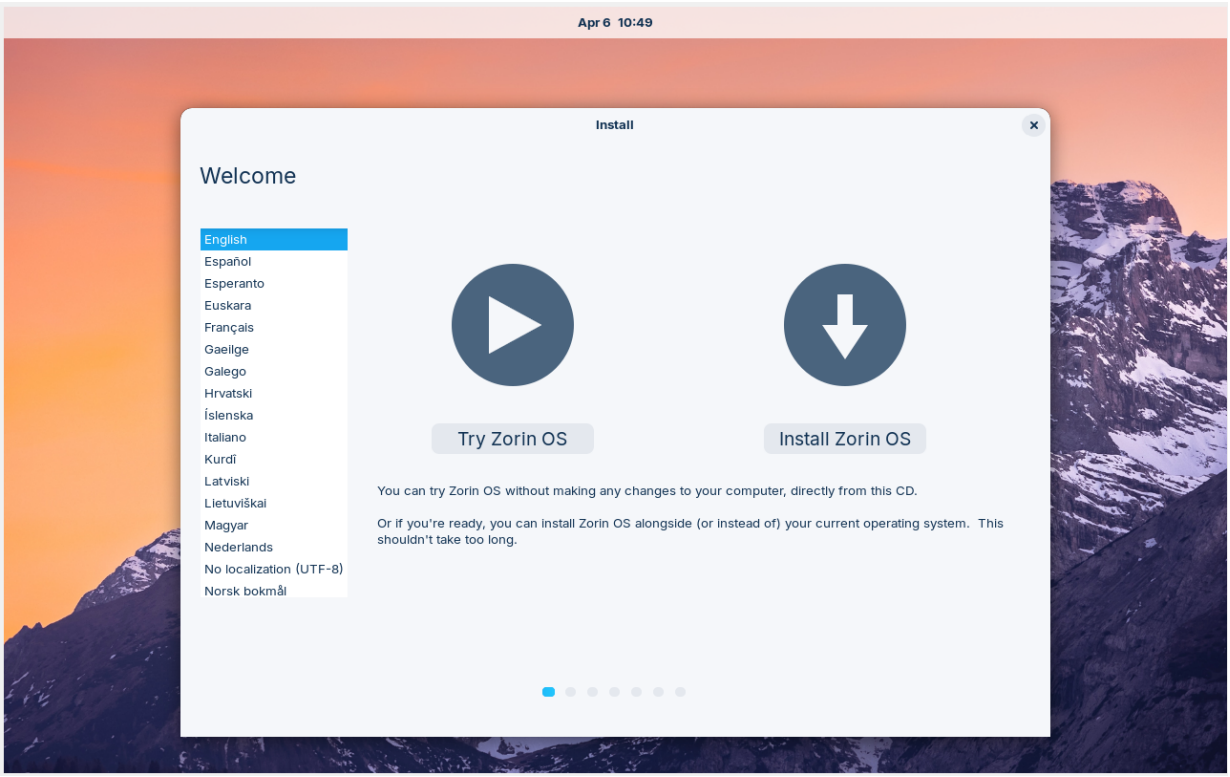


Figure 21: Since I needed important analysis for the project, I have installed it on my laptop rather than simply trying it.

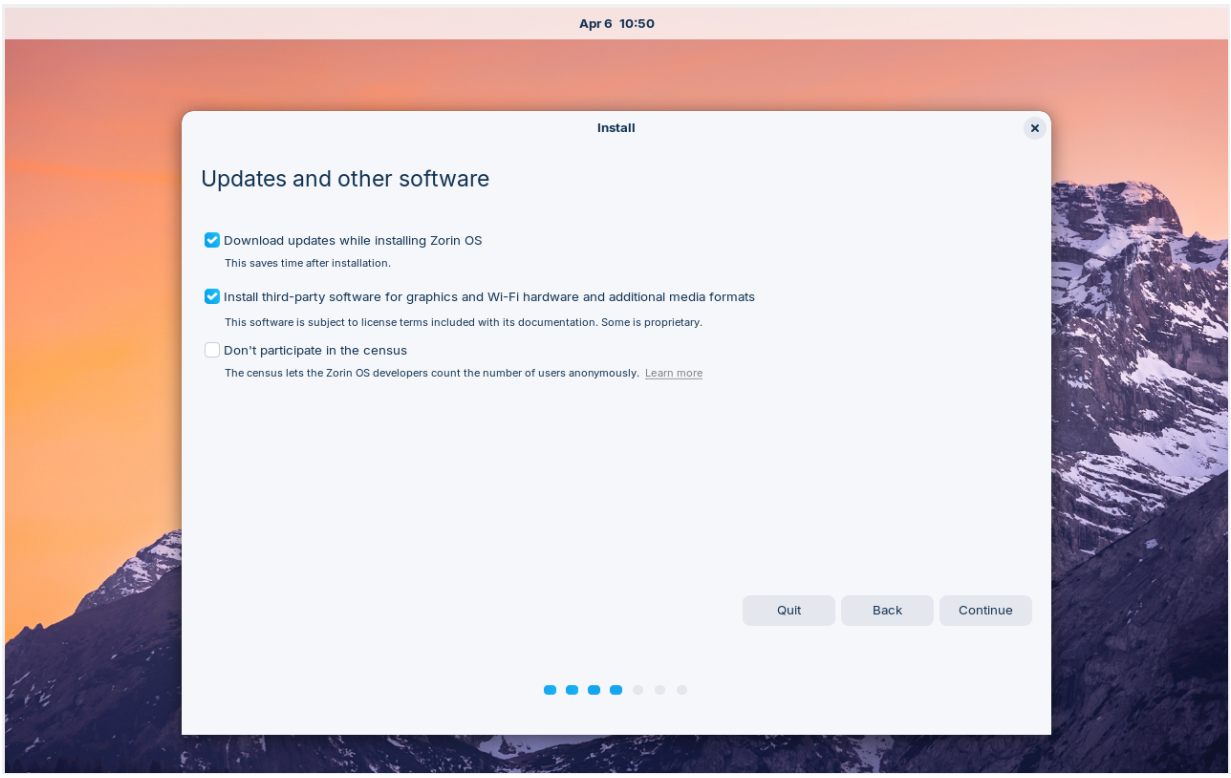


Figure 22: Choose based on your preference

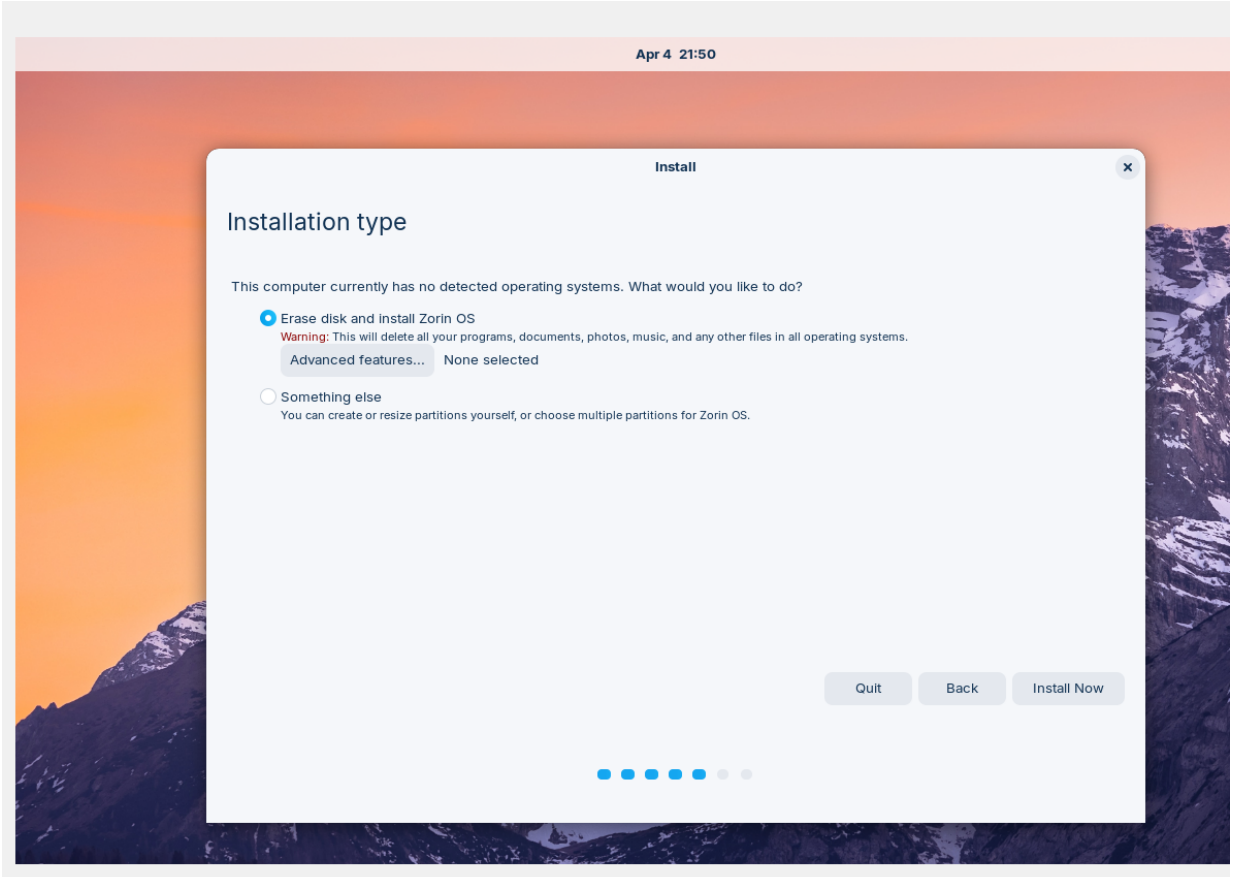


Figure 23: Choose Erase Disk and Install Zorin. Note that this will cause no changes to your actual Windows OS because this in on a Virtual Machine.

