The *Unbreakable* System: A NixOS Case Study

CSCI 312

Instructor: Dr. Zubaidah Alhazza
April 29, 2025

Hazim Kaloub
2023005883
2023005883@aurak.ac.ae

Adnan Shanbour
2022005845
2022005845@aurak.ac.ae

Ahmed Gani
2022005563
Ahmed.amrangani@aurak.ac.ae

Saif Alrowahi
2021005253
2021005253@aurak.ac.ae

Mostafa Harbia
2022005486
Mostafa.harbia@aurak.ac.ae

Marwan Sayed
2021005030
marwan.sayed@aurak.ac.ae

Department of Computer Science and Engineering
American University of Ras Al Khaimah

2024–2025

Contents

Table 1: Contributions

Member	Contributions
Hazim Kaloub	Introductory, Conclusionary, and Appendices
	Implementation Chapter
	Section 1.6 [File Management]
Adnan Shanbour	Section 1.3 [Synchronization]
	Section 1.4 [Deadlocks]
Ahmad Gani	Section 1.7 [Legal & Ethical Issues]
	Appendices
Mostafa Harbia	Section 1.1 [Process & Threads]
Saif Alrowahi	Section 1.2 [Process Scheduling]
Marwan Sayed	Section 1.5 [Memory Management]

Collaboration Statement

This report was collaboratively written by the team members. We used GitHub as our primary platform for collaboration, where we created a private repository to manage our work. Each member contributed to different sections of the report, and we regularly reviewed each other's contributions to ensure consistency and quality. We also held weekly meetings to discuss our progress and address any challenges we encountered. The final report is a collective effort that reflects the contributions of all team members.

Abstract

This paper dives deep into **`NixOS`**: a declarative Linux operating system that redefines system management through its purely *functional* approach. It provides a comparative analysis of traditional Linux and NixOS methodologies across process and thread management, synchronization, memory, and file systems. We will dive into both the **`traditional Linux approach`** and the **`functional NixOS approach`** when applicable. Additionally, the study addresses legal and ethical concerns, including licensing, Telemetry[!], and privacy. Furthermore, the study will include an installation guide to NixOS, whether you would like to install NixOS on a VM or on bare-metal. Lastly, there will be a section for monitoring resources and tracing system calls on NixOS.

Author's Note: The mark "[!]" entails that the preceded term is defined in the Glossary.

Introductory

0.1 Introduction

An operating system (OS) is the most fundamental piece of software in computer hardware. It manages all the resources of computer hardware and enables communication between the computer hardware and application programs. Some operating systems such as *Windows* and *MacOS* are proprietary, meaning the source code is not available to the public. While other operating systems such as *Linux* and *Unix* are open source, which means the source code is available to the public and thus can be analyzed and developed by whoever wills. This open-source nature of Linux was a key motive for us to choose it as our target operating system to research for this case study. We decided to limit our choice to Linux distributions because accessing the source code is crucial for us to analyze how things work under the hood.

0.2 What is Linux

When people mention Linux, they often refer to the whole Linux operating system. In reality, Linux is a free and open-source kernel written in C by Finnish student Linus Trovalds on September 17, 1991. The GNU packages by Richard Stallman combined along with the Linux Kernel by Linus Torvalds is the "GNU/Linux" operating system that we know today and represents 62.7% global marketshare to server operating systems and 4% to desktop operating systems. Given Linux's open source nature, anyone with enough passion can develop their own *distribution* of Linux. A Linux distribution (distro) is a special configuration of Linux, often tailored for certain use-cases or target audience. Examples of Linux distros include Ubuntu, Debian, NixOS, Arch Linux, and Fedora. Although all those aforementioned distros use the same Linux kernel, each of them may give the user a completely unique experience. The reason for that is the fact that distros may feature different Desktop Environments such as KDE[!] or GNOME[!], Package Managers such as pacman[!] or apt, Window Managers such as Hyprland[!] or i3wm[!]; different combinations of these tools can provide the unique experiences that Linux is known for.

0.3 Why NixOS

NixOS is a distribution of Linux that is built around the Nix package manager. To understand why we chose NixOS, we need to look into Nix's philosophy as a purely functional package manager where every package is built from source in a declarative manner and stored in a unique path; this is discussed in detail in 1.6.2. Nix was created by Eelco Dolstra as a PhD student at Utrecht University. Subsequently, NixOS was created by Armijn Hemel as part of his Master's thesis in 2006. NixOS uses the Nix ideology to build an entire operating system where the system configuration is also built and managed by Nix declaratively.

OS Analysis

1.1 Process and Threads

1.1.1 Processes and Threads in the Linux Kernel

In Linux-based systems (including NixOS) processes and threads are implemented using a unified abstraction model based on the concept of **tasks**. At the kernel level, both processes and threads are represented as tasks, each identified by a unique **task_struct** data structure in memory. The kernel does not distinguish between threads and processes in the way that some other operating systems do; instead, it provides a flexible framework through the **clone()** system call. By adjusting the flags passed to **clone()**, the kernel can create new tasks that share or do not share specific resources such as memory space, file descriptors, or signal handlers. This design allows the Linux kernel to treat threads as lightweight processes, a model sometimes referred to as the "1:1 thread model," where each thread is scheduled independently by the kernel as if it were a separate process.

```c
/* Prototype for the glibc wrapper function */

#define _GNU_SOURCE
#include <sched.h>

int clone(typeof(int (void *_Nullable)) *fn,
          void *stack,
          int flags,
          void *_Nullable arg, ...
          /* pid_t *_Nullable parent_tid,
             void *_Nullable tls,
             pid_t *_Nullable child_tid */ );
```

clone() Implementation in the Linux Kernel

Every task in the Linux kernel — whether it is a full-fledged process or a thread — is tracked in the scheduler and managed individually. Each task is assigned a **Process Identifier (PID)**. However, in the case of threads, the PID may differ from the **Thread Group Identifier (TGID)**, which represents the main thread of a process. Threads created within the same process share the same TGID but have separate PIDs for internal kernel tracking. This flexible process model makes Linux highly suitable for both multi-process and multi-threaded applications.

1.1.2 Inheritance of Linux Process Model in NixOS

Since NixOS is built atop the Linux kernel, it inherits this process and thread model directly. All running programs in a NixOS system (whether invoked by users, background services, or system Deamon[!]) are scheduled as Linux tasks. However, the declarative and immutable philosophy of NixOS introduces a higher level of determinism in process-related configurations. System services in NixOS are managed using **systemd**, which itself relies on the kernel's process and thread

mechanisms. Each service is defined declaratively in the system's configuration file and then spawned as a separate unit, backed by one or more Linux tasks. This approach governs service definitions, permissions, dependencies, and runtime behavior through reproducible configurations. For example, a system daemon running on NixOS is instantiated with predictable resource limits, sandboxing behavior, and environment variables, all defined within the **`configuration.nix`**[!] file [see 1.6.2]. This removes ambiguity and the risk of runtime drift common in more imperatively managed systems.