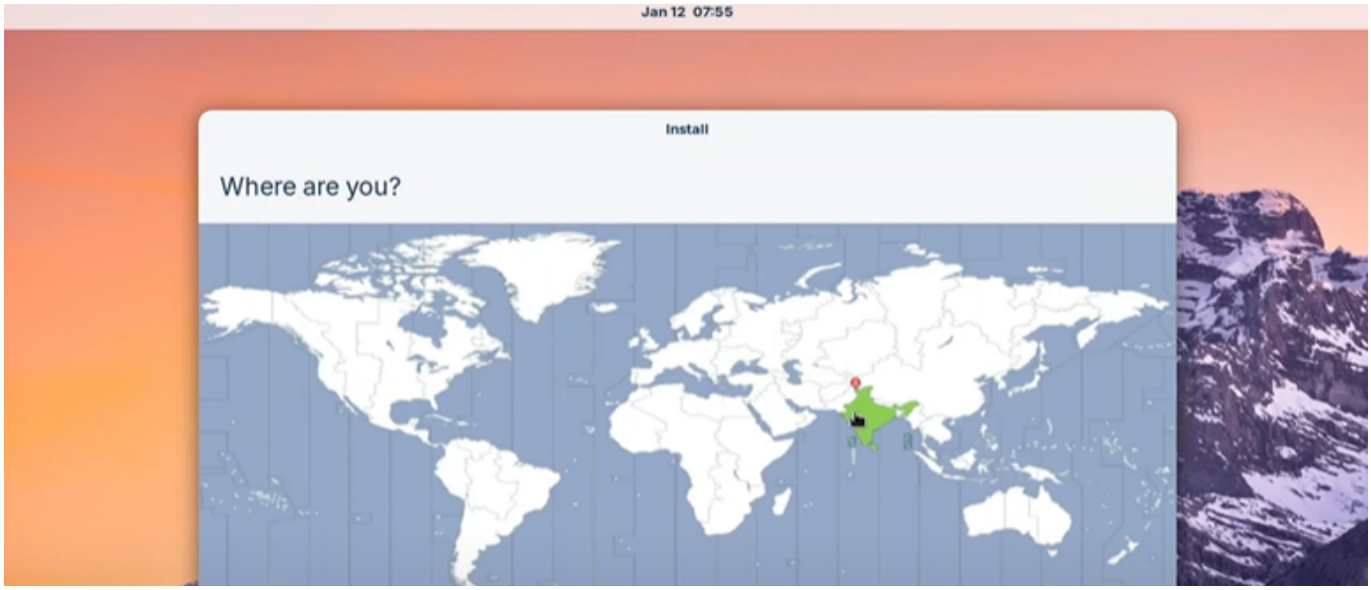


Figure 24: Enter your location

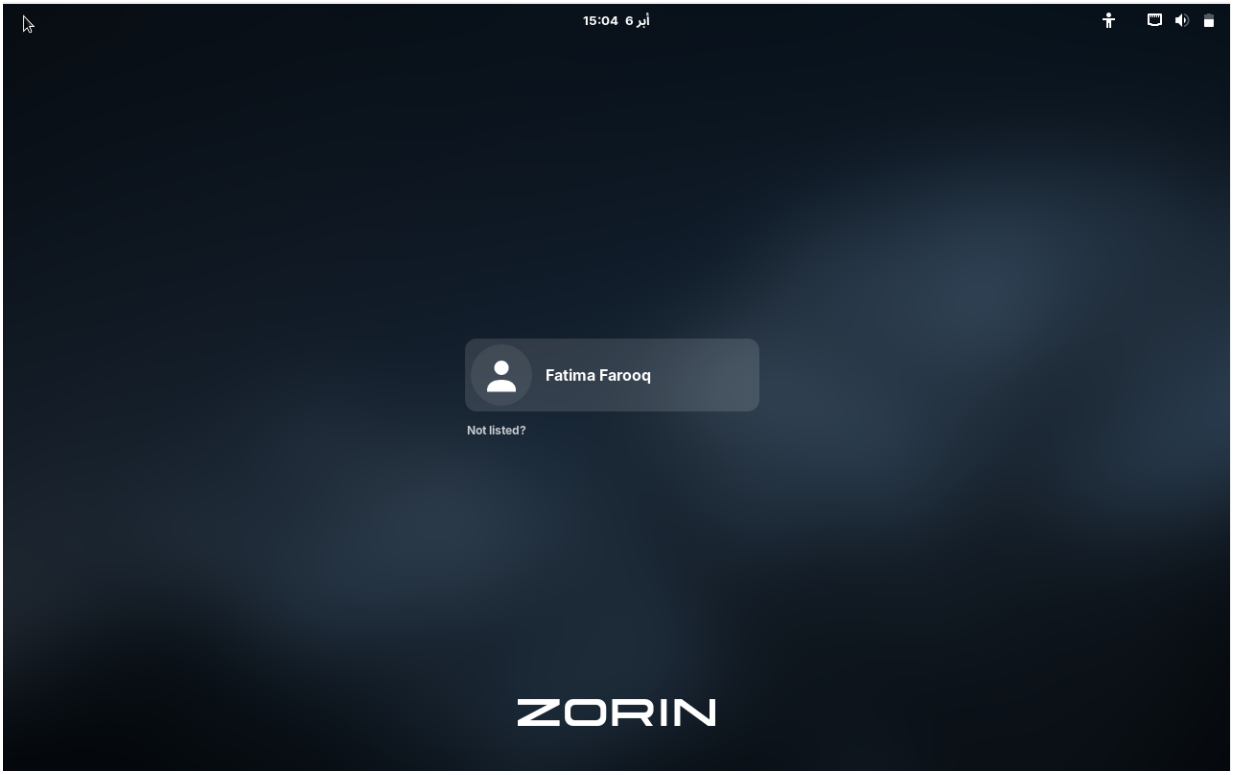


Figure 25: Finally login to your system

12.4 Install System Monitoring Tools

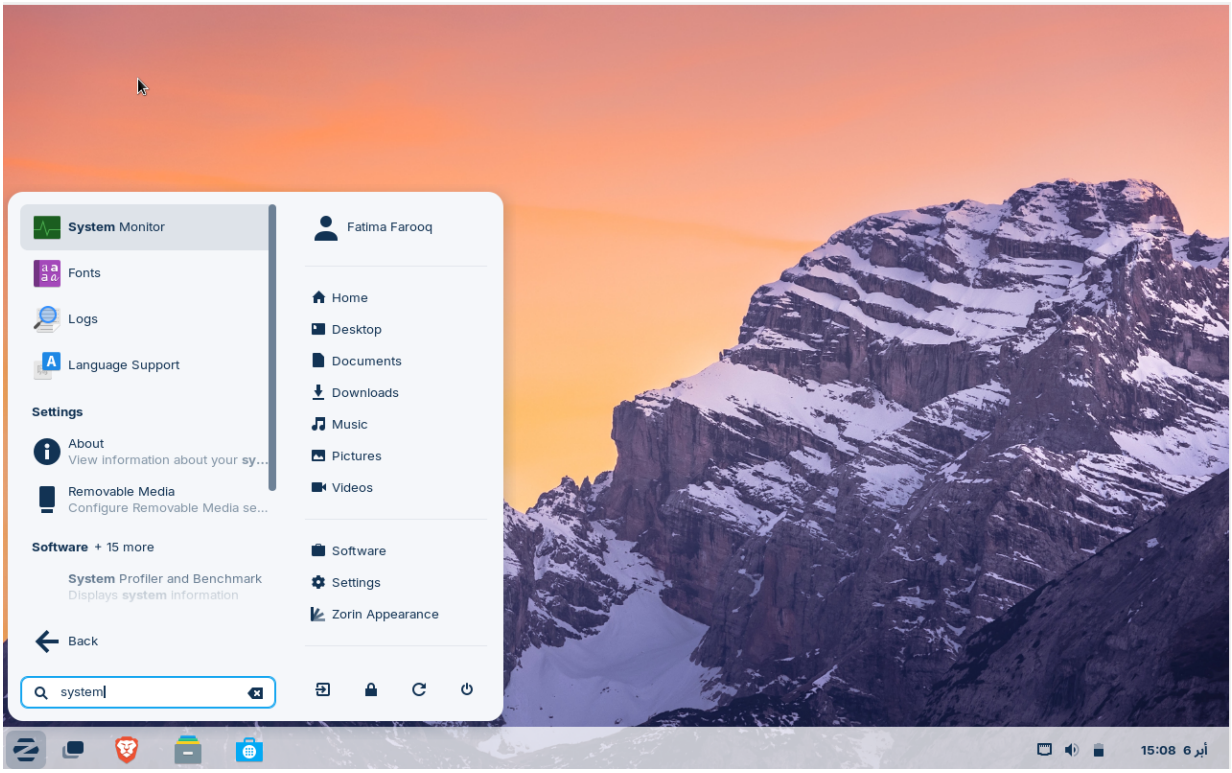


Figure 26: System Monitoring Tools are by default, installed

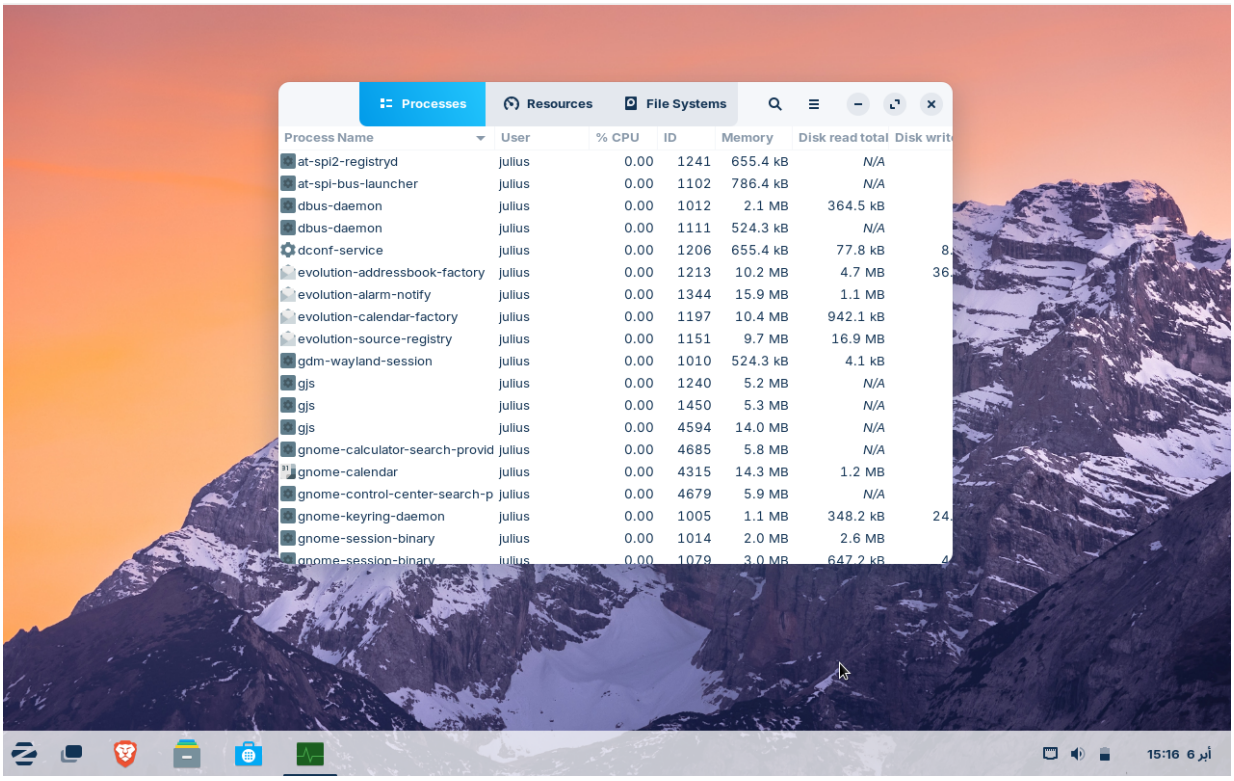


Figure 27: Opens the current processes running

12.5 Analyze Process and Thread Management

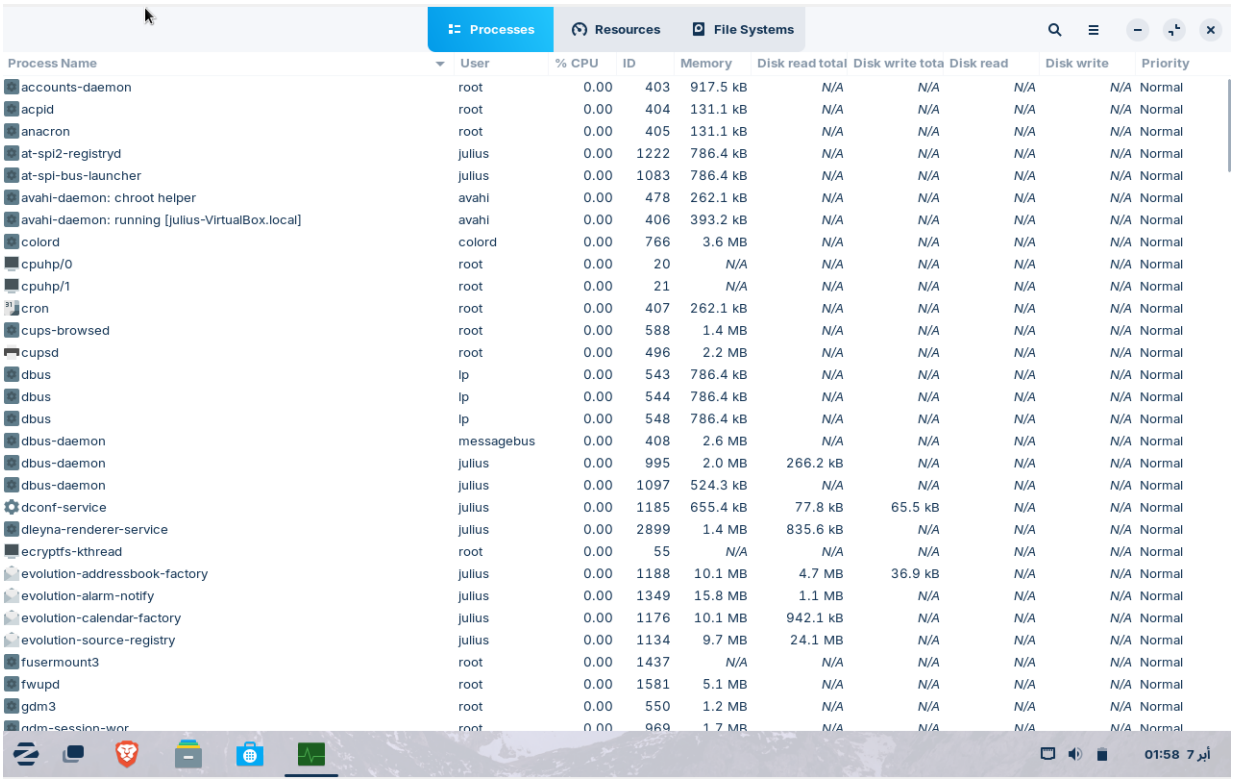


Figure 28: The list of running processes

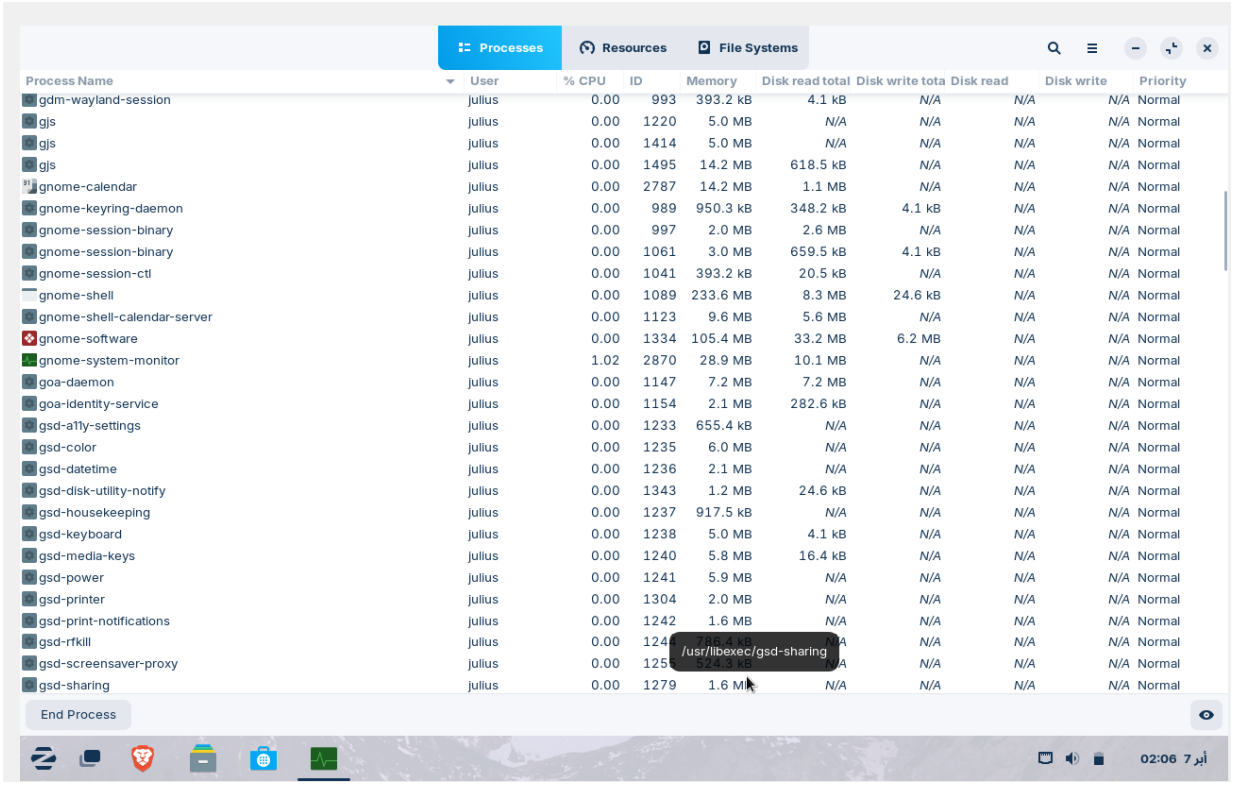


Figure 29: The list of running processes

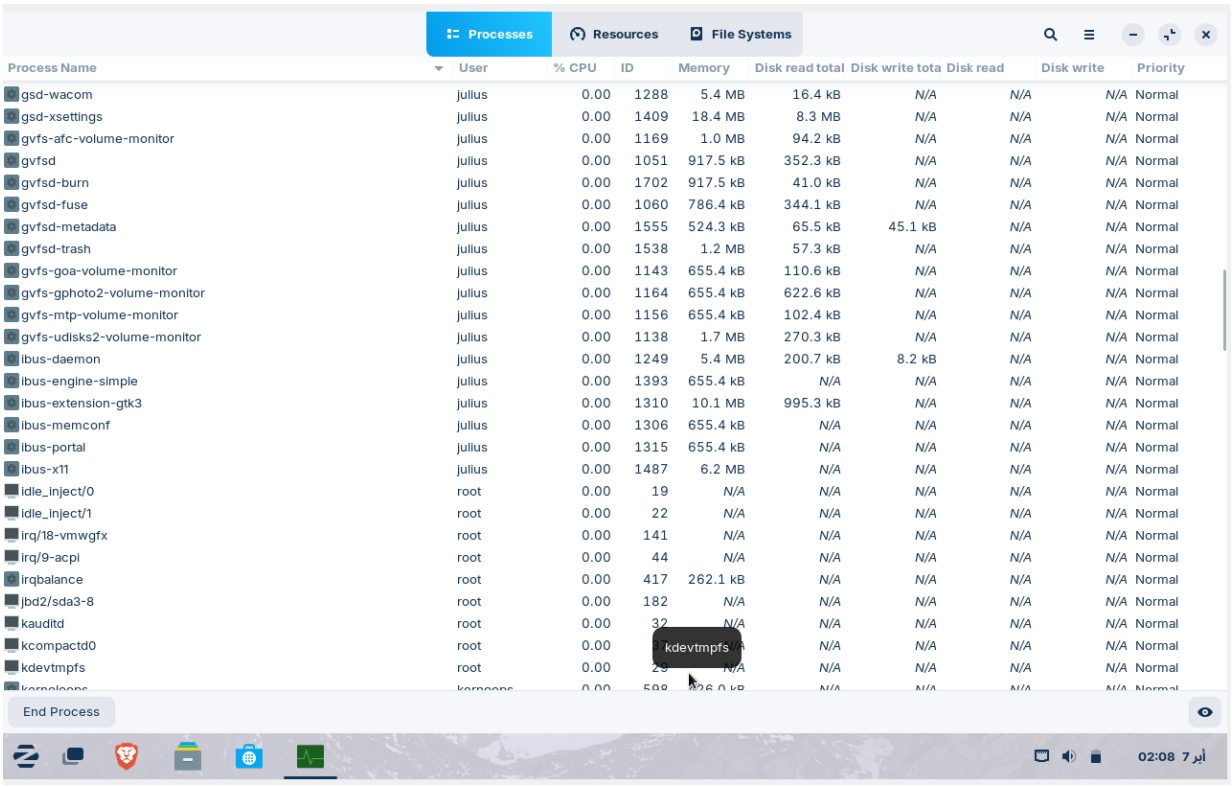


Figure 30: The list of running processes

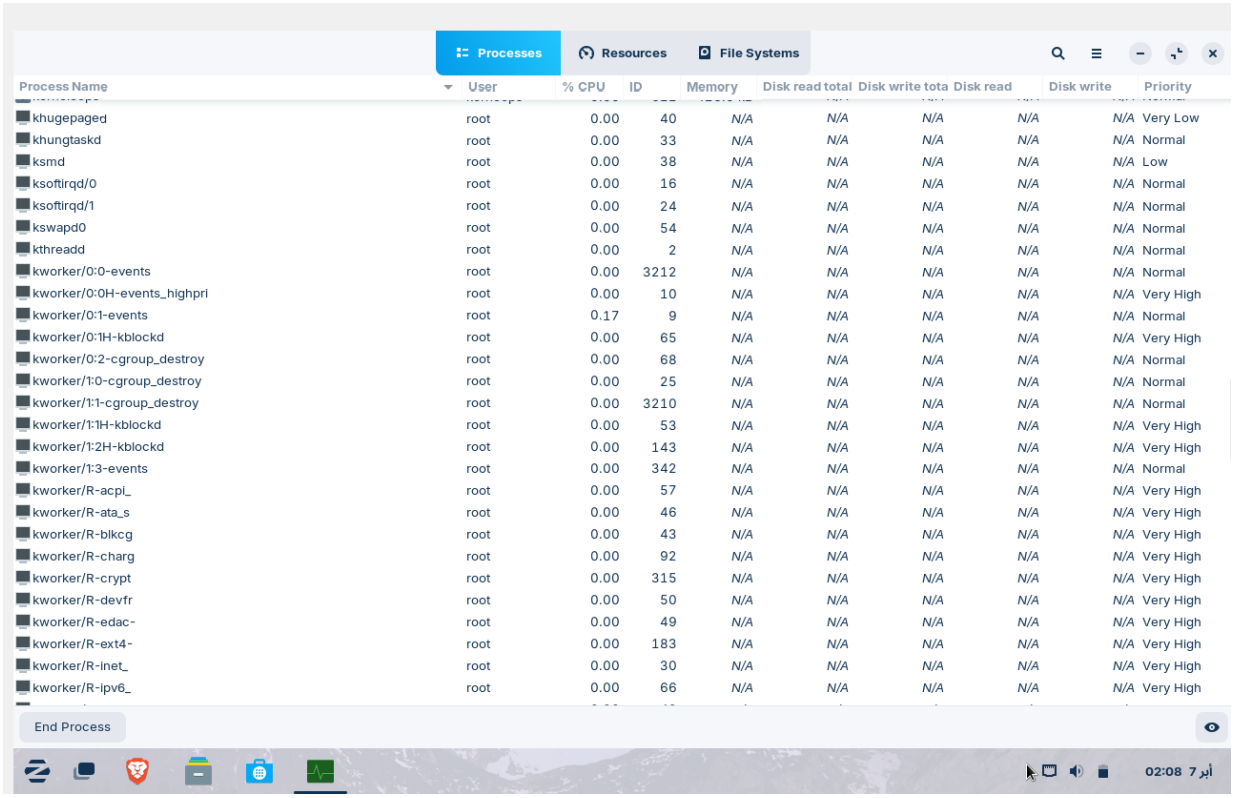


Figure 31: The list of running processes

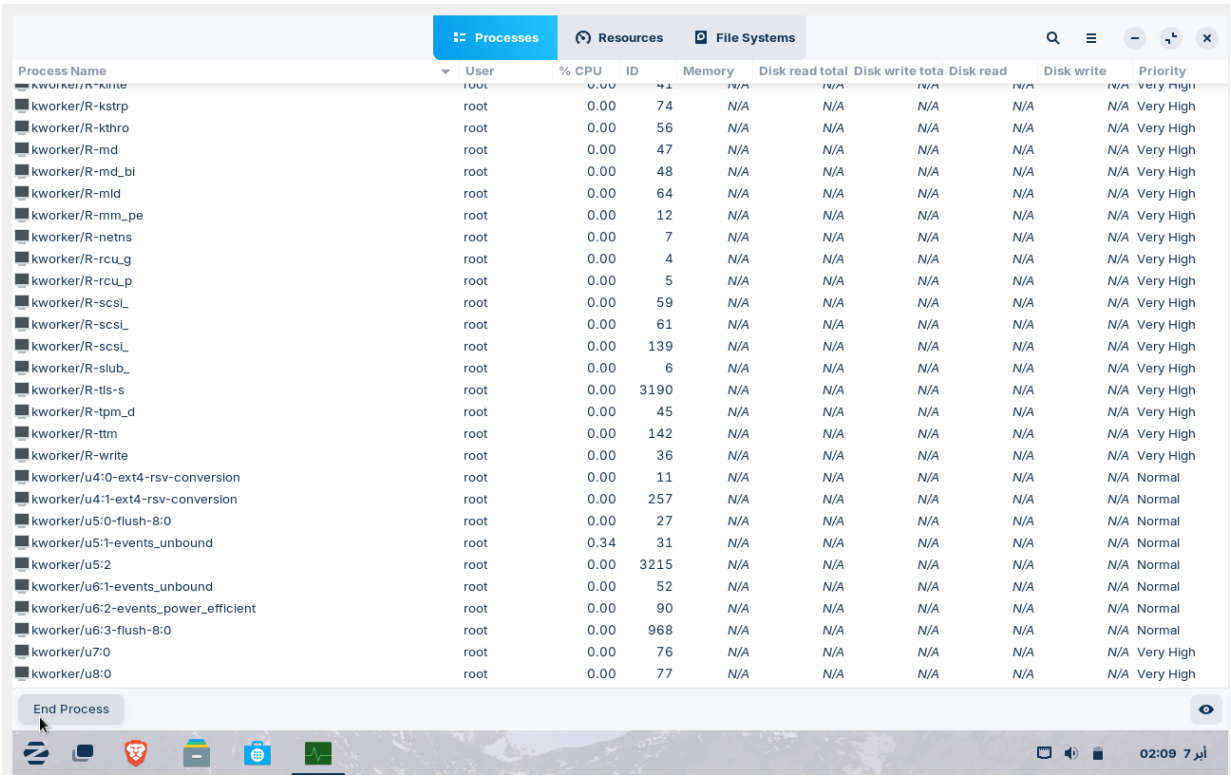


Figure 32: The list of running processes

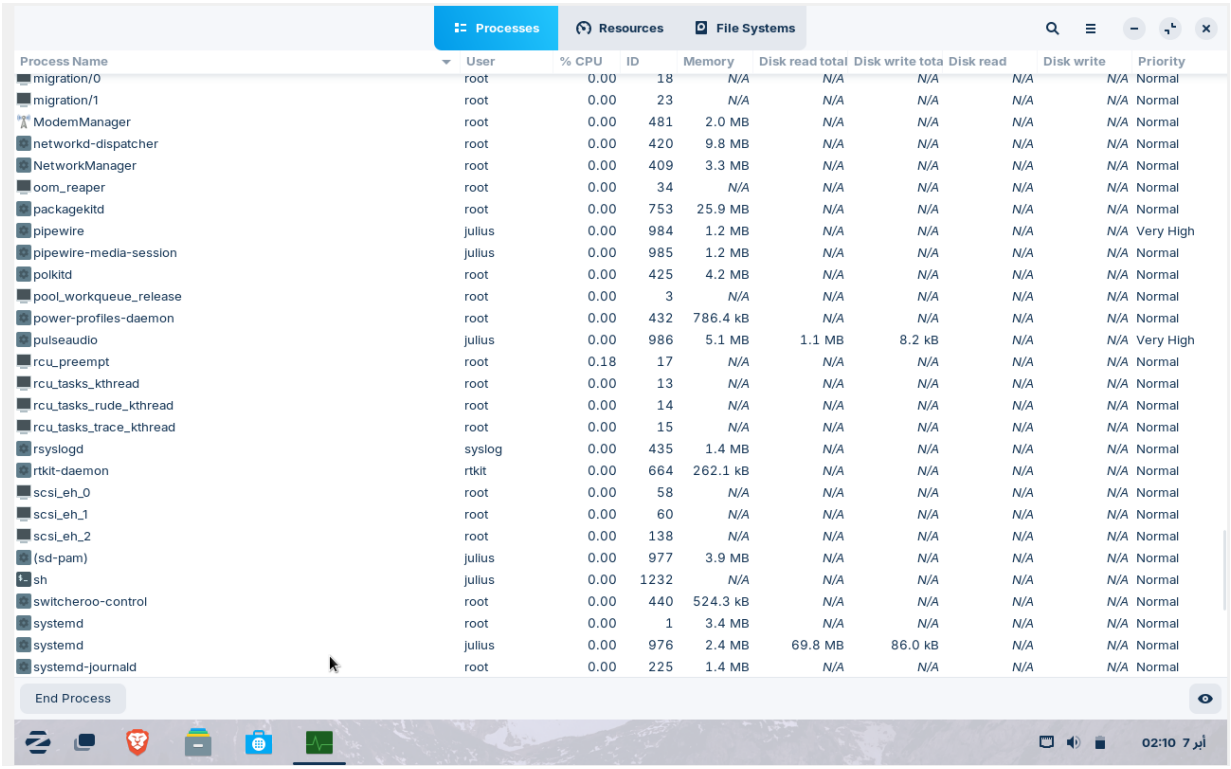


Figure 33: The list of running processes

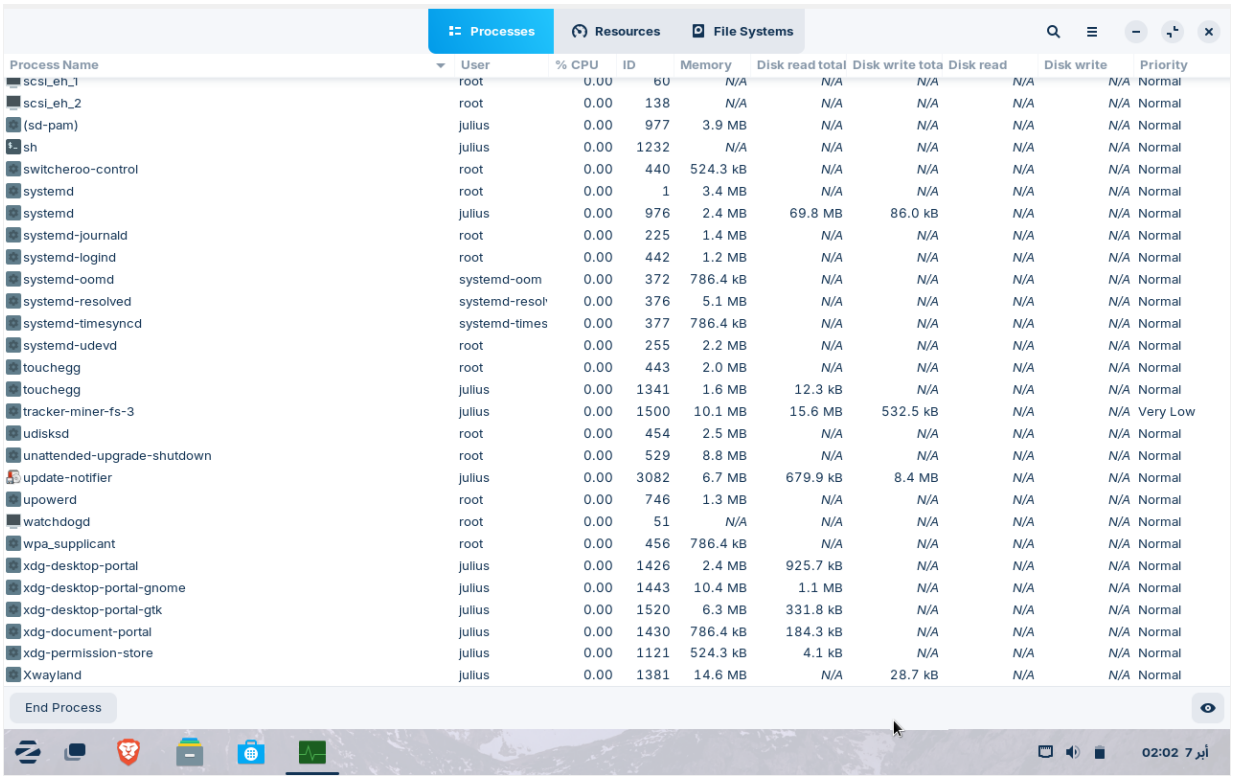


Figure 34: The list of running processes

Figure 35: The Resource Graph

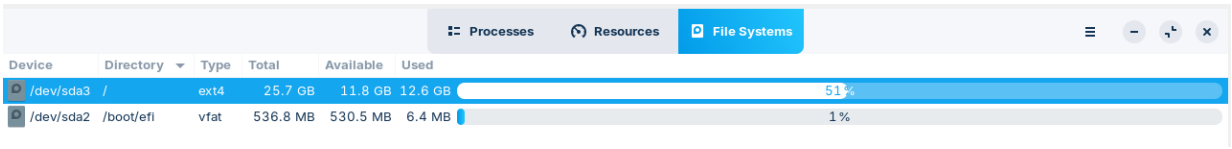


Figure 36: The File Management

13 Analyze Process and Thread Management

13.1 Overview of the Running Processes on Zorin OS

In this list of processes we, can view a mix of different kinds of processes such as system-level services, desktop environment components and user applications. These can be categorized into:

13.1.1 System and Background Services

- `dbus-daemon`: this process aids the inter-process communication (IPC) framework for messaging and coordination utilized by many other processes.
- `NetworkManager`: it allows the managing of networking interfaces and ensuring connectivity, while making sure that the system has an established connection and can adjust to changes caused by network availability.
- `systemd`: This is the system and service manager of the Linux Operating Systems.
- `cupsd`: This process is responsible for handling printing services.
- **Audio Server** (`pulseaudio` or `pipewire`): This handles the stream mixing of audio, its routing and volume control for both the sounds of the system and the user applications.
- `avahi-daemon`: This provides a service discovery on a local network.
- `sshd`: This daemon Secure Shell (SSH) enables remote access.
- `gvfsd (GNOME Virtual File System Daemon`: This enables you to gain access to different file systems like remote and local so that file operations are ideal across distinct protocols.
- `udisksd`: This process provides Disk Management Services.
- `cron`: This process will schedule and execute periodic tasks.
- `tracker`: This process runs in the background all the time to index files and metadata, allowing for a quicker search and retrieval operations.
- `ryslogd`: This daemon allows for system logging.
- `polkitd`: This process manages the authorization policies.

13.1.2 GNOME/Desktop Components

- `gnome-shell`: This is the core process of the GNOME desktop environment that takes care of the graphical interface, window management and the overall desktop composition. Because it is largely multi-threaded, it can separate UI rendering, input entry and other desktop functions.
- `gnome-session`: This process manages and deals with the GNOME session.
- `gnome-settings-daemon`: This process allows the implementation of different settings for the GNOME desktop.
- `gnome-shell-calendar-server`: This allows us to manage the calendar events in the GNOME shell.
- `gnome-software`: This is an application for Software Management.
- `GNOME Settings Daemons (gsd-*)`: These incorporate a list of dedicated daemons which handle particular roles in settings:
  1. `gsd-media-keys`: This allows the management of global keyboard shortcuts for media control.
  2. `gsd-power`: This is useful for monitoring and handling events involving power.

3. `gsd-disk-utility-notify / gsd-printer`: Signals notifications for disk and printer events.
4. `gsd-keyboard & gsd-xsettings`: They are responsible for dealing with input entry devices and display sett

- `goa-daemon (GNOME Online Accounts Daemon`: This makes the integration and synchronization of online accounts like Google, Facebook with GNOME applications which are usually running in user space.
- `ibus-daemon`: This process means that the users can smoothly shift between different languages resulting from multilingual support and input techniques.
- `xdg-desktop-portal`: Through this, we can provide a standardized API for applications involving sandbox such as `Flatpak` or `Snap` to communicate with the host desktop environment, to deal with work involving file dialogs and screenshot requests.

13.1.3 User Applications

- `Evolution`: This includes email, calendar, contacts and tasks which acts as a frontal implementation for management for personal information.
- `evolution-data-server`: For purposes of Evolution, it runs in the background to save and manage the data such as email, calendar and contacts.
- `evolution-calendar-factory / evolution-addressbook-factory`: By these processes, we can run and handle the required parts for calendars and contacts from the Evolution.

13.2 Observation from screenshots

13.2.1 Process Distribution and Categorization:

In the images attached, the processes are categorized by user ownership. Most of the processes run under the "julius" user which is my alias name. These likely represent active user applications and desktop components. On the other hand, the background and system processes run under system and service-specific users which represent the Layered approach model of Linux in security and management.

13.2.2 Resource Utilization:

- **CPU Usage:** Most processes represent a near-zero CPU usage, which means that they are idle or in a waiting state, which is supportive that is because, I was not running any programs such as web browsers, or media players on the laptop at that moment. Processes like `gnome-shell` and `gnome-settings-daemon` represent a significant amount of CPU usage, however it is a continuous and low activity, which means that they are functioning to perform tasks that needs computational power.
- **Memory Footprint:** There is a detectable change in the memory allocation. *Lightweight* Daemons require a minimum amount of memory such as a few megabytes. Nonetheless, applications that involve user interactions like web browsers or multimedia tools consume a larger memory. Through this distribution, we can highlight the efficiency of memory management so that important processes are placed an emphasis on response while user related requests and applications may have more varied resource demands. We see that `gnome-shell` uses a significant amount of the memory which is assertive of the fact that the graphical user interface along with other system functionalities needs to be maintained.
- **Disk I/O**: Because of the several number of processes running in the background, these tend to be updated at regular cycles by the monitoring tools, because some processes may not show active disk usage. There can be temporary spikes or inactivity in the disk I/O which can signal which stage the process is at, reading or writing or even loading while performing tasks like file transfers or software updates.

13.2.3 Thread Distribution:

The system monitoring tools does not explicitly state the number of threads per process but may be viewed as a part of command line as shown below. For instance, `gnome-shell` may internally have several threads running to deal with rendering, extensions or even event managements. These thread counts allow us to understand the concurrent activities taking place within a single process, which is really just a part of the thread management strategies of the Linux Kernel.

13.2.4 Priority and Scheduling Indicators:

In the processes running in the images attached above, it is clear that the **priority** column suggests that most of the processes are running at the **normal** level which means that the kernel scheduler's Completely Fair Scheduler (CFS) is working properly by making sure that processes are being monitored and handled in a balanced manner, so that each process or thread gets an appropriate part of the CPU's time depending on the priority. Although, the threads are not each itemized, a higher thread count in an interfacing process might interfere with greater granular scheduling decisions so that the user interface is active.

13.3 Terminal Implementation for Processes

13.3.1 Processes Status (ps): `ps aux`

It is used in Unix-like systems (including Zorin OS, which is based on Ubuntu) to view running processes.

`ps`: stands for *process status* which is used to list processes running on the system.

`a`: shows processes from all users

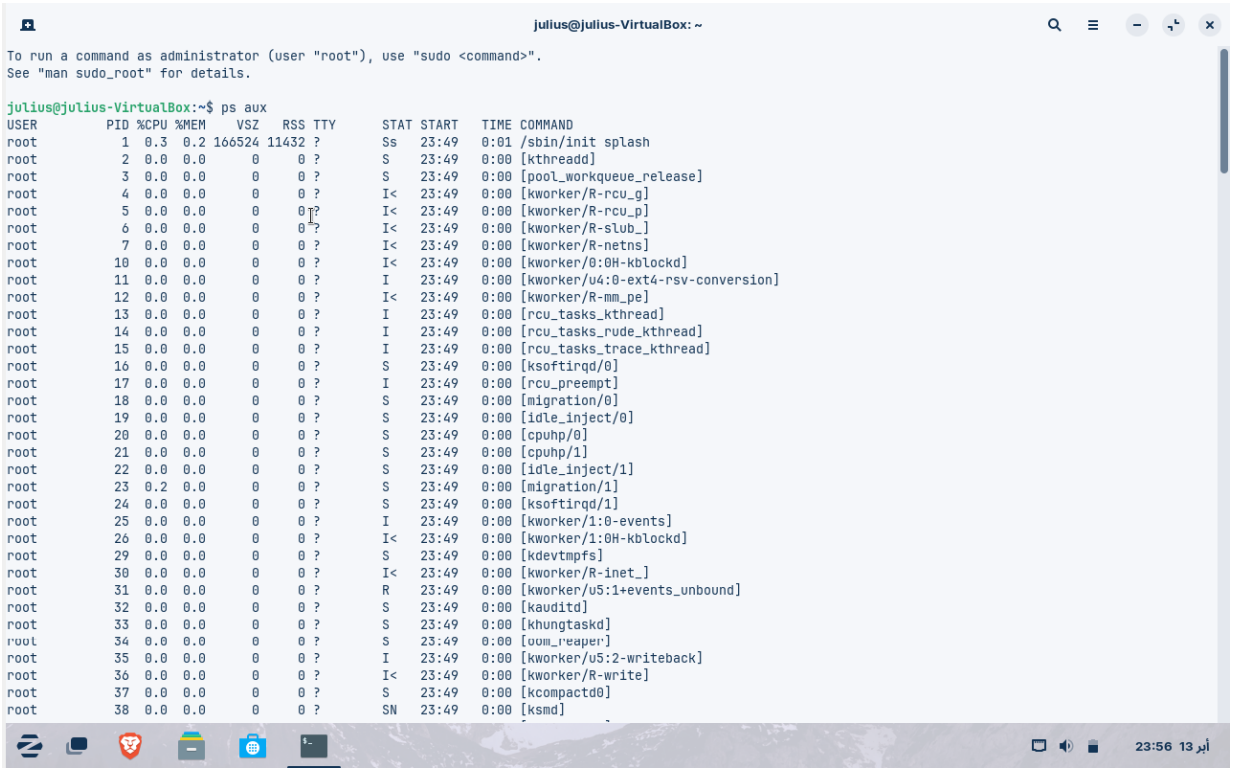
`u`: shows user/owner of the process

`x`: shows processes not attached to a terminal

So `ps aux` = "Shows all running processes in a detailed, user-friendly format, including background ones."

Sample Output Format:

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.1	168360	6560	?	Ss	Apr10	0:03	/sbin/init
username	2345	0.2	1.5	275000	61240	?	Sl	13:45	0:12	/usr/lib/firefox/firefox



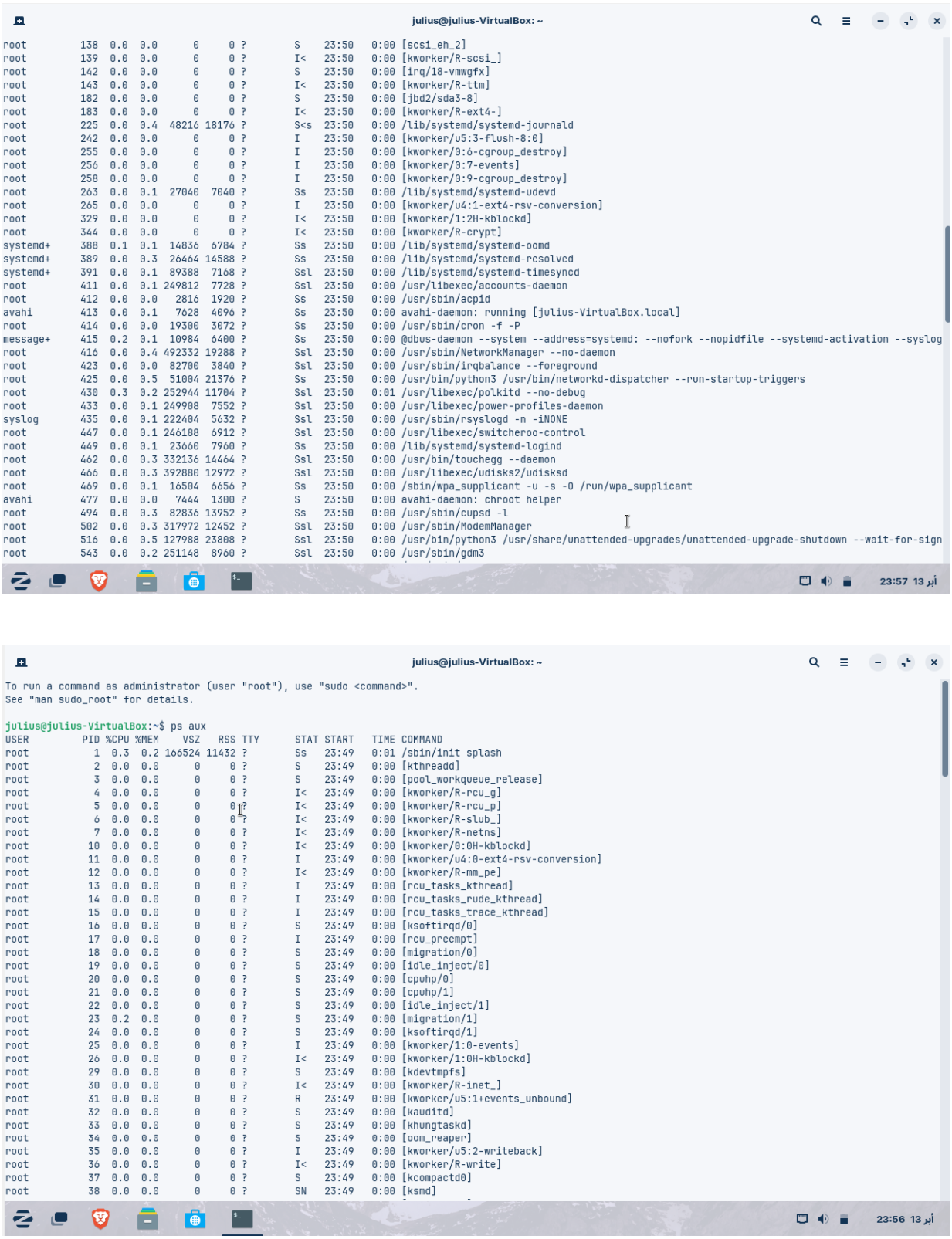


Figure 37: Gives us a list of running processes, including system and background daemons.