1.2 Process Scheduling

Process scheduling in NixOS is underpinned by the Linux kernel's Completely Fair Scheduler (CFS[!]), which governs the distribution of CPU time across all runnable tasks. The scheduler's philosophy is grounded in fairness, attempting to emulate an ideal multitasking processor where every process receives an equal share of CPU time proportional to its priority. This is achieved by continuously tracking each task's **`vruntime`**, a virtual runtime that accumulates as the task executes. Processes with lower **`vruntime`** values are prioritized over others, ensuring that no single task dominates the CPU and that all runnable tasks make forward progress. CFS employs a red-black tree to maintain an ordered set of runnable tasks, using each process's **`vruntime`** as the key. This data structure allows for efficient insertion, deletion, and lookup, maintaining the performance of the scheduler even when managing a large number of tasks. The leftmost node in the tree, representing the process with the smallest **`vruntime`**, is selected as the next task to execute. This selection occurs in the function **`pick_next_task_fair`**, one of the core components of the CFS implementation found in the kernel source file **`fair.c`**

```
static struct task_struct *pick_next_task_fair(struct rq *rq) {
  struct cfs_rq *cfs_rq = &rq->cfs;
  struct sched_entity *se = pick_next_entity(cfs_rq);
  if (!se) return NULL;
  return task_of(se);
}
```

Simplified logic from pick_next_task_fair

In this function, **`pick_next_entity`** traverses the red-black tree to select the leftmost node: the most under-served task. The result is then transformed into a **`task_struct`** pointer, which represents the task to be scheduled next. NixOS inherits this scheduling behavior directly from the Linux kernel, but adds flexibility through its declarative configuration model. System administrators can customize scheduler parameters such as **`sched_latency_ns`**, **`sched_min_granularity_ns`**, and others by setting kernel tunables within the system configuration file, typically **`/etc/nixos/configuration.nix`**. These options are applied consistently across rebuilds, allowing users to tailor scheduling behavior to suit specific use cases; depending on whether the system is being optimized for desktop responsiveness, real-time latency, or server throughput.

```
boot.kernel.sysctl = {
"kernel.sched_latency_ns" = "6000000";
"kernel.sched_min_granularity_ns" = "750000";
```

```
};
```

Example of configuring scheduler parameters in NixOS

Ultimately, process scheduling in NixOS reflects a synergy between the deterministic design of the system and the fairness-driven dynamics of the Linux scheduler. Through a combination of kernel-level mechanisms and user-space configuration, NixOS provides a predictable and adaptable environment for managing CPU-bound workloads.

1.3 Synchronization

In multi-processing systems, when multiple concurrent processes execute and update shared resources, the operating system needs to preserve the order of execution to achieve correct results. This requires interacting processes to execute in a coordinated manner. Process synchronization is the procedure that achieves the desired coordination. Process synchronization involves the coordination and control of concurrent processes to ensure correct and predictable outcomes. Its primary purpose is to prevent race conditions, data inconsistencies, and resource conflicts that may arise when multiple processes access shared resources simultaneously. As NixOS is a Linux-based operating system, it leverages the Linux kernel for low-level system operations, including process synchronization and concurrency control. At the kernel level, NixOS does not introduce new mechanisms for handling critical sections or race conditions; instead, it inherits and utilizes the well-established synchronization primitives provided by the Linux kernel. These mechanisms include (but are not limited to):

1.3.1 Spinlocks

Spin locks (sometimes spelled spinlocks) are similar to semaphores, but they force the process that tries to acquire an already locked spin lock, to busy wait. It means that the code check in a loop if the spin lock is unlocked. Spin locks are used for protecting bigger than a single variable resources, like queues, lists and other data structures. They ensure operations on such resources are indivisible, which means that when one such operation is performed on a given shared resource, no other indivisible operation on this resource can be started, until the first one finishes. Spin locks are variables of the structure **`spinlock_t`**. The API of spin locks consists of several functions and macros such as: **`define_spinlock(name)`** that declares and initializes spin lock of a given name, **`spin_lock()`** which acquires (locks) the spin lock, and many others. Now, many resource sharing issues in the kernel can be expressed in the terms of the readers-writers problem. Linux provides a special version of spin locks for solving the readers-writers problem (also called the readers-preference). These are known as Reader-Writer Spin Locks or simply R-W Spin Locks. These types of locks enhance performance by permitting multiple readers simultaneously but ensuring only one writer at a time. Its API has functions like: **`read_lock()`** that acquires the R-W spin lock for a reader. The function **`rw_is_locked()`** returns true if the R-W spinlock is already acquired.

1.3.2 Mutexes and Semaphores

A mutex allows only one process at a time to access a shared resource. The mutex object allows all the processes to use the same resource but at a time, only one process is allowed to use the resource. Mutex uses the lock-based technique to handle the critical section problem. Semaphore is an integer variable S, that is initialized with a finite number instances of each resource present in the system. It uses two functions to change the value of S which are **`wait()`** and **`signal()`**. Both of these functions are used to modify the value of semaphore, but the functions allow only one process to change the value at a particular time meaning no two processes can change the value of semaphore simultaneously. The **`wait()`** function checks for available instances, and when there is any, it acquires one and decrement the variable S. The definition of **`wait()`** is as follows:

```
wait(S) {
  while (S<=0)
  ; // busy wait
  S--;
}
```

`wait()` Definition

On the contrary, the **`signal()`** function is used to increment S when a process or a thread is no more using the resource. The definition of **`signal()`** is as follows:

```
signal(S) {
  S++;
}

```

`signal()` Definition

The original definitions of the **`wait()`** and **`signal()`** semaphore operations suffer from busy waiting. To overcome the need for busy waiting, we can modify the definition of the **`wait()`** and **`signal()`** operations as follows: When a process executes the **`wait()`** operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can block itself. The **`block`** operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute. A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a **`signal()`** operation. The process is restarted by a **`wakeup()`** operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.) Now, the **`wait()`** semaphore operation can be defined as:

```
wait(semaphore *S) {
  S->value--;
  if (S->value < 0) {
    add this process to S->list;
    block();
  }
```

```
}
```

wait() Semaphore Operation

and the **signal()** semaphore operation can be defined as:

```
signal(semaphore *S) {
  S->value++;
  if (S->value <= 0) {
    remove a process P from S->list;
    wakeup(P);
  }
}
```

signal() Semaphore Operation

1.3.3 Atomic Operations

Atomic operations provide instructions that execute atomically without interruption. Just as the atom was originally thought to be an indivisible particle, atomic operators are indivisible instructions. The kernel provides two sets of interfaces for atomic operations; one that operates on integers and another that operates on individual bits. The atomic integer methods operate on a special data type, **atomic_t**. This special type is used, as opposed to having the functions work directly on the C **int** type, for a couple of reasons. First, having the atomic functions accept only the **atomic_t** type ensures that the atomic operations are used only with these special types. Likewise, it also ensures that the data types are not passed to any other non-atomic functions. Also, the use of **atomic_t** ensures the compiler does not (erroneously but cleverly) optimize access to the value. It is essential that atomic operations receive the correct memory address, not an alias. In addition to atomic integer operations, the kernel also provides a family of functions that operate at the bit level.