13.3.2 Filter with grep: `ps aux | grep gnome`

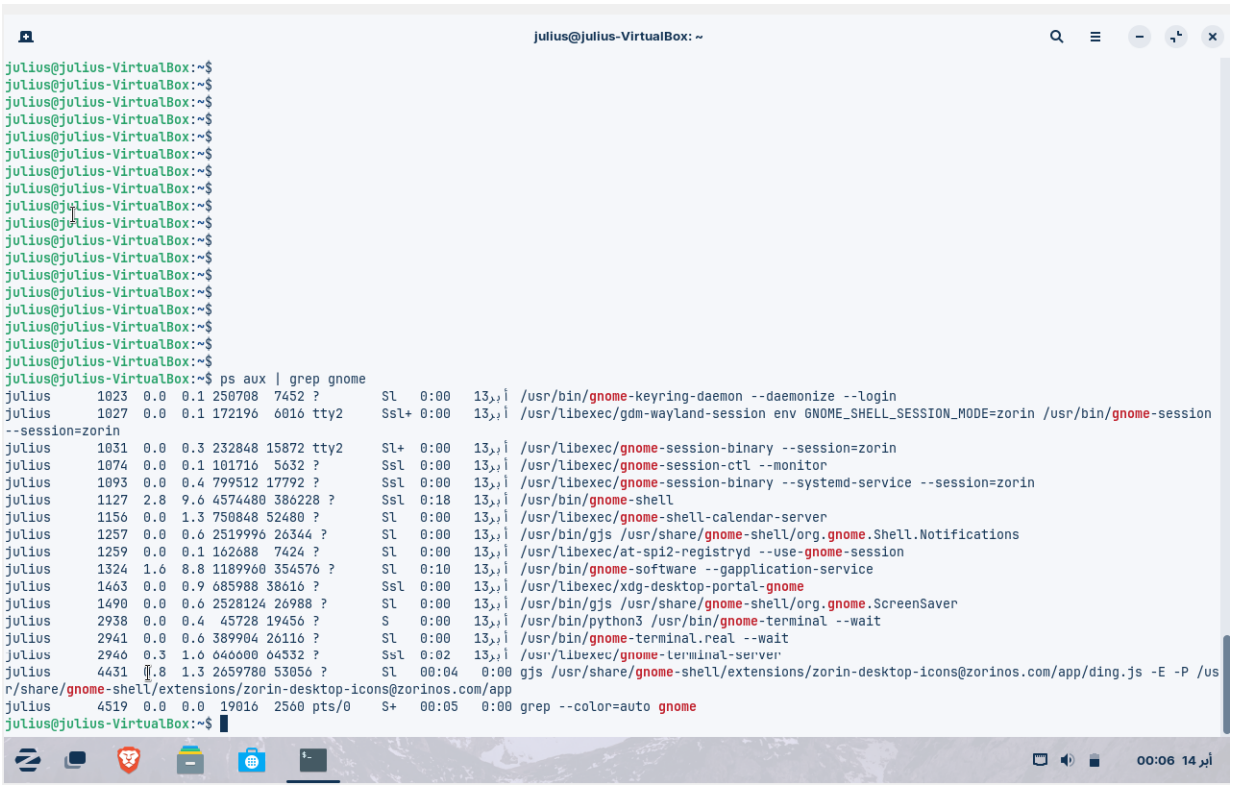


Figure 38: Filters to show only processes with `gnome` in their name.

13.3.3 Real-Time Process Viewer: `top`

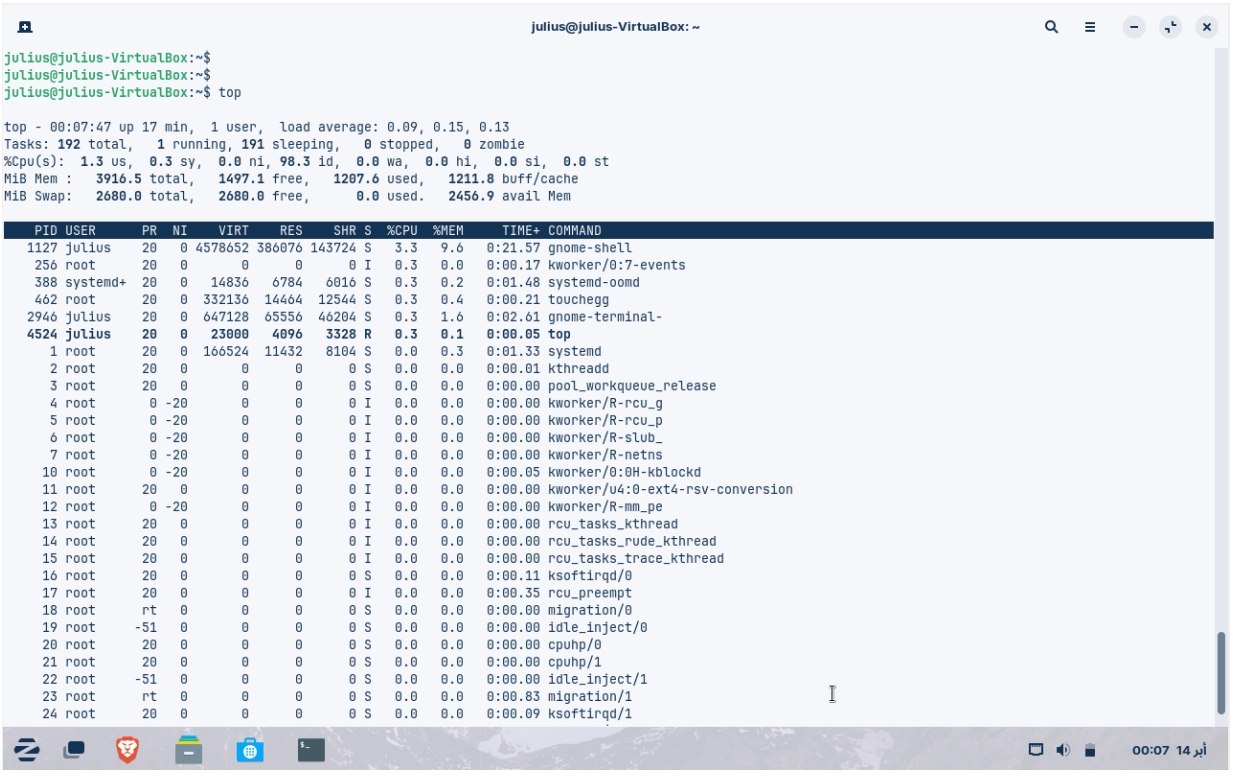


Figure 39: Shows live CPU and memory usage for processes. It refreshes every few seconds.

13.3.4 Improved top Interface: `htop`

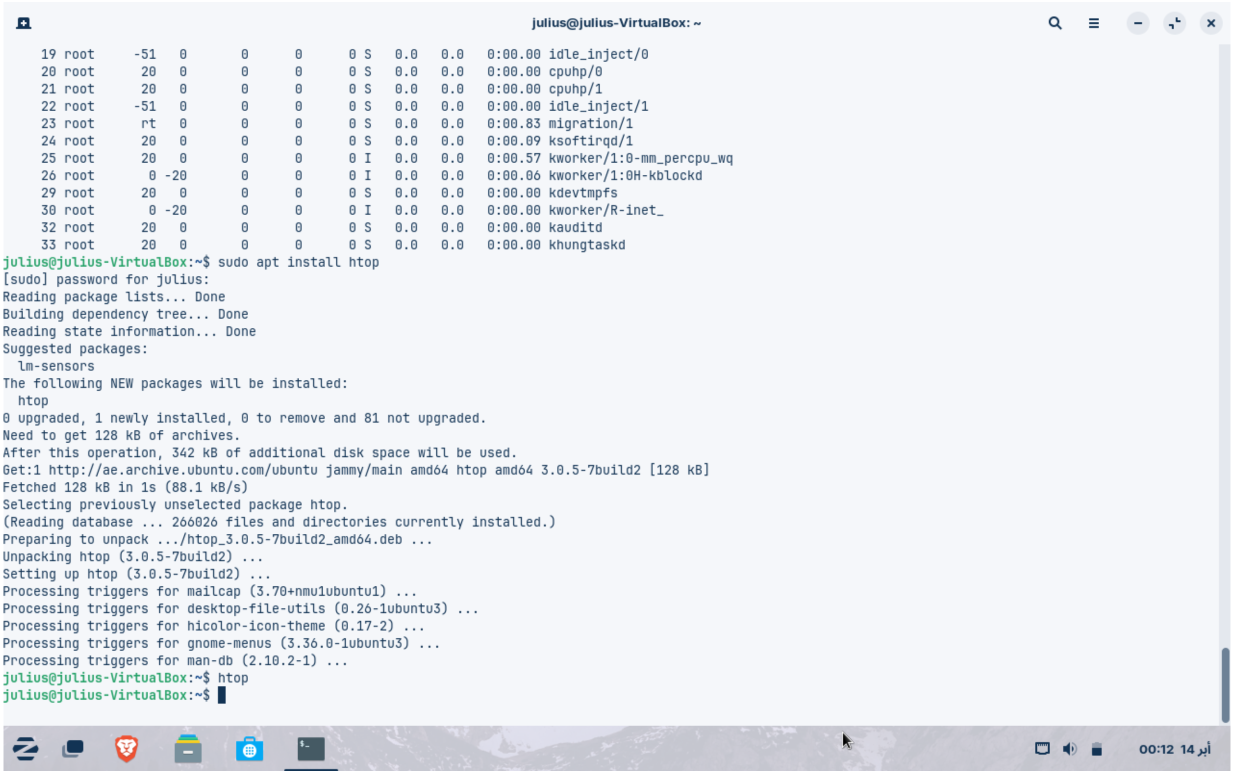


Figure 40: Install it first, if it is not already installed):

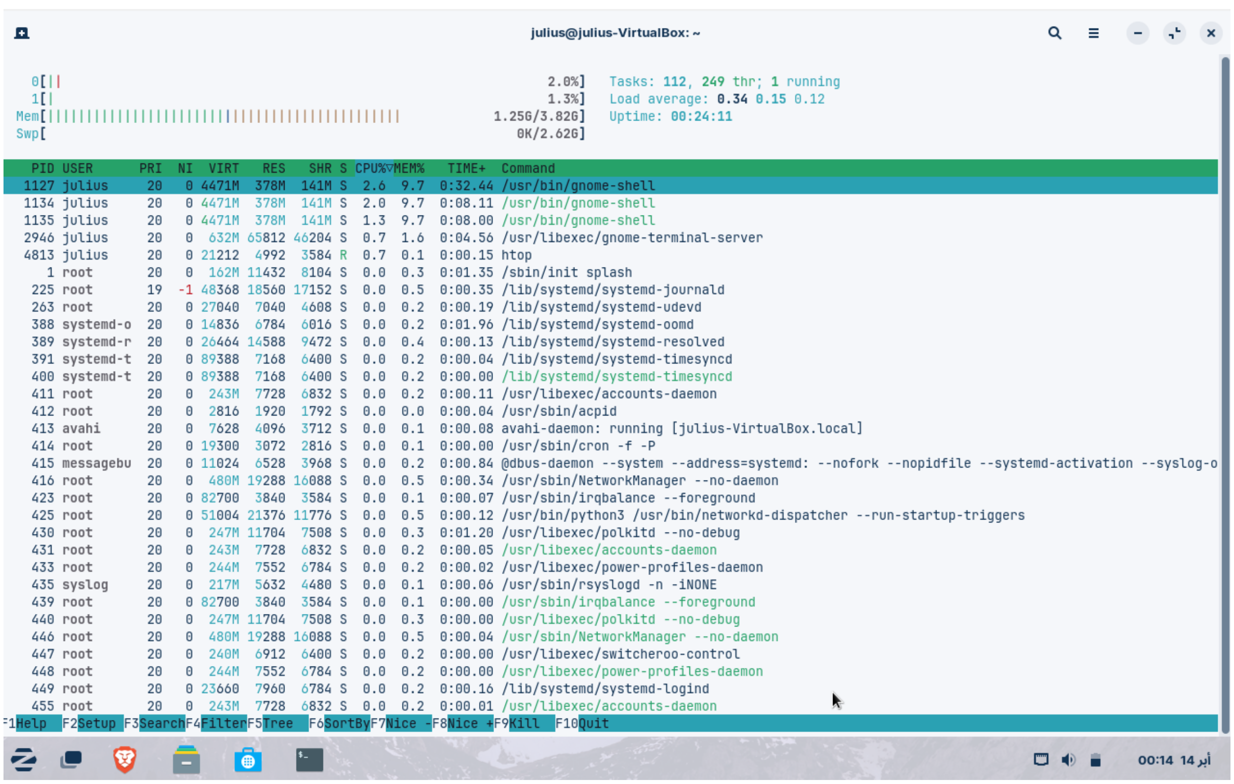
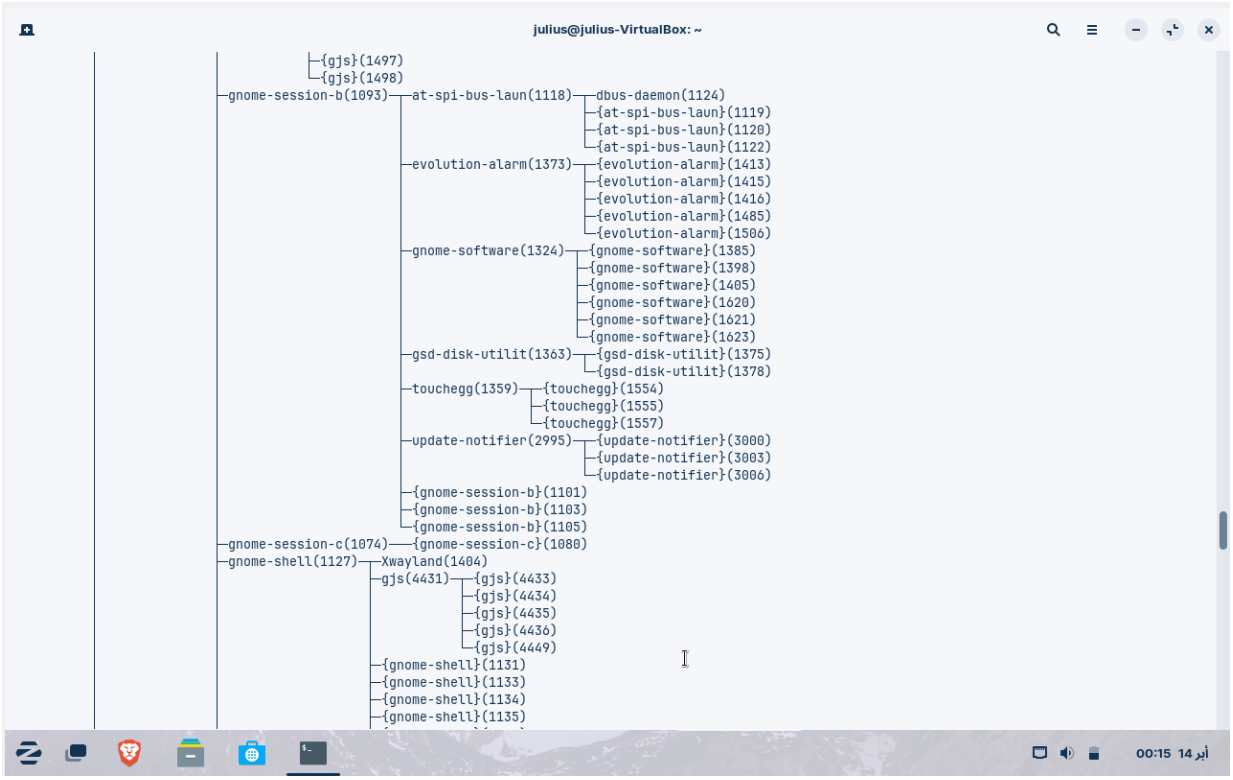


Figure 41: Provides a colorful, interactive process list, can be exited by using `ctrl+c`. It even allows you to scroll, search (/), and sort by CPU/memory as shown in the image in descending order:

13.3.5 Process Hierarchy: `pstree -p`



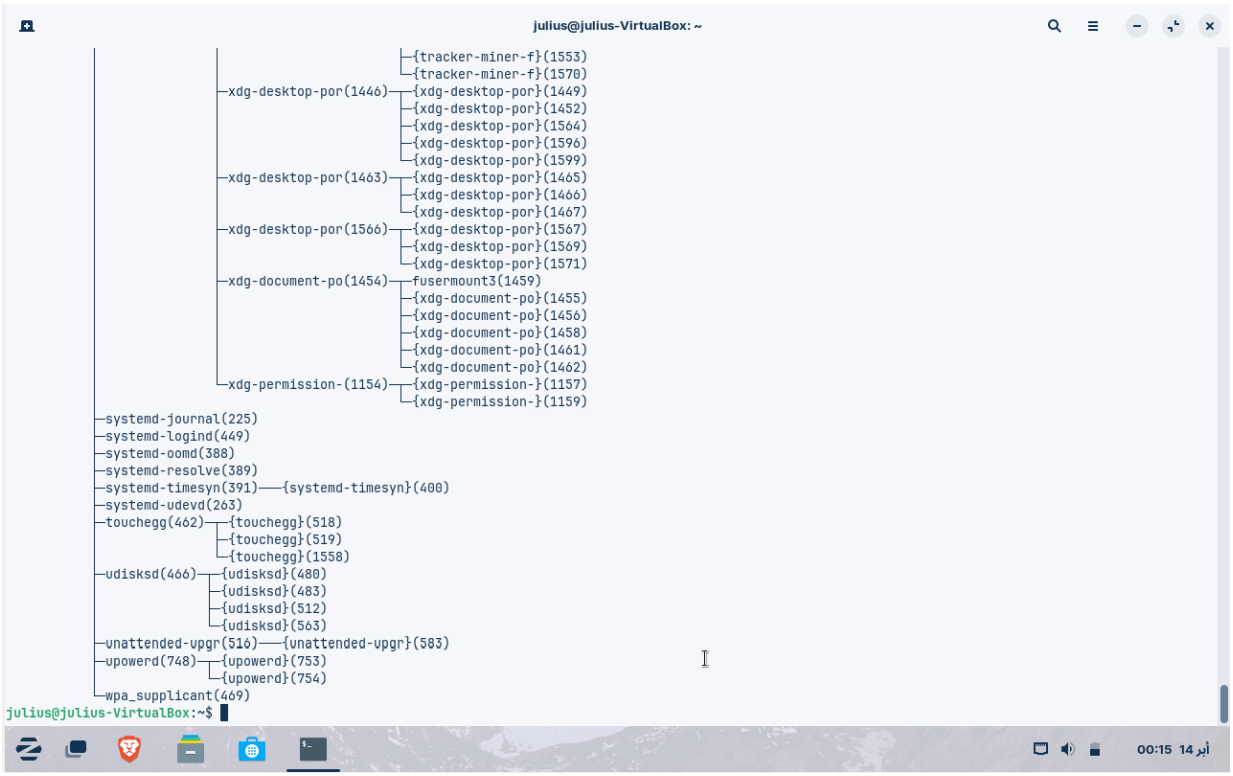


Figure 42: Shows a tree view of all processes

13.3.6 Viewing Daemons: `systemctl list-units --type=service`

Figure 43: Lists all the system services and their statuses

13.3.7 Specific Daemon: `systemctl status NetworkManager.service`

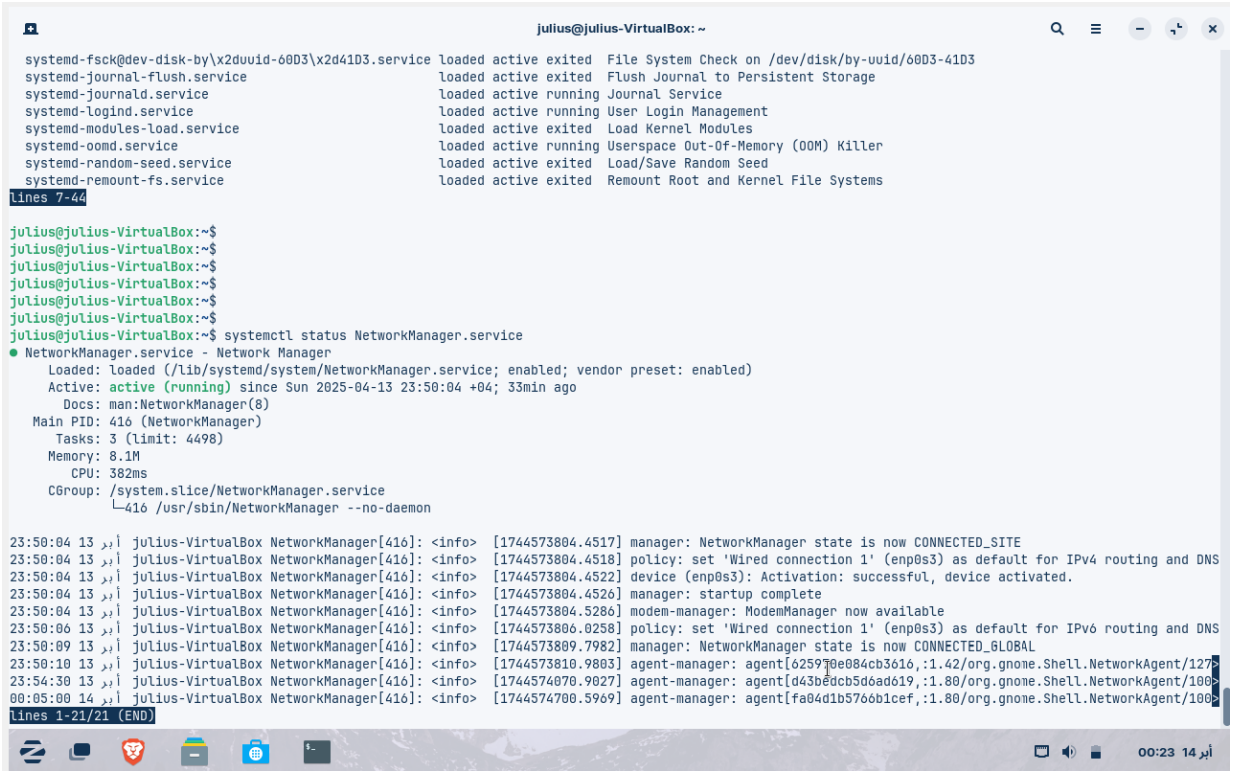


Figure 44: Checks background daemons not shown in GUI

13.3.8 Checks Processes for Current User: `ps -u $USER`

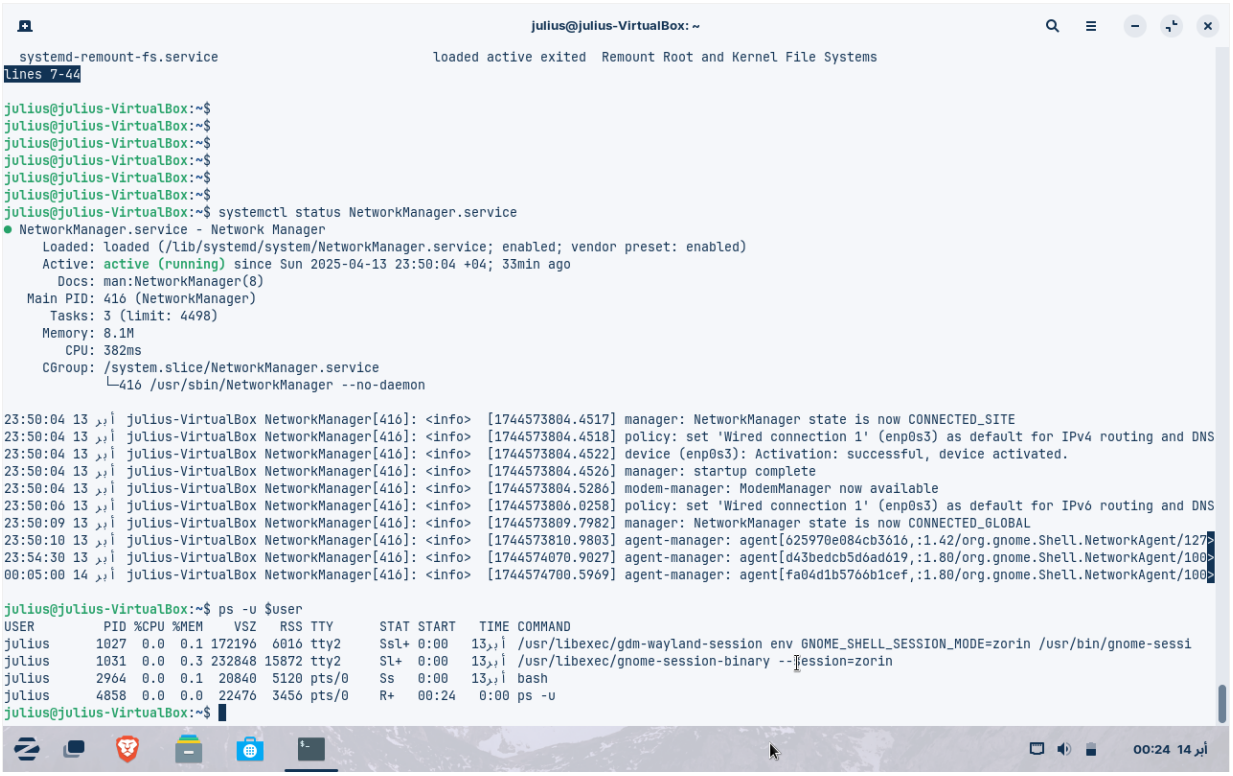


Figure 45: Shows only the processes running under your current user *julius*

13.3.9 View Active Daemons `ps -eo pid,ppid,cmd,tty | grep -v pts`

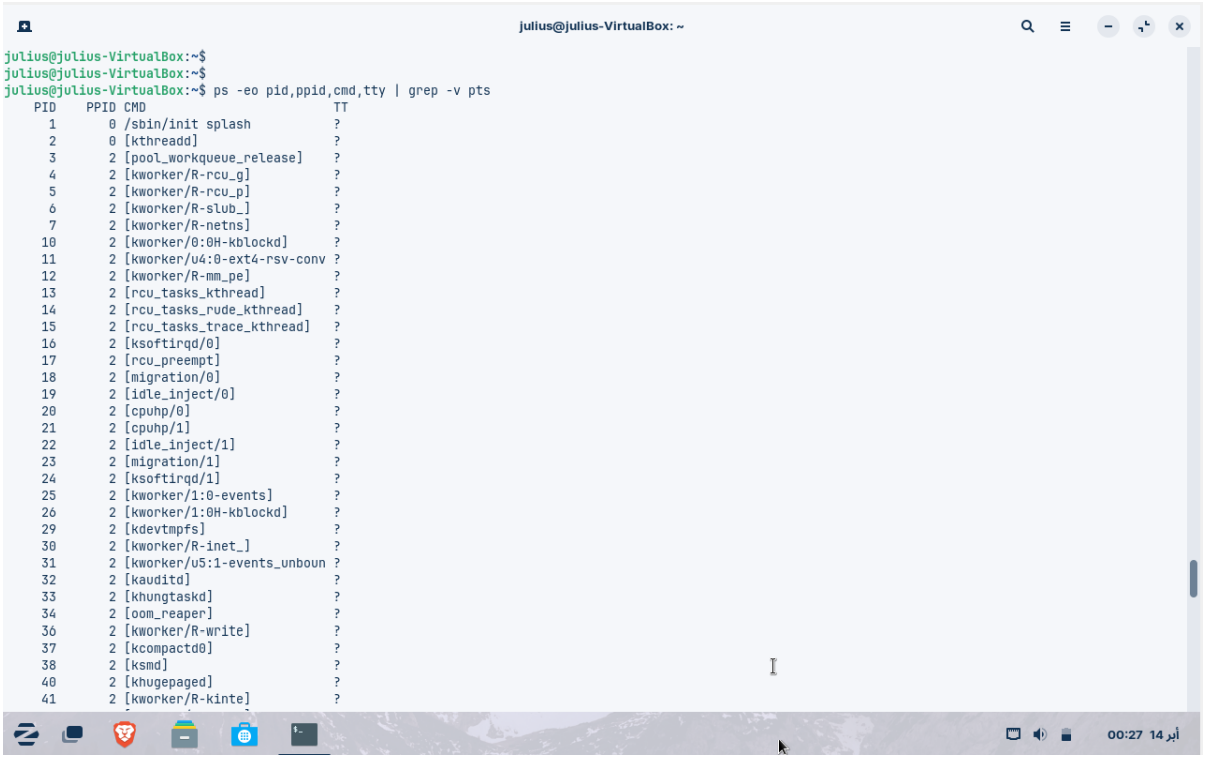


Figure 46: Filters out user terminal processes and shows only the Daemons

13.4 Terminal Implementation for Checking Threads

13.4.1 Find the PID (Process ID): `ps aux | grep gnome` then find the Thread using `ps`: `ps -T -p <PID>`

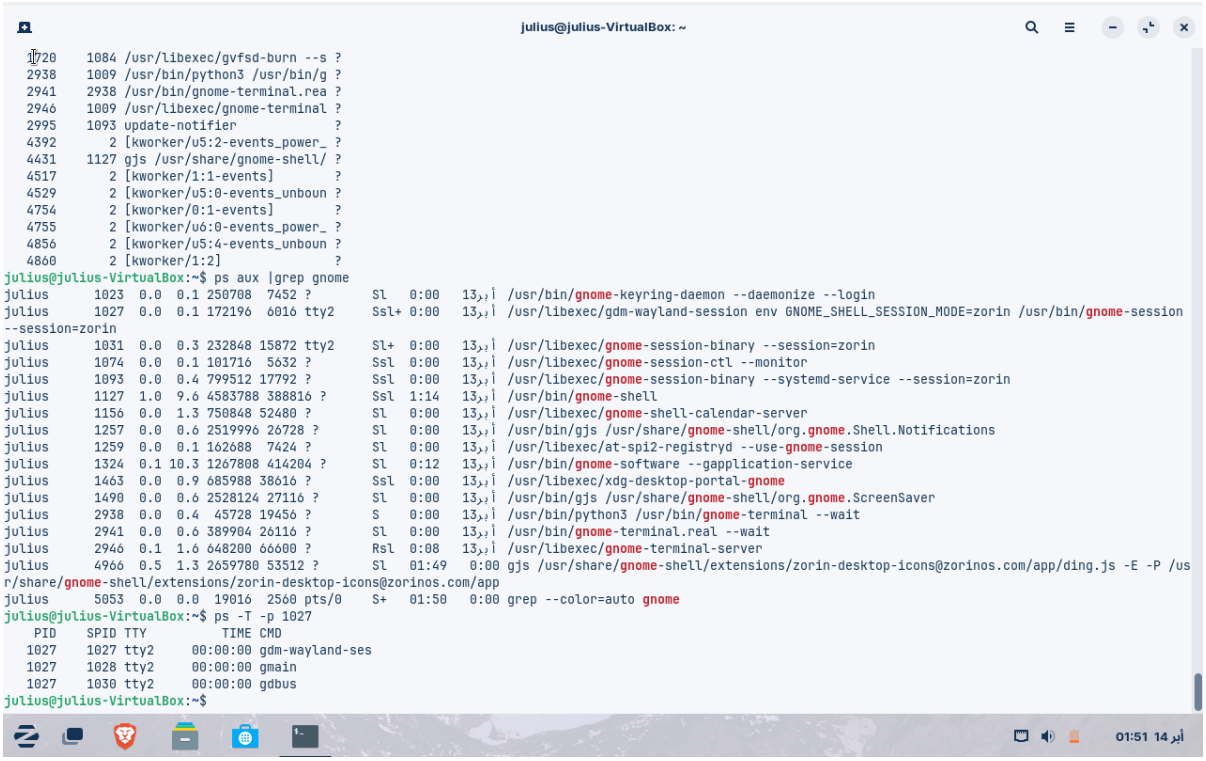
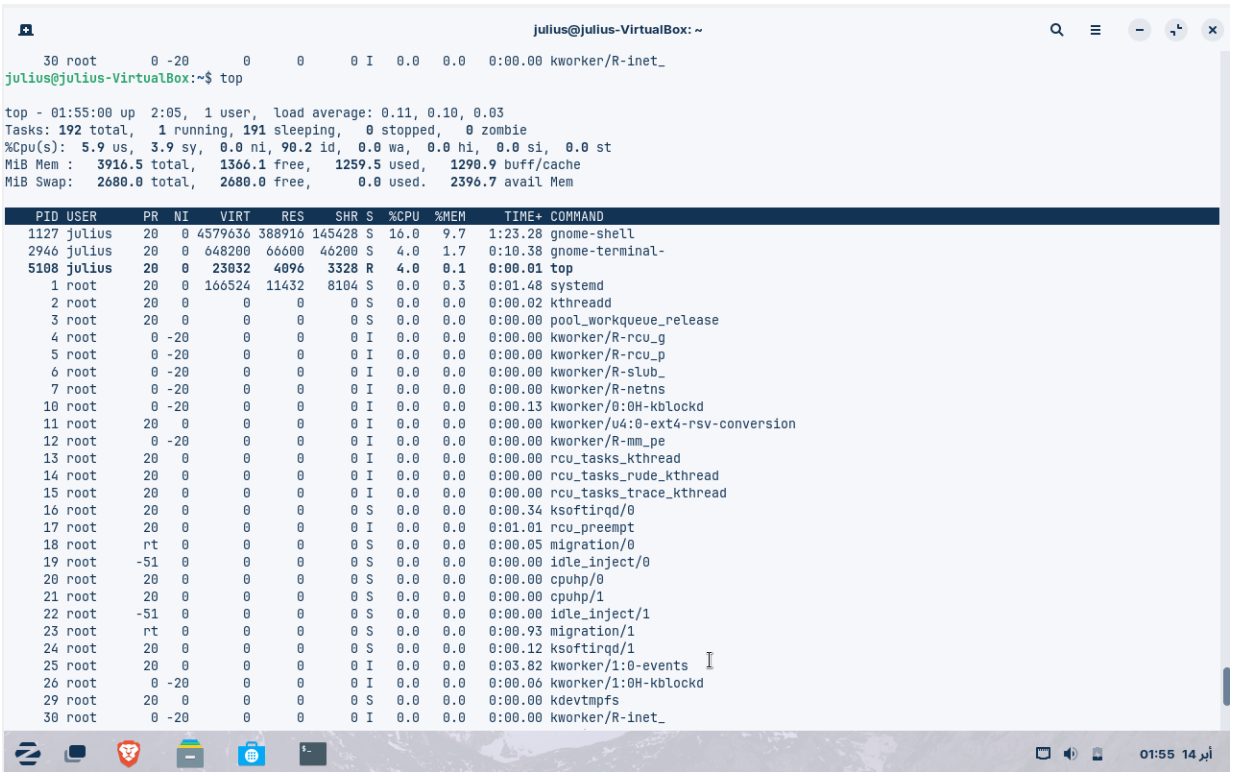


Figure 47: This will list all threads known as LWPs (Light Weight Processes) under that process

13.4.2 Using top to View Threads, start with `top` then press `H`



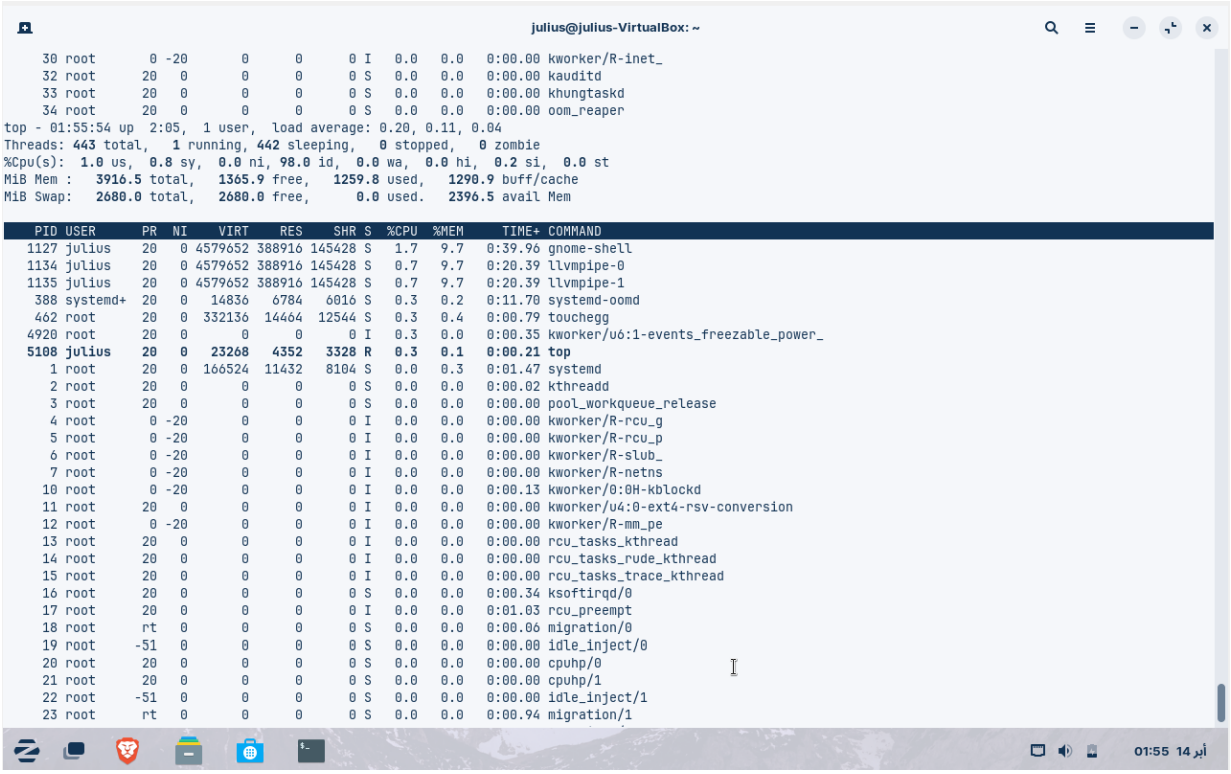


Figure 48: By pressing `H` we can change from showing processes to threads, so each thread will have its own LWP ID

13.4.3 Use of `htop` for Thread view. Install `sudo apt install htop`, if not already installed. Begin with `htop` then press `H`