1.4 Deadlocks

All the processes in a system require some resources such as central processing unit (CPU), file storage, input/output devices, etc to execute it. Once the execution is finished, the process releases the resource it was holding. However, when many processes run on a system, they compete for the resources necessary for execution. This may result in a deadlock. A deadlock occurs when multiple processes are blocked because it is holding a resource and also requires some resource that is acquired by some other process. Therefore, none of the processes gets executed. A deadlock situation can arise if the following four conditions hold simultaneously in a system:

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. If another process requests that resource, it must wait until the resource is released.

2. **Hold and Wait:** A process must hold at least one resource and simultaneously wait to acquire additional resources currently held by other processes.

3. **No Preemption:** Resources cannot be forcibly taken from a process. A resource can only be released voluntarily by the process holding it once it has completed its task.

4. **Circular Wait:** There must be a circular chain of processes `P0, P1, ..., Pn` where `P0` is waiting for a resource held by `P1`, `P1` is waiting for a resource held by `P2`, and so on, with `Pn` waiting for a resource held by `P0`.

NixOS, as a Linux-based operating system, employs various strategies to prevent and handle deadlocks, ensuring system stability and reliability. These strategies include (but are not limited to):

1. **Resource Ordering:** One common approach is to define a global order in which resources should be acquired. If all threads acquire resources in the same order, circular waits and deadlocks are prevented.

2. **Timeouts:** Rather than letting threads wait indefinitely, it sets a timeout period. If a thread cannot acquire the necessary locks within this timeframe, it releases any resources it currently holds and backs off, allowing other threads to progress.

3. **Resource Preemption:** A technique used in deadlock prevention and handling where the system forcibly takes back allocated resources from certain processes to break circular wait conditions. When a deadlock is detected or anticipated, the system selects a victim process — often based on priority or resource usage — and preempts its resources, rolling back its state if necessary. This reallocation allows other processes to proceed, resolving the deadlock.

1.5 Memory Management

Memory management is one of the most critical functions of any modern operating system. In the case of Linux, it plays a central role in managing the system's physical and virtual memory resources. The kernel is responsible for allocating memory to processes, maintaining isolation between them, preventing memory leaks, and ensuring optimal performance. The kernel achieves this through mechanisms such as paging, swapping, and caching. Linux's memory management subsystem provides granular control over how memory is allocated and reclaimed, supporting features like transparent huge pages, NUMA[!] awareness, and control groups (cgroups) to efficiently balance memory usage across workloads. Memory management in NixOS is a layered process that coordinates the efficient use of memory across the functional package management system, the underlying Linux kernel, and running applications. The approach is shaped by the system's declarative philosophy and its reliance on the Nix package manager, which treats system configuration and software installation as reproducible, isolated processes. While this model offers reliability and rollback capabilities, it also presents unique memory challenges, particularly during system evaluation and package builds. High memory usage can occur when evaluating large derivation sets or rebuilding system configurations, due to the computational cost of dependency resolution and the creation of build environments in isolation. The Linux kernel within NixOS is responsible for lower-level memory management, handling core tasks

such as virtual memory provisioning, page caching, memory paging, and memory protection. Virtual memory in this context allows the system to simulate more memory than is physically available by using storage devices as temporary memory space. The kernel's memory manager ensures that memory pages are swapped in and out of RAM as needed, depending on system load and memory pressure. When memory usage becomes critical, the kernel may invoke the out-of-memory (OOM[!]) killer, which selects and terminates processes to free up memory resources and maintain system stability. To mitigate system hangs under memory pressure, some NixOS users enable earlyoom, a daemon that proactively frees memory before the kernel OOM killer intervenes. User-space memory management in NixOS operates at the level of individual programs and services, which either rely on manual memory handling or delegate the task to automatic memory managers within programming environments. Developers working in languages like C or C++ often implement explicit memory allocation and deallocation, ensuring that data structures receive the necessary memory and that memory is released when no longer in use. Other applications, particularly those written in higher-level languages like Haskell or Python, benefit from garbage collection systems that automatically reclaim unused memory space. These systems are integrated into the language runtime and operate independently of the kernel's memory subsystem. To support performance on machines with constrained resources, NixOS allows customization of memory behavior through system configuration options. For instance, enabling compressed RAM-based swap using `ZRAMSwap`[!] permits the system to extend effective memory by compressing data stored in RAM, reducing the reliance on slower physical swap devices. Such features can be defined declaratively in the system's configuration file and applied consistently across reboots or deployments, aligning with the overall reproducibility goals of NixOS.

1.6 File Management

1.6.1 File Management in Traditional Distributions

In traditional and modern Unix-like systems, configuration files are managed *imperatively*: every action that the user or system takes to install, update, or remove software is a stateful action. In the imperative model, new versions of packages overwrite older ones. Certain packages are used/shared by other packages as dependencies. Packages are distributed throughout the filesystem hierarchy (e.g., `/etc, /usr, /bin, /var, /lib.`) In this case, upgrading one application might trigger an upgrade of another, which might cause other packages to break due to the dependency on a prior version of an upgraded application. This phenomenon is known as "DLL Hell" or "Dependency Hell". This statefulness also makes it nearly impossible to support multiple versions of the same application. If two applications depend on two different versions of the same package, we cannot satisfy both at once. A common workaround is to include the version of the package in the name directly such as: `python33` and `python34`. However, this approach does not scale well and requires explicit actions from the package manager. Furthermore, such stateful configurations upgrades are not *atomic*: during the time where a package is being upgraded, it is in an inconsistent state. If this process is interrupted or executed during that time by any means, it can cause permanent damage to the package. This problem

can escalate to the level of full operating system upgrades. For instance, if your computer gets unplugged during a traditional Linux OS upgrade (or a windows update), you are likely to end up with a corrupted system!

1.6.2 File Management in NixOS

Nix, the package manager that NixOS relies on, implements a purely *functional* model. This means that all static parts of the system (such as packages, configuration files, and build scripts) are built by declarative functions and are thus immutable. Each package is described by pure functions in a **`Derivation`**[!]: A build recipe for packages from source along with its dependencies. The output of building derivations is stored in its own unique path in the **`Nix Store`** (**`/nix/store/`**). These paths are based on a cryptographic hash of all the inputs used in the derivation. For example, consider the path: [**`/nix/store/2f5nwg9sn7cmmhqgq8xx5xchsahlkmrj-lunarclient-3.3.5`**]. This path points to a specific build of **`lunar client`**, version 3.3.5. The hash starting with [**`2f5nw`**...] represents the exact inputs used to build it. If any of those inputs were changed (due to an update, for example) a new hash would be produced, and consequently, a new path would be created.
NixOS takes Nix a step further, by building the entire configuration of the operating system in a purely functional manner. By utilizing a singular configuration file for all system tools and packages, users can alter the inputs by modifying the **`configuration.nix`** file. Additionally, modifying the **`configuration.nix`** and rebuilding the system will *not* overwrite the old configuration. Rather, it will store the old configuration(s) in **`/nix/var/nix/profiles`**, allowing rollbacks to any previous configuration. Furthermore, the upgrading process between a configuration generation and another is *atomic*. This means that the system can either be updated or not, which eliminates the possibility of your system breaking down during an update.