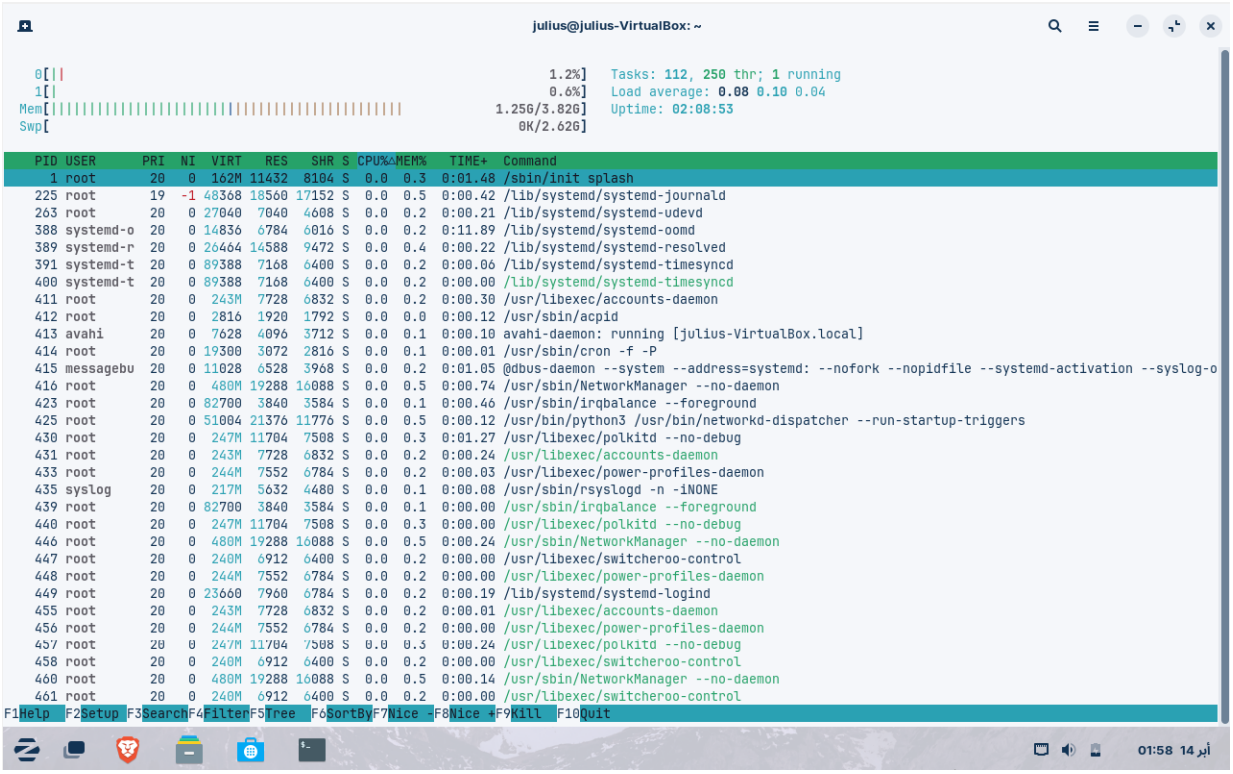


Figure 49: Shows threads indented under parent process with LWP numbers and different CPU usage per thread

13.4.4 Deep Dive of File System: `ls /proc/1234/task/`

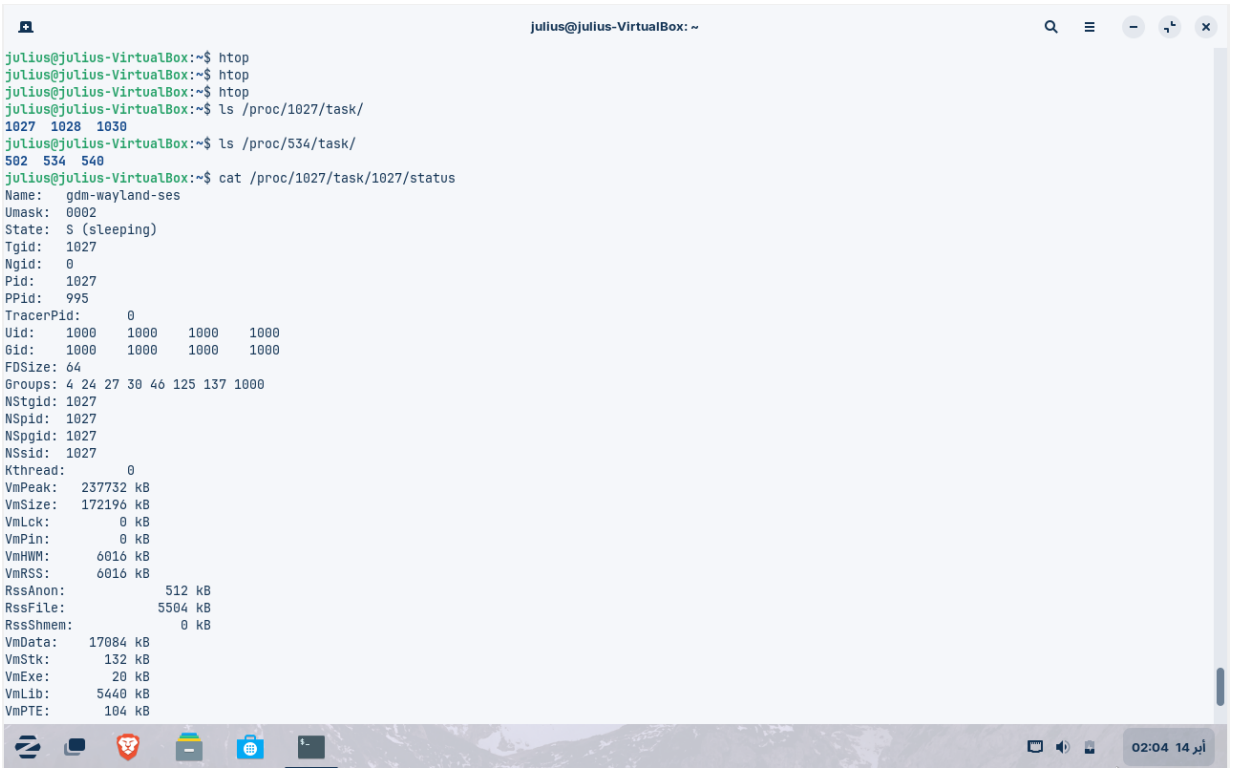


Figure 50: For PID 1027 and 534, each folder inside `/proc/1027/task/` and `/proc/534/task/` corresponds to a thread ID (TID) of that process

13.4.5 An even Deeper Deep Dive: `cat /proc/1027/task/1027/status`

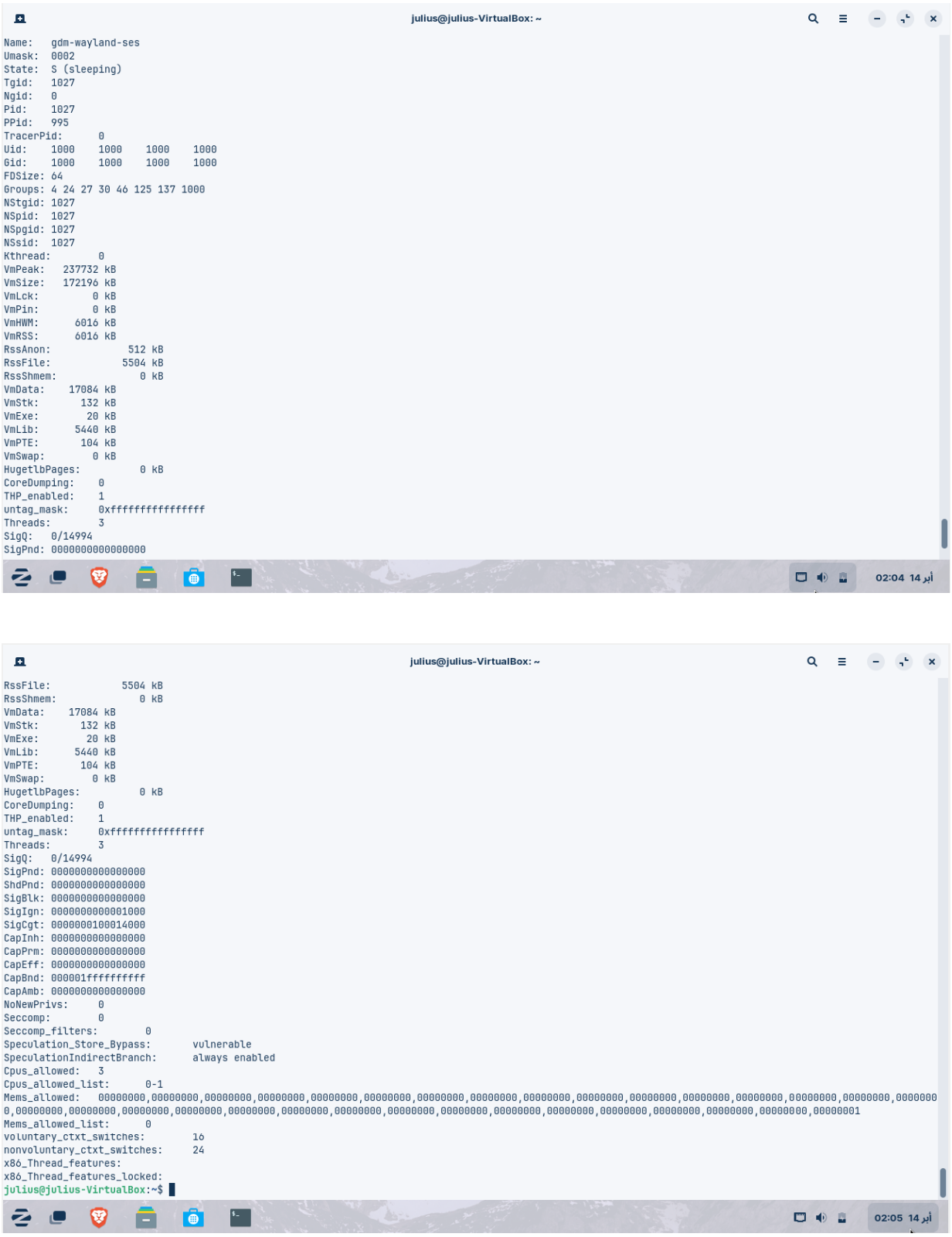


Figure 51: Shows the status of the main thread. By using `/proc`, we can do manual inspection through which Linux displays all threads and processes.

13.5 Thread Analysis

Through these screenshots, we can get a detailed overview of the thread behavior in Zorin OS. The primary tool used here was terminal, more of cmd version of Windows OS. The primary commands used as written above are `top, htop, /proc` filesystem inspection and `ps` shows us how the threads are handled in the user and kernel space. The system showcases a typical Linux thread architecture where Light Weight Processes (LWP) or threads share significant resources like memory and PID namespaces simultaneously monitoring different states of running.

One important takeaway here is from `/proc/1027/status` which refers to the process `gdm-wayland-ses`. Generally, this process was randomly chosen to represent in screenshots and analyze. It is responsible for dealing with the GNOME display manager, and spawns three threads (IDs 1027, 1028 and 1030). As indicated in the image, these threads are in a sleeping state `S` which means they are idle and hence waiting for events like user input or system calls. Moreover, through these memory metrics, we can see the shared memory usage like `VmRSS` for resident memory and even `RssAnon` for thread-private allocations which signifies that threads in a same process share resources but have separate stacks and execution contexts.

13.6 System-Wide Thread Distribution

The outputs resulting from the command `top` allow us to view from a wider perspective. We can observe that the system reports between 192 to 443 total threads with most of them in a sleeping state. One of those threads however, is running actively when the screenshot was taken, representing low system load. With high resource consumers such as PID 1127 `(gnome-shell)` and PID 2946 gnome-terminal are at the highest in user-space threading while threads related to kernel like `kworker, rcu` are used the least in terms of resource utilization but are crucial for operations in systems. The difference in total thread counts in the images accounts to the dynamic nature of thread creation and destruction, especially in GNOME which is GUI environment.

13.7 Thread Hierarchy and Resource Allocation

In the image which includes the `htop`, there we can clearly observe the threads hierarchically, allowing us to understand the importance of parent-child relationships. For instance, Systemd and its child processes like `systemd-logind, systemd-resolved` illustrate how modern Linux based systems depend only on designs that are threaded for servicing. Because of the color coding produced in `htop`, we can further differentiate threads by CPU or memory use which tells us that GUI processes that are a part of GNOME components spawn more threads and drain a large amount of resources relative to background services or kernel threads. This also aligns with the data produced by `/proc` where `gdm-wayland-ses` threads share their memories like `VmLib` but have different execution stacks `VmStk`.

13.8 Kernel VS User Threads

In the results produced by the command `top`, the Kernel threads as `[kworker]` or `[rcu]` are running with the least amount of overheads with near-zero memory and CPU usage but are critical for tasks such as interrupt handling and memory management. On the other hand, user threads, particularly which come under `gnome-shell` exhibit higher resource demands. We can observe that threaded applications like GNOME terminal `gnome-terminal-` and systemd services `systemd-udevd` leverage threading to allow for parallelism. Through Python interpreters `/usr/bin/python3` in the process list, we also see the multithreaded scripting, which is common in modern Linux distributions.

13.9 Key points

- **Thread Efficiency:** We have seen that threads allow us to enable processes like `gdm-wayland-ses` in order to handle concurrent tasks like input handling and rendering, at the same time also sharing memory, which reduces the overheads compared to multiprocessing.
- **Dynamic Behaviour:** We have seen the fluctuation of thread counts based on the workload such as the 192 vs 443 threads, which displays the Zorin's ability to scale resources.
- **Tool insights:** `/proc` allows us to see the low-level thread details like the states or the memory while `top/htop` offers real-time management. When these tools are combined, it gives us a holistic view of threading.

13.10 Tracing System Calls

13.10.1 Using `strace`

Figure 52: Update and install manpages using `sudo apt update and sudo apt install manpages-dev`

Figure 53: Trace system calls for the `ls` command using `strace ls`

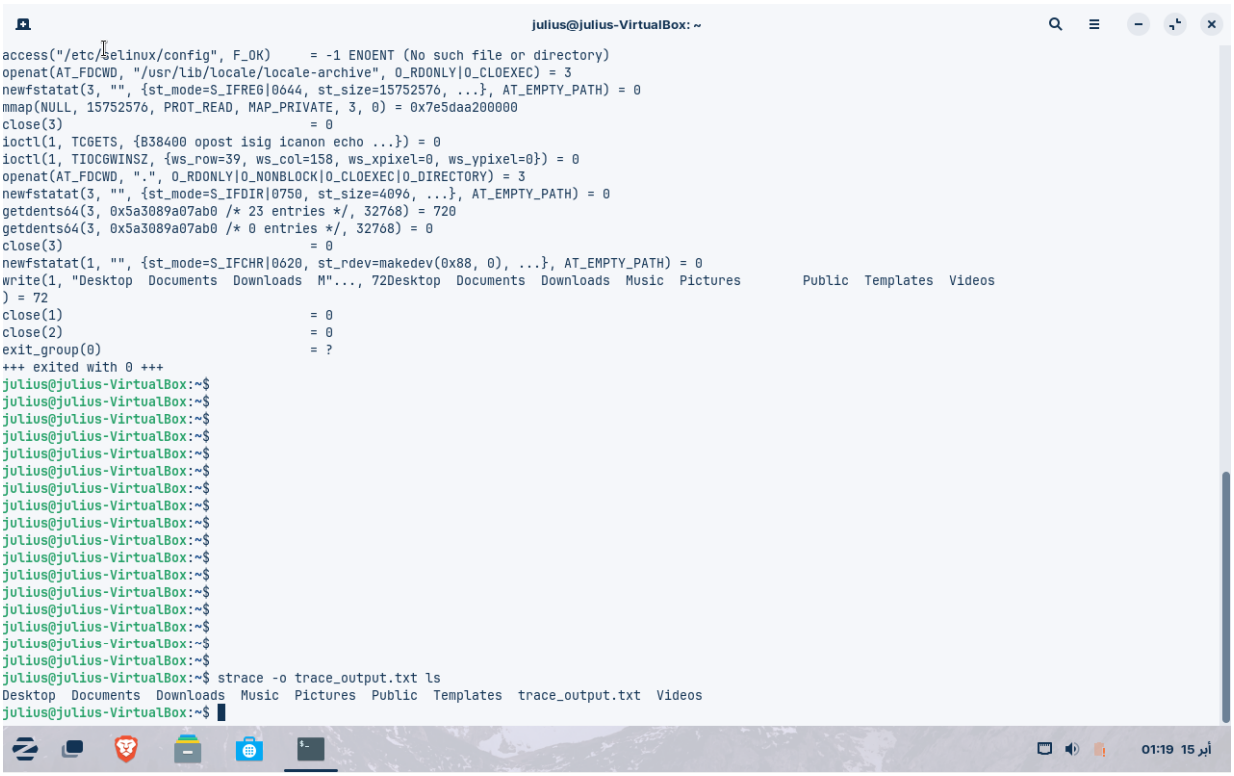


Figure 54: Redirecting output to a file using `strace -o trace_output.txt ls`

13.10.2 Use of `man` to see available system calls

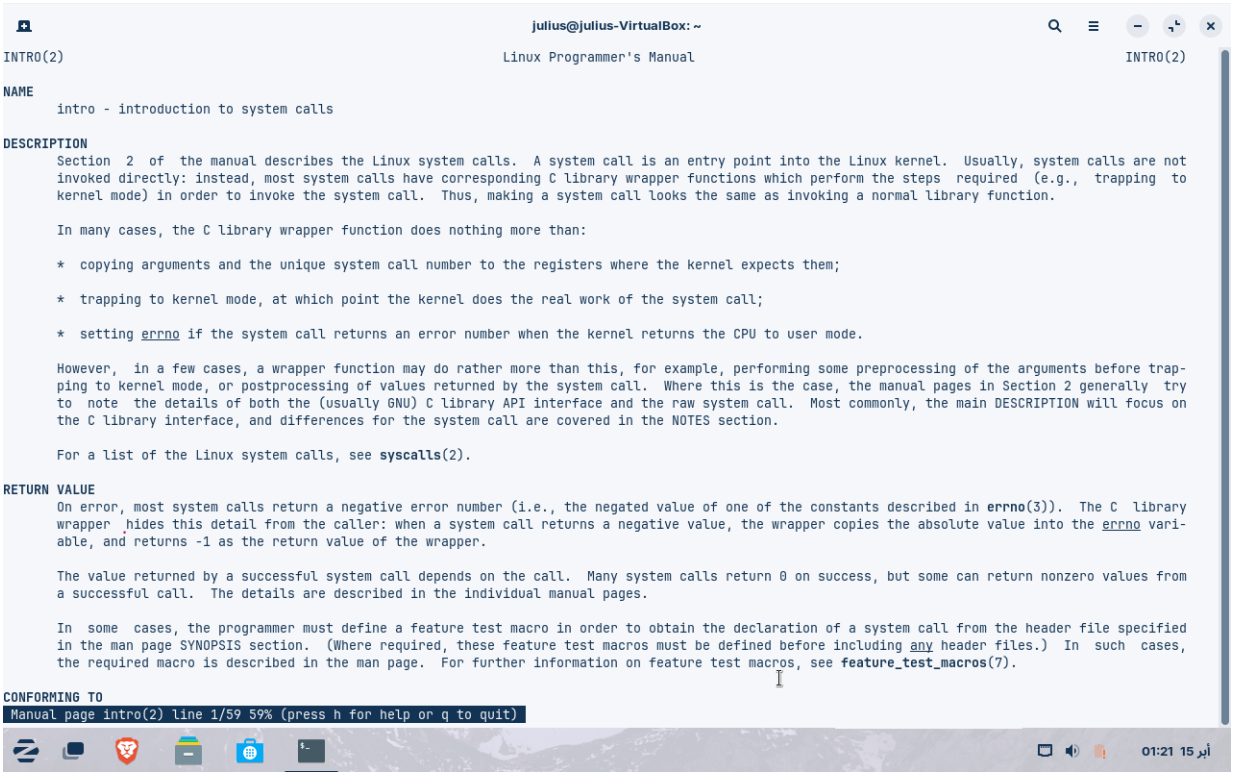


Figure 55: `man 2 intro` shows an introduction to section 2 of the manual, which is dedicated to system calls