Example **`/nix/store`** directory would look like so:

```
/nix/store
|-- 001s0prw5w55i4f4i4bnphmd4ihwv9q1-ran_toks-59515-tex
|   `-- tex
|       `-- latex
|           `-- ran_toks
|-- 00bihryvkr02rbnvk5yfn02rx4h42lwd-tikz-bayesnet-0.1-tex
|   `-- tex
|       `-- latex
|           `-- tikz-bayesnet
|-- 00dhyjgnlmfzssz9s7ls9i2bp5sffj75-aomart-1.28-tex
|   |-- bibtex
|   |   `-- bst
|   |       `-- aomart
|   `-- tex
|       `-- latex
|           `-- aomart
|-- 00dn0wh9afy98xja69p7i404jgznfkdk-qtbase-6.8.2
|   |-- bin
```

```
{ config, pkgs, ... }:
{
  imports =
    [ # Include the results of the hardware scan.
      ./hardware-configuration.nix
    ];

  # Use the GRUB 2 boot loader.
  boot.loader.grub.enable = true;
  boot.loader.grub.version = 2;
  boot.loader.grub.device = "/dev/sda"; # or "nodev" for efi only

  time.timeZone = "Asia/Singapore";

  # Enable the OpenSSH daemon.
  services.openssh.enable = true;

  # Define a user account. 'Dont forget to set a password with passwd.
  users.extraUsers.jin = {
    isNormalUser = true;
    uid = 1000;
    extraGroups = [ "wheel" ];
  };

  # The NixOS release to be compatible with for stateful data such as
  # databases.
  system.stateVersion = "17.03";

  virtualisation.virtualbox.guest.enable = true;
}

```

example **configuration.nix** file

1.7 Legal and Ethical Issues

Operating systems are the foundation of modern computing. They manage hardware resources and deliver the core services that applications rely on to function. Operating systems raise a range of legal and ethical concerns because of their central role. These concerns include everything from software licensing and data privacy to system security and the broader societal impact of how these systems are used. NixOS, a distinctive Linux distribution known for its declarative configuration style and strong emphasis on reproducibility, faces many of these same issues. While its unique design offers advantages, it also brings with it some specific legal and ethical considerations worth exploring. But what exactly do we mean by legal and ethical issues when it comes to operating systems? Legal issues typically involve matters governed by law, such as using unlicensed software, violating open-source license terms, or failing to properly protect users' personal information. Ethical issues, in contrast, focus on questions of right and wrong. These include respecting user privacy, being transparent about how the system works, and ensuring that the technology remains accessible and fair for everyone.

1.7.1 General Legal and Ethical Issues in Operating Systems

Software Licensing and Intellectual Property

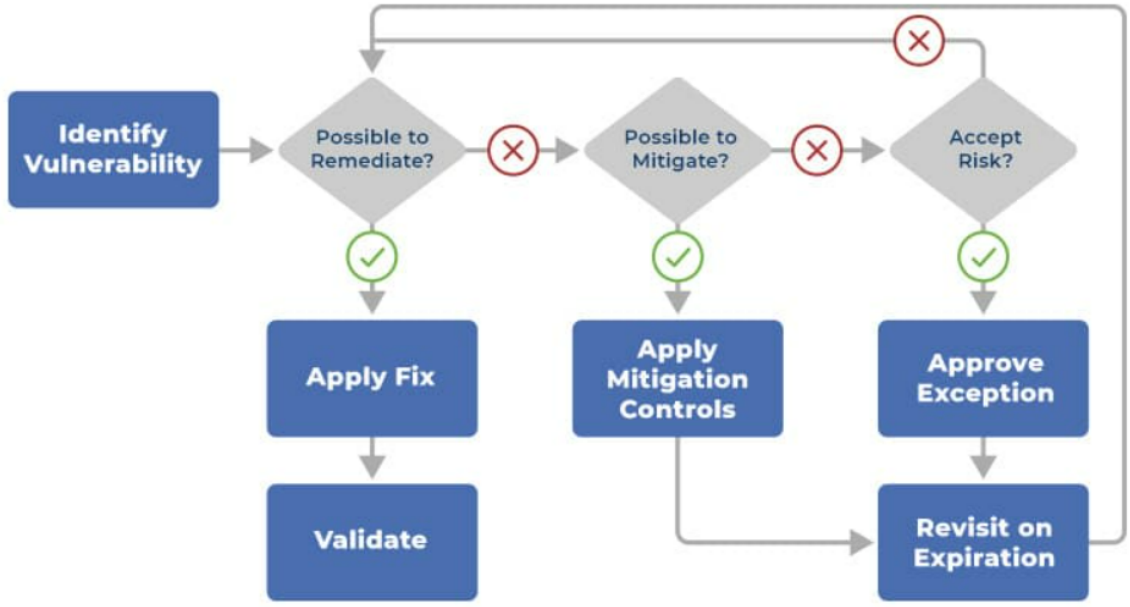
Operating systems come with specific rules regarding how they can be used, changed, and shared. These rules are defined by software licenses. Proprietary systems like Microsoft Windows have strict licensing terms that limit what users can do, especially in terms of modification or redistribution. On the other hand, open-source systems such as Linux are usually released under licenses like the GNU General Public License (GPL), which grant users the freedom to access, modify, and distribute the source code. Complying with these licensing terms is not just a good practice; it is a legal obligation. Failure to follow them can lead to serious consequences, including lawsuits and financial penalties. Copying or distributing software without permission constitutes copyright infringement and can result in major legal trouble.

Data Privacy and Protection

Operating systems handle a large amount of personal and sensitive information. This is why laws like the General Data Protection Regulation (GDPR) in Europe exist. These laws establish clear guidelines on how such data should be collected, processed, and stored. Companies that fail to comply with these rules risk facing significant fines and reputational damage. For OS developers and vendors, protecting user data is not only a legal requirement but also a matter of trust. Designing systems with strong privacy protections from the start is essential.

Security Vulnerabilities and Liability

Security flaws in operating systems can be exploited by hackers to gain unauthorized access, steal data, or cause other harm. If vendors are aware of these vulnerabilities but fail to act quickly to resolve them, they may be held legally accountable for any resulting damage. Therefore, staying ahead of security threats is critical. Timely updates and patches are not optional; they are a fundamental part of responsible software development.