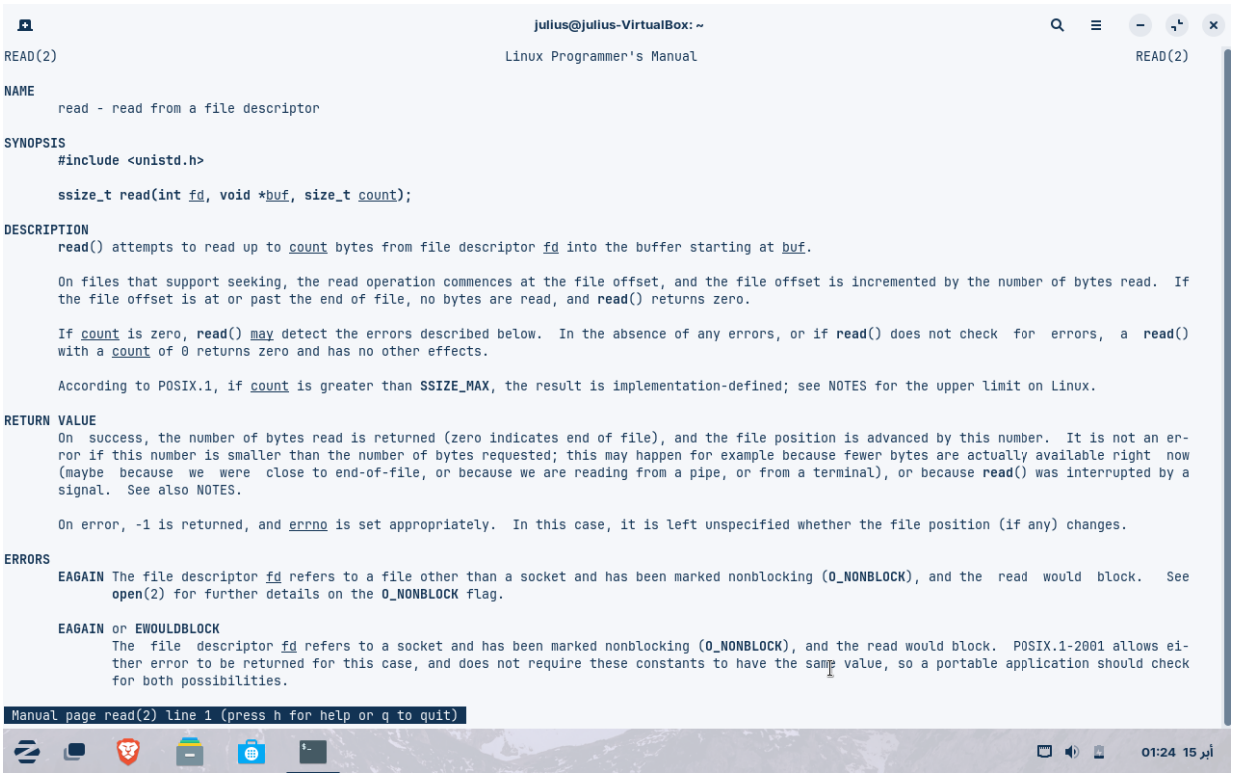


Figure 56: Use of `man 2 read` shows the manual page for the *read* system call

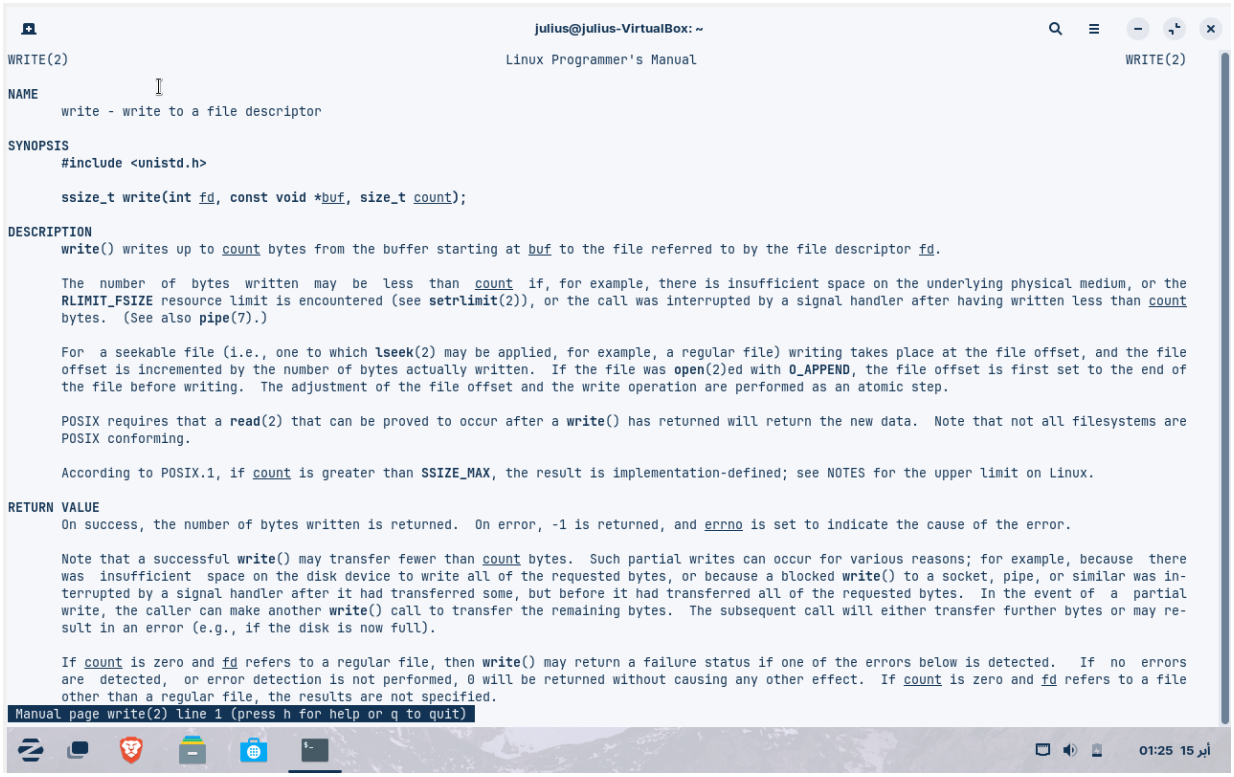


Figure 57: Use of `man 2 write` shows the manual page for the *write* system call