Access and Digital Divide

The way operating systems are designed and distributed can either help bridge the digital divide or contribute to widening it. From an ethical perspective, developers have a responsibility to create systems that are accessible to people with disabilities and available to those in underserved communities. Open-source operating systems play an important role in this effort. Since Operating systems are free and highly customizable, they provide an inclusive alternative that can empower users around the world, regardless of their financial situation.

User Autonomy and Control

At its core, ethical computing is about giving users control over their digital environments. People deserve to understand how their systems operate, and they should have the freedom to modify or customize them to suit their needs. Unfortunately, some systems limit that control by enforcing updates without permission or collecting personal data without clear consent. Practices like these raise serious ethical concerns and can erode trust in technology.

Security vs. Privacy

One of the most difficult challenges in operating system development is balancing security with privacy. On one hand, strong protections are necessary to keep users safe. On the other hand, security measures that are too aggressive can infringe on personal privacy. Striking the right balance is not easy, but it is essential. Developers must carefully consider both priorities in order to build systems that are secure while still respecting individual rights.

1.7.2 Legal and Ethical Issues Specific to NixOS

Software Licensing in NixOS

NixOS pulls together thousands of software packages, and each one comes with its own license that dictates how it can be used, shared, or modified. The Nixpkgs repository, which is the heart of NixOS's package ecosystem, includes both free and "unfree" software, and it's quite transparent about which is which. For example, the `licenses.nix` file clearly outlines the types of licenses used and flags those that are outdated or not fully free. This level of openness helps users make informed decisions about the software they choose to install. However, keeping track of compliance across such a diverse ecosystem is no small task. Conversations within the community often highlight the challenges of ensuring that all packages respect their individual licensing requirements. In fact, concerns have been raised about whether current practices are sufficient to meet legal obligations under Free and Open Source Software (FOSS) licenses. This highlights that legal risk extends beyond individual maintainers — it could potentially affect the entire project.

Security Practices and Concerns

NixOS offers some unique security benefits thanks to its design. Features like atomic upgrades and system rollbacks help ensure that updates don't break the system. This can be a lifesaver for stability. However, while these features are valuable, they do not address all aspects of system security. Scholars and practitioners have noted that NixOS still has room to improve when it comes to comprehensive security policies. For example, Security-Enhanced Linux (SELinux), a Linux kernel security module that enforces strict access controls to protect system resources, isn't officially supported in NixOS. This may be a drawback for users or organizations that require mandatory access controls for high-security environments. Another concern is the reliance on pre-built binaries and containers. While these make installations faster and more consistent, they can introduce security challenges. Since these components are built externally and delivered as complete packages, Nix may not always be able to inspect them deeply enough to detect embedded vulnerabilities or outdated libraries in time.

Governance and Community Dynamics

The way NixOS is governed has been the subject of ongoing debate within the community. A key concern is the lack of clarity around who is actually in charge. On one hand, there's the NixOS Foundation, a legally registered organization based in the Netherlands. On the other hand, there's the Steering Committee, a more informal group focused on technical direction. The issue is that the Foundation holds the legal and financial authority, while the Steering Committee does not have any official power. This division has created confusion and even friction, especially when the Foundation makes decisions — such as handling funds or setting policies — without clear input from the wider community or the Steering Committee. Many contributors feel that this lack of transparency and accountability risks alienating those who help keep the project going.

Ethical Sponsorship and Industry Associations

The acceptance of sponsorships from defense and military-related organizations has stirred significant ethical discussions within the NixOS community. One notable example involves Anduril Industries, a U.S.-based defense contractor known for developing autonomous surveillance systems, AI-powered drones, and border security technologies. Anduril reportedly uses NixOS internally to manage and deploy its software infrastructure, and at one point, the company attempted to sponsor NixCon, the official NixOS community conference. This sponsorship offer sparked strong backlash from members of the open-source community, many of whom opposed it on ethical grounds. The main concern was that accepting money from a company primarily involved in military technology and surveillance clashed with the values many contributors hold, such as peace, civil liberties, and the non-violent use of technology. Such associations may risk damaging the community's reputation, compromise its ethical stance, and push away contributors who are uncomfortable with military or surveillance-related applications. This case also underscores broader concerns regarding how open-source projects should approach sponsorship. This is especially important when sponsors may use the technology in ways that are ethically troubling. While financial support is crucial for conferences and development, many in the community believe decisions about sponsorships should involve transparency, ethical review, and open public discussion.

Handling of Security Vulnerabilities

The discovery of security vulnerabilities in NixOS has raised concerns about how the project manages its response procedures and communicates with users. A notable example was a critical flaw found in Nix 2.24 that allowed potential privilege escalation. In this case, an attacker could gain more access than they were supposed to, creating a serious security risk. This incident underscored the importance of timely patches, clear vulnerability reports, and proactive communication with users. In open-source projects like NixOS, the community expects not just quick technical fixes but also open coordination with downstream users and maintainers. This includes publishing advisories and clearly listing affected versions. Handling vulnerabilities properly is essential for maintaining user trust, ensuring system stability, and keeping the environment secure.

Data Privacy and Telemetry

The integration of software with built-in telemetry features into NixOS has sparked ongoing debates about user privacy and whether the system complies with data protection laws like the GDPR. In this context, telemetry refers to the automatic gathering and sharing of usage data — such as system performance, behavior, or how users interact with the software — which is sent back to developers. One example that raised concerns was Firefox, a widely used browser in NixOS, which had telemetry enabled by default. This meant data could be sent without users clearly agreeing to it. Such practices raised alarms about violating GDPR regulations, which require users to be fully informed and to give explicit consent before any personal data is collected. In response, the NixOS community began discussing ways to make telemetry practices

more transparent, ensure users are properly informed, and offer easy options to opt out. These efforts aim to protect user control while remaining legally compliant.

Code Attribution and Intellectual Property

There have been allegations of improper code attribution in Nix-related projects, raising both legal and ethical concerns. In the open-source world, code attribution means giving proper credit to the original authors when their code is reused or modified. This is not just a matter of courtesy — it is a core requirement of many open-source licenses. Failure to credit contributors properly can result in legal conflicts, license violations, and a breakdown of trust within the community. Ethically, it also undermines the values of transparency, fairness, and mutual respect that are central to open-source culture. For these reasons, it is crucial for NixOS and similar projects to establish clear guidelines and adopt tools that help track contributions. This ensures that contributors are recognized and the integrity of the project is maintained.

Non-Compliance with Filesystem Hierarchy Standard (FHS)

NixOS takes a different approach from most Linux distributions by not adhering to the traditional Filesystem Hierarchy Standard (FHS[!]), which can sometimes cause compatibility issues. The FHS is a set of conventions that outline where key system files and directories should be located, such as `/bin`, `/etc`, and `/usr`, and it is widely followed across the Linux ecosystem. NixOS, however, uses a unique, declarative structure for managing packages. As a result, those familiar directory paths don't always exist in the expected way. While this model offers strong benefits in terms of reproducibility and isolation, it can cause problems when software expects a standard FHS layout. This can be a stumbling block for users and developers, especially when running applications that were not designed with NixOS in mind. Consequently, this divergence has led to ongoing discussions about how to balance innovation with the practical need for compatibility.