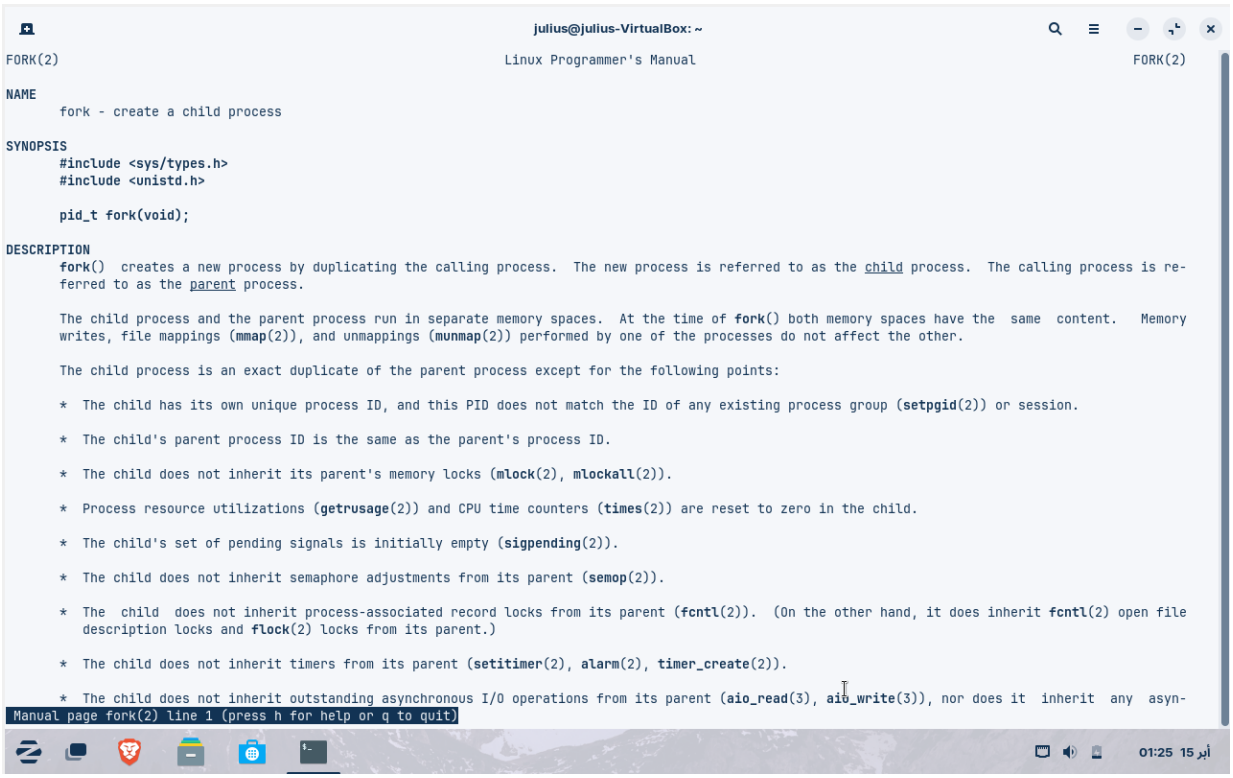


Figure 58: Use of `man 2 fork` Shows the manual page for the *fork* system call

13.10.3 Use of `ltrace` for library calls (different to system calls)

Figure 59: `Install the ltrace first using sudo apt install ltrace`

Figure 60: Running the `ltrace`

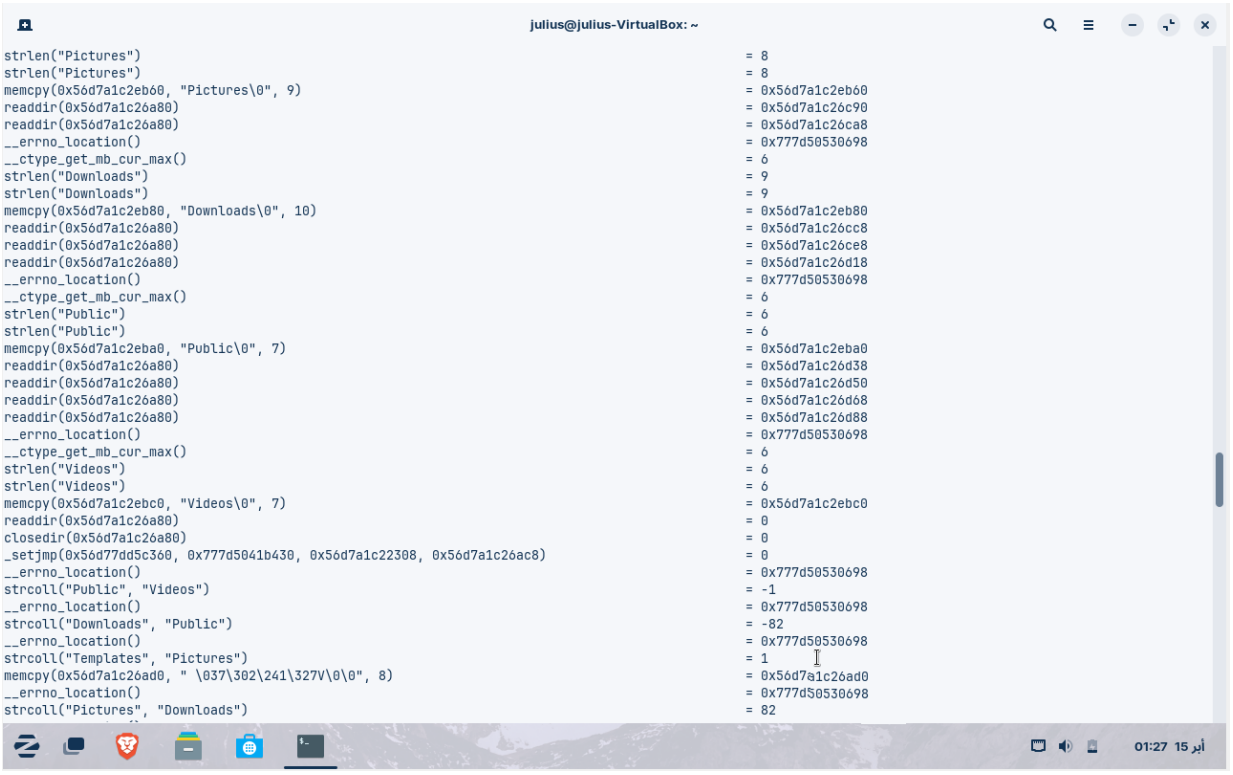


Figure 61: Running the `ltrace`

13.10.4 System Calls using C

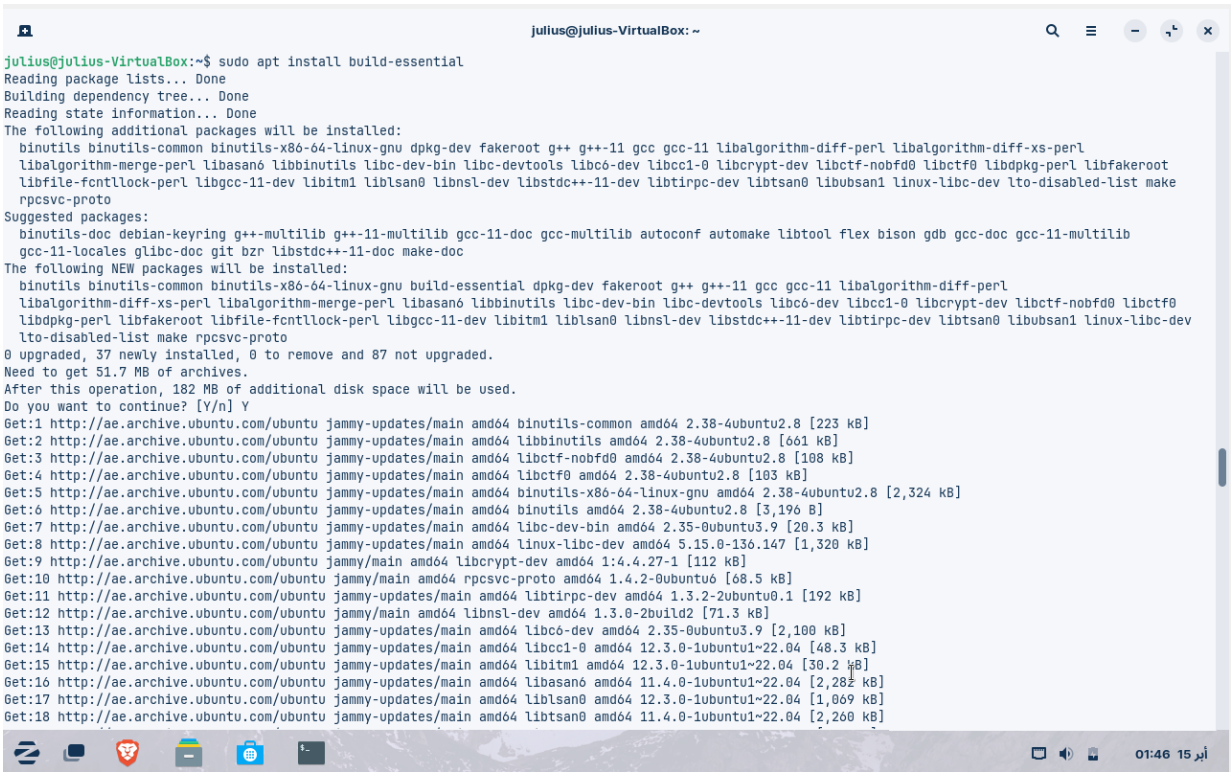


Figure 62: Install the `gcc`, the GNU compiler

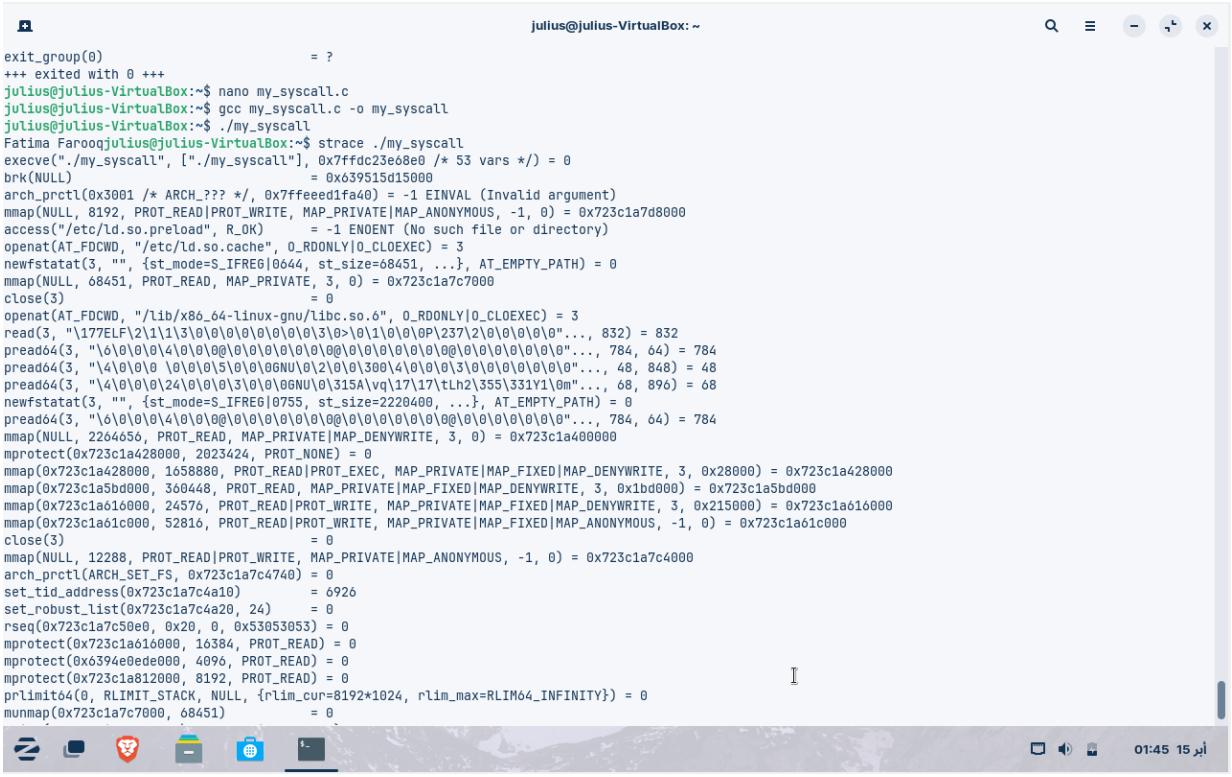


Figure 63: Use a text editor to create a C file, `nano`

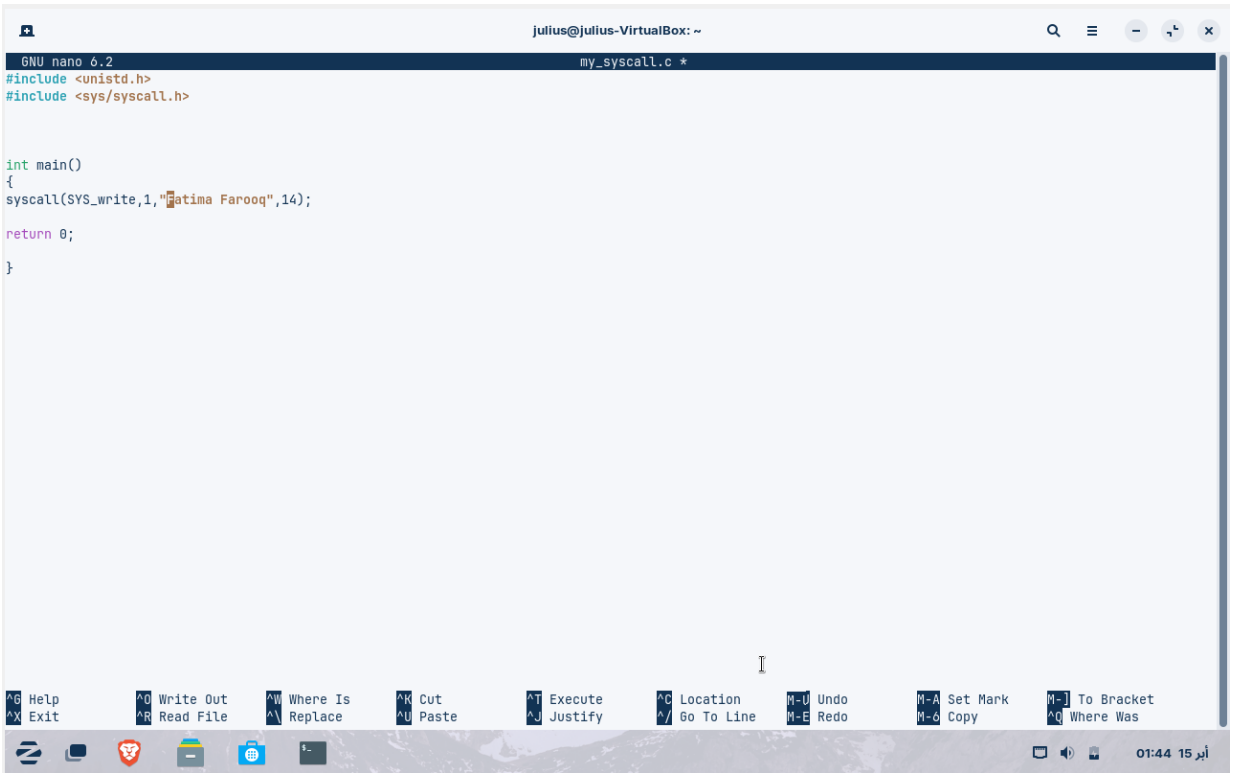


Figure 64: Type a simple C code and include the relevant libraries

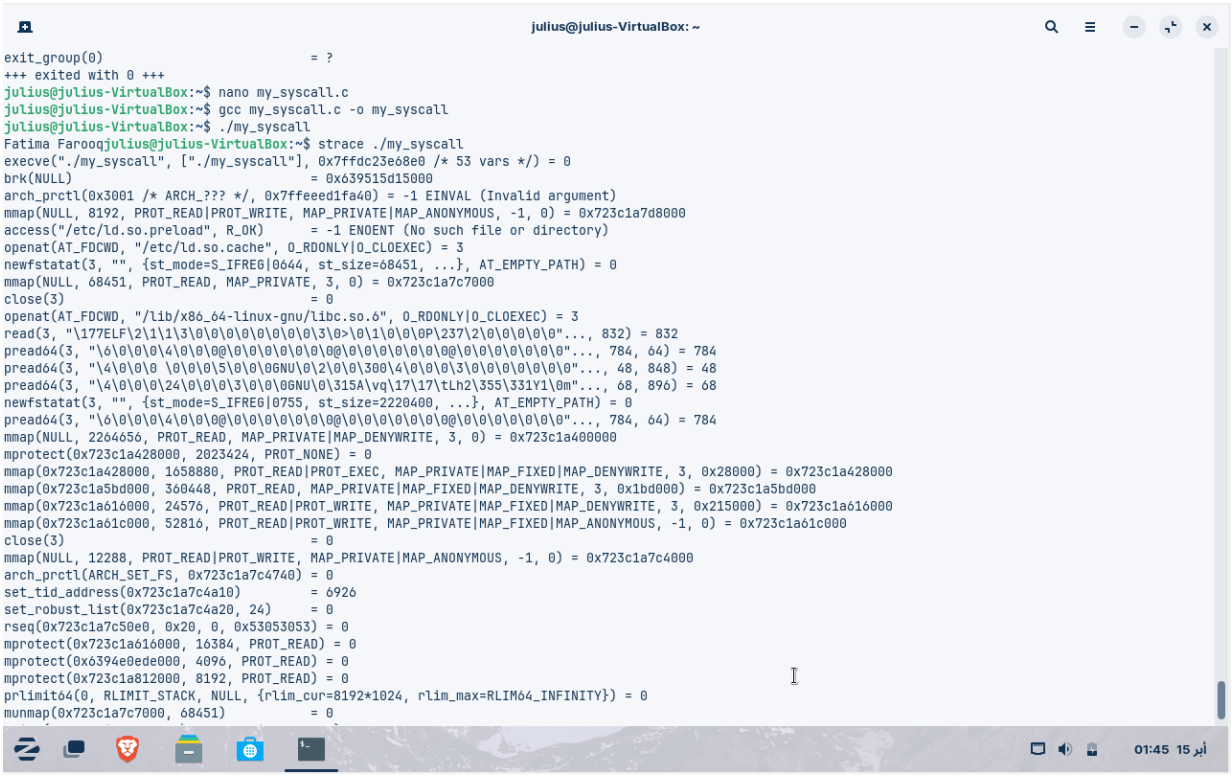


Figure 65: Compile and Run the program using `gcc my_syscall.c -o my_syscall` and `./my_syscall`

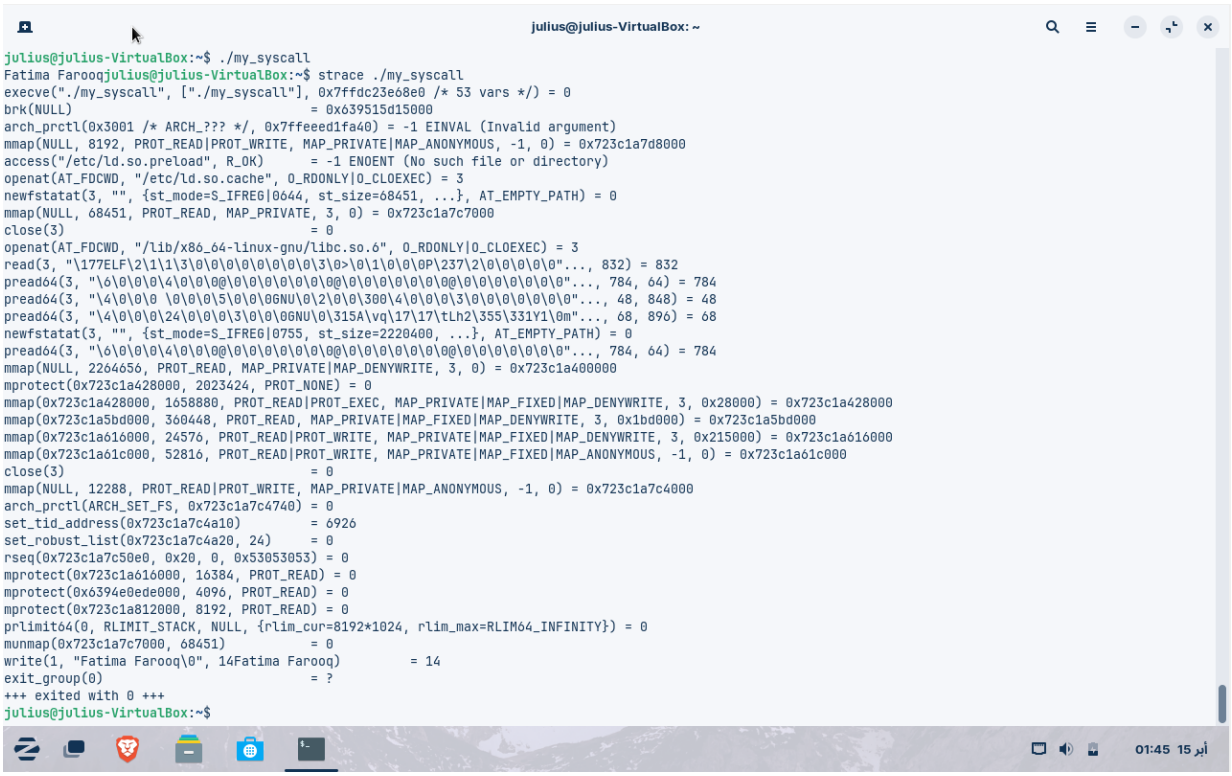


Figure 66: use `strace` to see the system call in action

13.11 Trace System Calls Explanation

Figure 52: Installing Man Pages
This figure illustrates the commands to update package repositories and install developer manual pages:

```
sudo apt update
sudo apt install manpages-dev
```

Explanation:

- `sudo apt update` refreshes the package index files from their sources, ensuring you have access to the latest versions of packages.
- `manpages-dev` contains documentation specifically for system developers, including detailed information about system calls, library functions, and kernel interfaces.
- These manual pages are essential for understanding the parameters, return values, and error codes of system calls.
- Without these developer manual pages, you would only have access to user command documentation, not the low-level system call information.

Figure 53: Using strace with ls Command
This figure demonstrates the basic usage of strace to trace system calls made by the `ls` command:

```
strace ls
```

Expanded explanation:

- When you run this command, strace intercepts and records all system calls made by the ls command and all signals received by it.
- The output shows each system call with its parameters and return value.
- For example, you'll see calls like `execve()` (to execute the program), `brk()` (for memory allocation), `access()` (to check file permissions), `openat()` (to open directories), and `getdents64()` (to read directory entries).

- This provides a real-time view of how the `ls` command interacts with the kernel to list directory contents.
- The verbose output reveals the underlying operations that happen when a seemingly simple command like `ls` is executed.

Figure 54: Redirecting strace Output to a File
This figure shows how to save strace output to a file for later analysis:

```
strace -o trace_output.txt ls
```

Explanation:

- The `-o` option redirects all strace output to the specified file instead of standard error.
- This is particularly useful for programs that generate hundreds or thousands of system calls, making real-time analysis difficult.
- Saving to a file allows you to:
  - Search for specific system calls using tools like `grep`
  - Analyze patterns of system calls
  - Compare traces between different program runs
  - Share the trace with others for collaborative debugging
- The command will execute normally without cluttering your terminal with trace information.
- The resulting file contains the complete trace in the same format as would be displayed on screen.

Figure 55: Viewing System Call Documentation
This figure shows how to access the introduction to system calls using:

```
man 2 intro
```

Explanation:

- The `man` command accesses the manual pages, and the number 2 specifically refers to the section for system calls.
- Other important sections include: 1 (user commands), 3 (library functions), 4 (special files), 5 (file formats), 7 (miscellaneous), and 8 (administration commands).
- The `intro` page in section 2 provides:
  - A general overview of what system calls are
  - How system calls are implemented in Linux
  - Common conventions for system call parameters and return values
  - Error handling mechanisms (using `errno`)
  - Information about system call wrappers in the C library
- This page serves as a foundation for understanding the system call interface before diving into specific calls.

Figure 56: Viewing Read System Call Documentation
This figure demonstrates accessing documentation for the `read` system call:

```
man 2 read
```

Explanation:

- The `read()` system call is one of the most fundamental operations in Unix/Linux systems.
- This manual page explains:
  - The function signature: `ssize_t read(int fd, void *buf, size_t count)`
  - Parameters: file descriptor, buffer to read into, and number of bytes to read
  - Return values: number of bytes read, 0 for end-of-file, or -1 for errors

  - Possible error conditions (with corresponding `errno` values)
  - Special behaviors and edge cases
  - Thread safety considerations
  - Related system calls like `pread()`, `readv()`, etc.
- Understanding `read()` is essential as it's the primary way programs obtain data from files, pipes, sockets, and other input sources.

Figure 57: Viewing Write System Call Documentation
This figure shows how to access documentation for the `write` system call:

```
man 2 write
```

Explanation:

- The `write()` system call is the counterpart to `read()` and is used for output operations.
- The manual page covers:
  - The function signature: `ssize_t write(int fd, const void *buf, size_t count)`
  - Parameters: file descriptor to write to, buffer containing data, and number of bytes to write
  - Return values: number of bytes written or -1 for errors
  - Partial writes: when fewer bytes than requested are written
  - Blocking behavior and how it can be modified
  - Interactions with file offsets
  - Special cases for different types of files (regular files, pipes, sockets, etc.)
- `write()` is used whenever a program needs to output data, whether to files, the terminal, network connections, or other processes.

Figure 58: Viewing Fork System Call Documentation
This figure demonstrates accessing documentation for the `fork` system call:

```
man 2 fork
```

Explanation:

- The `fork()` system call is the primary mechanism for process creation in Unix/Linux systems.
- The manual page explains:
  - How `fork()` creates a child process that is an almost exact copy of the parent
  - The function signature: `pid_t fork(void)`
  - Return values: 0 in the child process, child's PID in the parent, or -1 for errors
  - What resources are shared or copied between parent and child
  - Memory behavior (copy-on-write semantics)
  - Race conditions between parent and child
  - Limitations and potential failures
  - Alternatives like `vfork()` and `clone()`
- `fork()` is fundamental to understanding how processes are created in Linux, including how shells execute commands and how daemons detach from terminals.

Figure 59, 60, 61: Tracing Library calls
In these figures, we have traced the library calls, different to system calls.

```
sudo apt install ltrace
ltrace ls
```

Explanation:

- We begin by installing the `ltrace` command that is `sudo apt install ltrace`.
- `sudo:` This allows the command to be run using the superuser privileges which is necessary in order to install a software.
- `apt:` For Debian-based systems like Ubuntu (or Zorin OS), this is like a package manager.
- `install:` This signifies to the apt command, that a package needs to be installed.
- `ltrace:` This is the name of the package for the tool that is sued to make trace calls to the Library functions.
- `ltrace ls` is used to display and trace library function calls that are made using the `ls` command so the developers or technical users can observe the dynamic linking of the program, particularly the interaction between the programs and the shared libraries.
- Using `ltrace`, calls to standard library functions which are used by `ls` can we tracked.
- The parameters which are passed to the function are also showed and the return values are displayed as well, from those functions.
- `ltrace` can be used in cases of debugging and even understand the behavior of the program without accessing the source code. Moreover to learn to use standard tools that work *under the hood*.

Figure 62: installing `GCC` which is a `GNU` compiler.
Since we wanted to test this implementation using C, we started by installing the `GCC GNU` compiler.
`sudo apt install build-essential`

Explanation:
This command allows the installation of package on Debian-based systems as Zorin or Ubuntu (in our case) which consists of several tools that allow the compilation of source code into the software. It allows the installation of,

- `gcc` which is the GNU C compiler.
- `g++` which is the GNU C++ compiler.
- `make`, a tool that is used to control the process.

Figure 63 and 64: Type a test code
The C code was to be typed in a text editor such as `nano` in Zorin.

`nano my_syscall.c`

Explanation:

- Using this command, we can open the Nano text editor which allows us to write or edit the OS-level C source code. The command will create a new file with that name if it does not exist or opens the existing the file for editing the code.
- As in figure 64, the file called `my_syscall.c` has been opened of the `GNU nano` 6.2 version.
- We can see the relevant libraries `#include <unistd.h>` which is a header file used in UNIX based systems such as Linux that enables the POSIX operating system API.
- `#include <sys/syscall.h>` which is a header file that allows the definition of syscall numbers which are used in the `syscall()` function.
- The main function is a simple syscall function with which I have defined my name and a numeric parameter along with `SYS_write` parameter and return of 0. Now when it is run in the terminal, it should display my name and the number (14).

Figure 65: Compile and run the code
After exiting the `nano` text editor, we can use further commands for compilation and running of the code and finally trace it using system calls.

`gcc my_syscall.c -o my_syscall ./my_syscall`

Explanation:

- This command will instruct the `GCC GNU` compiler to compile the C source file `my_syscall.c` and produce an output result of an executable file in the form of my_syscall.
- `gcc:` This is the C code compiler.
- `my_syscall.c:` This is our C source code file that we worked on.
- `-o my_syscall:` This represents the output file name which in our case is `my_syscall.c` rather than the normal `a.out`.

Figure 66: Executing the code and using system trace calls
We then run the C code using the following command and then trace it using system calls.

```
./my_syscall strace ./my_syscall
```

Explanation:

- `./my_syscall` runs the executable from the current directory that is `./` refers to "here" and hence it executes the logic of system calls as we have seen above `fork() or write()`.
- When running `strace`, we will observe the output of the `main()` function that is my name and the number 14 so all the system calls will be shown that were made by program.
- This is useful for troubleshooting problems such as file not found or permission denied.
- It enhances our understanding of the code interaction with the OS.
- Helps us with the performance details, comparisons and statistics which is what this project is about.

14 Conclusion

The Zorin OS case study has provided a useful analysis of how a contemporary Linux-based operating system handles system functions such as process scheduling, thread management, synchronization, and deadlock avoidance. The case is taught in class and students are given real-time tools such as ps, top, htop, and strace to analyze the system during review, thus reinforcing the theoretical concepts with empirical application. Zorin OS, which is an Ubuntu bake off with a Linux kernel, showcases a powerful, efficient and a user-friendly OS especially designed for Windows founders. Zorin OS contains a marked characteristic of the Completely Fair Scheduler (CFS) which guarantees that every process gets fair allocation of the CPU, thus enhancing responsiveness and smoothness when multitasking. Thread management is done with the Native POSIX Threading Library (NPTL) which allows a higher degree of multithreading – it is a must for modern applications. Mutexes and futexes and other synchronization tools plays important role in maintaining consistency of data and coordination among processes. Also in the case study, we looked at how deadlocks detection and resolution are done on Zorin OS and how these methods make the system reliable under complex workloads. In the nutshell, this project showed that Zorin OS not only fulfills technical requirements but also provides a practical ground for analyzing the operation of systems in their authentic environments. Its performance, stability, security, and user-friendliness make it an asset to both end users and developers.