Implementation

2.1 Installing NixOS

2.1.1 Minimum Requirements

Before installing NixOS, make sure your system (either a VM or Bare-Metal) meets the minimum requirements:

- **Processor**: `64-bit dual-core processor`
- **Storage**: `10GiB`
- **RAM**: `1GiB`

2.1.2 Installing NixOS on a VM

Here is a detailed video on how to install any Linux-based operating system on a virtual machine using VirtualBox

2.1.3 Installing NixOS on Bare-Metal

Installing NixOS or any Linux distribution is fairly similar to installing it on a VM, ~~with a slight chance of permanently wiping out all your data~~. After downloading the ISO Image[!] from nixos.org/download, you need to flash it to a usb drive which converts it into a bootable usb. You can use a software such as Rufus to flash the iso image onto a usb drive. Once you do that, simply reboot your device with that usb plugged in and go through the graphical installation tool.

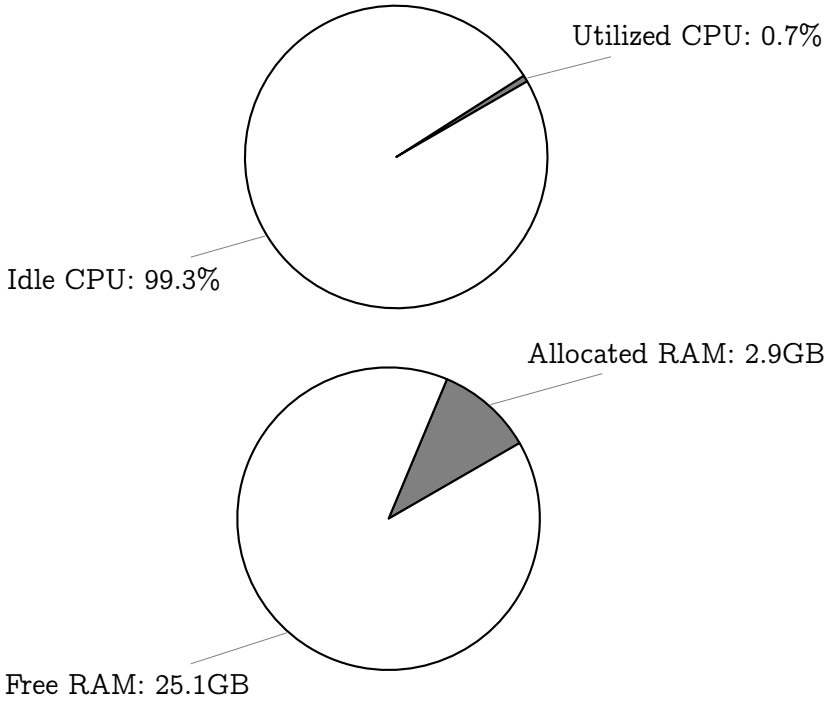
2.2 Monitoring Resources

In this section, we will use resource monitoring tools provided by the KDE desktop environment. We will be using Hazim's installation of NixOS on his laptop; his system specifications are as follows:

- **CPU**: 12th Gen Intel i5-12450HX
- **GPU**: NVIDIA GeForce RTX 2050
- **RAM**: 28GB DDR5
- **Disk**: 512GB M.2 SSD

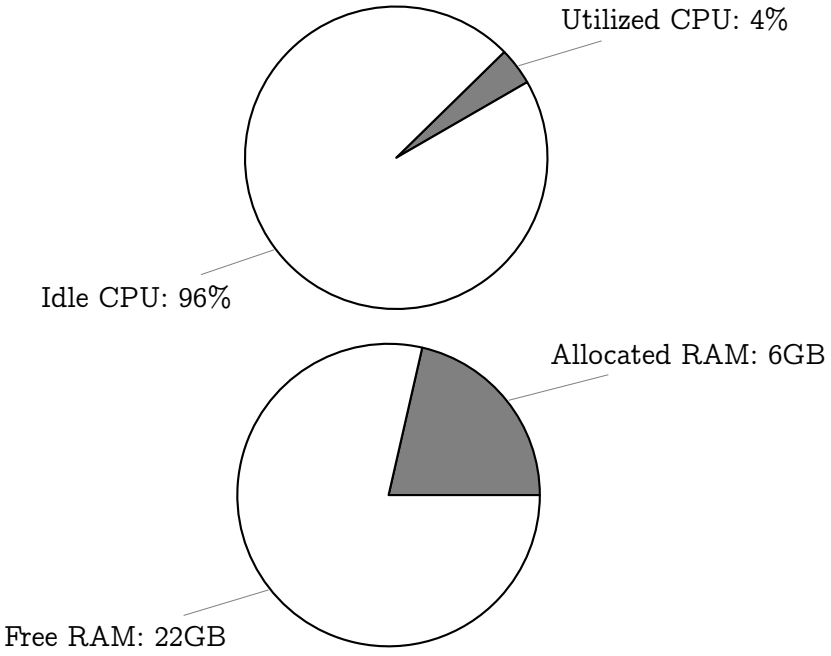
2.2.1 Idle Consumption

Running NixOS with the KDE desktop environment and necessary software only.



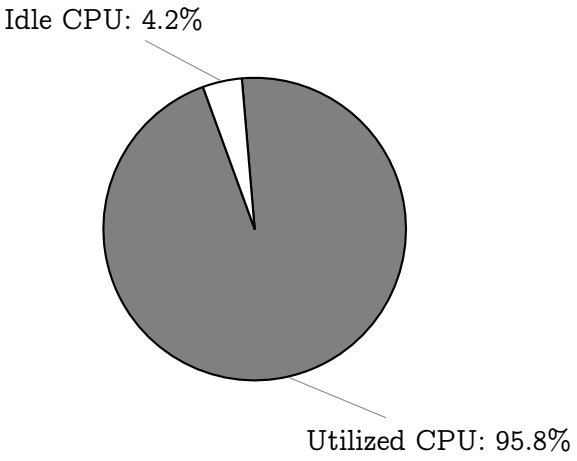
2.2.2 Average Work Consumption

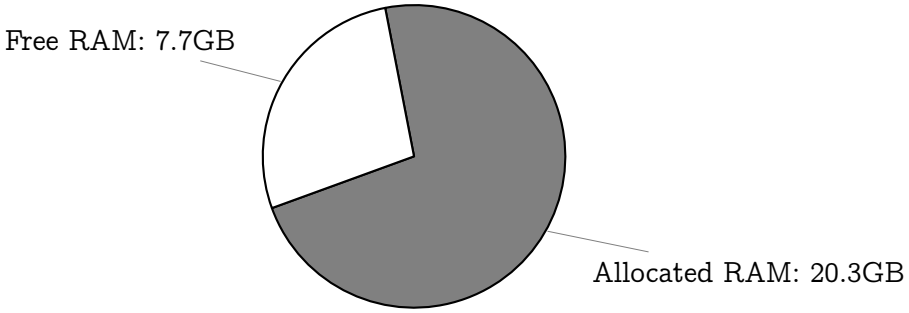
Performing everyday tasks such as browsing the web, coding in Visual Studio, using Discord, and operating the terminal.



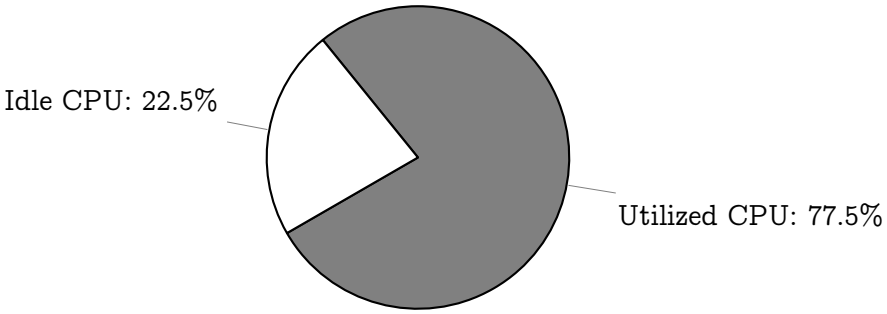
2.2.3 Stress Testing

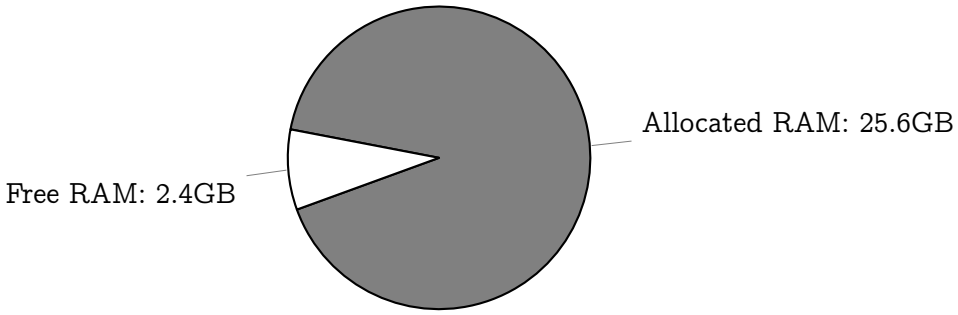
In this section, we attempted to reach the highest value of RAM and CPU by running an absurd number of lightweight tasks and a few heavy tasks.
To reach the desired CPU utilization, we had to run vivaldi browser with over 100 tabs and constantly refresh them, alongside the applications used in average-work tasks.





To reach the desired RAM Allocation, we had to add an instance of minecraft running over 250 in-game modifications





2.3 System Calls

System calls are the interface between user-space programs and the Linux kernel. On NixOS, as with other Linux distributions, system calls can be traced using tools like *strace* for user-level inspection and *ftrace* for in-kernel tracing. This section demonstrates how to trace a simple system call using both tools.

2.3.1 Creating a Test Process

To trace system calls meaningfully, we begin with a simple program that invokes a system call. The following C program calls `getpid()`, which retrieves the process ID of the running process.

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    printf("PID is: %d\n", getpid());
    return 0;
}
```

Compile the program using `gcc` (available via `nix-shell -p gcc`):

```
gcc getpid.c -o getpid_example
```

2.3.2 Tracing Using *strace*

strace intercepts and records system calls made by a process. It is useful for observing user-level interaction with the kernel.
After running the program with *strace*:

```
strace ./getpid_example
```

the output includes the following line:

```
getpid() = 16580
```

This confirms that the `getpid()` syscall was made and returned the process ID. *strace* is effective for high-level debugging and logging of syscalls, but does not provide low-level kernel tracing or function call information.

2.3.3 Tracing Using *ftrace*

ftrace is an in-kernel tracing framework that provides detailed information about function calls within the kernel, including system call handlers. In order to use *ftrace*, we need to make a few preperations:
First, ensure the `debugfs` filesystem is mounted:

```
sudo mount -t debugfs none /sys/kernel/debug
```

Then prepare *ftrace* to trace the syscall:

```
cd /sys/kernel/debug/tracing
echo > trace
echo function > current_tracer
grep getpid available_filter_functions
```

From the list, identify the appropriate syscall function name (e.g., `__do_sys_getpid`), and set it as the filter:

```
echo __do_sys_getpid | sudo tee set_ftrace_filter
echo 1 | sudo tee tracing_on
```

Run the test program:

```
./getpid_example
```

Then immediately disable tracing and read the results:

```
echo 0 | sudo tee tracing_on
cat trace
```

The output shows the following trace line:

```
getpid_example-16580 [008] ..... 15903.006073: __do_sys_getpid <-do_syscall_64
```

The **`getpid_example`** represents the name of the process making the syscall, while **`16580`** indicates the process ID (PID) of that process. The **`[008]`** shows the CPU core (core 8) the process was running on. The **`.....`** is the flags field, which often includes information about priority and preemption, but it can generally be ignored for most tracing purposes. The timestamp **`15903.006073`** shows the event's occurrence time in seconds since boot, with the event happening at 15903.006073 seconds. The function being traced is **`__do_sys_getpid`**, which is the syscall handler in this case. Finally, the **`<-do_syscall_64`** signifies that the function **`__do_sys_getpid`** was called from the **`do_syscall_64`** function, which is the entry point for handling system calls in a 64-bit Linux kernel.
To complete the trace, clean up the tracer:

```
echo nop | sudo tee current_tracer
echo > set_ftrace_filter
echo > trace
```

Conclusionary

3.1 Summary

Throughout our journey in the realm of Nix, we've discovered an inspiring ideology gifted by Eelco Dolstra to a generation of declarative operating systems, of which NixOS emerges as a modern, reproducible, and atomic Linux distribution. In the first chapter of this case study, we analyzed NixOS from the kernel to the user experience. We had a look at processes and threads and how they're managed. We observed the synchronization process and how deadlocks are managed. We reviewed the CFS algorithm and how its utilized by the Linux kernel to schedule processes. We examined how memory is managed and swapped on demand, and lastly we inspected the beautiful file system in the nix store. In the second chapter of this case study, we implemented NixOS by installing it one of our devices. We inspected and monitored its resources' consumption (specifically RAM and CPU), was highly efficient, requiring significant effort to induce stress. Lastly, we traced system calls by creating a test program and tracing it using *strace* and *ftrace*.

3.2 Final Thoughts

Personally, I (Hazim) have been using NixOS over the past 3 months, and I can confidently say that it is the best operating system I have ever used. The declarative approach to package management and system configuration has made my life so much easier. I can easily roll back to previous configurations, and the atomic upgrades ensure that my system remains stable. I couldn't enjoy these features in my previous operating system Arch Linux or any other Linux distribution. Although NixOS has a steep learning curve, it is incredibly worth it once you get the hang of it. I highly recommend it to anyone who is looking for a *stable*, reproducible, and powerful operating system.

3.3 Honourable Mentions

During the time we were researching different operating systems to write this case study on, we stumbled across Arch Linux: a minimalist, rolling release, and DIY distro. Arch was quite tempting to research, but we decided against duplicating case studies that are already written (courtesy of Nour's TuxTide).
A few other honourable mentions are: TempleOS and CollapseOS. TempleOS was written and designed solely by the schizophrenic yet intelligent programmer Terry A. Davis. CollapseOS is designed for a post-apocalypse era where technology has been doomed. It operates on microprocessors found in calculators, game consoles, and old computers without internet.
I would also like to honour Mohammed Alshamsi and Zoha Farooq for their valuable feedback on the structure, formatting, and grammar-checking of this report.

Glossary

CFS The default Linux scheduler that allocates CPU time among tasks using a red-black tree and virtual runtime tracking. 4

configuration.nix The main configuration file used to build the entire NixOS instance. 4

Deamon A background process that runs continuously to provide services such as network management, printing, or system monitoring. 3

Derivation In Nix, a build instruction for software packages that includes dependencies and build logic. 10

FHS A standard that defines directory structures and naming conventions in Unix-like systems. 16

GNOME A free and open-source desktop environment used by many Linux distributions such as Red Hat Enterprise Linux, Fedora, Ubuntu, and Mint. 2

Hyprland A 100% independent, dynamic tiling Wayland compositor that doesn't sacrifice on its looks. 2

i3wm A tiling window manager designed for X11, inspired by wmii and written in C. It supports tiling, stacking, and tabbing layouts, which are handled manually. 2

ISO Image A single archive file that contains the entire contents of an optical disk. ISO files are used to install operating systems via USB or virtual machines. 17

KDE An international free software community that develops free and open-source software. As a central development hub, it provides tools and resources that enable collaborative work on its projects. 2

NUMA A memory design where access time depends on proximity of memory to the processor, used in multiprocessor systems. 8

OOM A Linux kernel feature that terminates processes to reclaim memory when the system runs out of resources. 9

pacman pacman is a utility which manages software packages in Linux. 2

Telemetry The automatic collection and transmission of system data for performance and usage monitoring. 1

ZRAMSwap A Linux kernel module that compresses data in RAM to reduce the need for disk swap, enhancing performance. 9

Bibliography

[1] barrage.net. *barrage.net*. https://www.barrage.net/blog/strategy/the-importance-of-ethics-in-technology. 2024.

[2] chrismcdonough.substack.com. *chrismcdonough.substack.com*. https://chrismcdonough.substack.com/p/report-on-nixos-governance-discussions. 2024.

[3] chrismcdonough.substack.com. *chrismcdonough.substack.com*. https://chrismcdonough.substack.com/p/the-nixos-conflict-in-under-5-minutes. 2024.

[4] denysvitali. *Open letter to the NixOS foundation*. https://news.ycombinator.com/item?id=40107370. 2024.

[5] discourse.nixos.org. *discourse.nixos.org*. https://discourse.nixos.org/. 2024.

[6] elikoga. *A leadership crisis in the Nix community*. https://news.ycombinator.com/item?id=40199153. 2024.

[7] fossid.com. *fossid.com*. https://fossid.com/articles/open-source-license-compliance-lessons-from-two-landmark-court-cases/. 2024.

[8] kryten.mm.rpi.edu. *kryten.mm.rpi.edu*. https://kryten.mm.rpi.edu/PRES/HAPOP3062516/sb_as_nsg_j-cp_ko_ethical_os_ab_0619162100NY.pdf. 2024.

[9] learn.saylor.org. *learn.saylor.org*. https://learn.saylor.org/mod/book/view.php?chapterid=6881&id=30945v. 2024.

[10] Linux Kernel Documentation. *Completely Fair Scheduler (CFS) Design*. https://docs.kernel.org/scheduler/sched-design-CFS.html. Accessed: 2025-04-19. 2024.

[11] lolinder. *NixOS and reproducible builds*. https://news.ycombinator.com/item?id=43448745. 2024.

[12] nixos.wiki. *nixos.wiki*. https://nixos.wiki/wiki/Security. 2024.

[13] philpapers.org. *philpapers.org*. https://philpapers.org/rec/ONEEOS-2. 2024.

[14] researchgate.net. *researchgate.net*. https://www.researchgate.net/publication/330285285_Ethical_Operating_Systems_Historical_and_Philosophical_Aspects. 2024.

[15] shealevy.com. *shealevy.com*. https://shealevy.com/blog/2024/05/08/broken-promises-the-nix-governance-discussions/. 2024.

[16] stackoverflow.com. *stackoverflow.com*. https://stackoverflow.com/questions/77585228/how-to-allow-unfree-packages-in-nix-for-each-situation-nixos-nix-nix-wit. 2024.

[17] todsacerdoti. *An Overview of Nix in Practice*. https://news.ycombinator.com/item?id=38237696. 2024.

[18] todsacerdoti. *Nix 2.24 is vulnerable to (remote) privilege escalation*. https://news.ycombinator.com/item?id=41492994. 2024.

[19] Linus et al. Torvalds. *Linux Kernel Scheduler Source Code - fair.c*. https://elixir.bootlin.com/linux/v6.14.2/source/kernel/sched/fair.c. 2024.

[20] varutra.com. *varutra.com*. https://varutra.com/ctp/threatpost/postDetails/NixOS-Package-Manager-Flaw-Exposes-Critical-Vulnerability/QlBncEJOUHhCdUFld096NlZ5WVNOQT09. 2024.

[21] Steven Vaughan-Nichols. *Can the Internet exist without Linux?* https://www.zdnet.com/home-and-office/networking/can-the-internet-exist-without-linux/. 2015.

[22] vncsb.dev. *vncsb.dev*. https://vncsb.dev/posts/nixos-hacking-machine/. 2024.

[23] washu.edu. *washu.edu*. https://washu.edu/policies/guide-to-legal-and-ethical-use-of-software/. 2024.

[24] willghatch.net. *willghatch.net*. https://www.willghatch.net/blog/2020/06/27/nixos-the-good-the-bad-and-the-ugly/. 2024.

Further Readings

- Integrating Software Construction and Software Deployment by Eelco Dolstra
- *NixOS: A Purely Functional Linux Distribution* by Eelco Dolstra
- A Deep Dive into NixOS: From Configuration To Boot by Chen Jingwen
- Improving reproducibility of scientific software using Nix/NixOS: A case study on the preCICE ecosystem by Simon Hauser
- *Build systems à la carte: Theory and practice* by ANDREY MOKHOV