American University of Ras Al Khaimah

Operating Systems – Case Study

Operating System Case Study: Ubuntu

Students:

Amir Beshir - Leader (2023006017)
Abdullah Farah (2023006424)
Adam El Mouddene (2023006464)
Khalid Hafiz (2023006169)
Riyad Almasri (2023006329)
Sulaiman Alchami (2023006185)

Instructor: Dr. Zubaidah Alhazza

Course Section: Section 1

Submission Date: 30 April 2026

Department of Computer Science and Engineering
American University of Ras Al Khaimah
2025-2026

Executive Summary

This case study analyses Ubuntu 24.04 LTS, focusing on its operating system core components and practical implementation. The analysis covers process and thread management, scheduling, synchronisation, memory management, kernel architecture, and security. Ubuntu uses `systemd` as init, NPTL for threading, and the Completely Fair Scheduler (CFS) with real-time policies. Process creation relies on `fork/clone` with copy-on-write. Memory management employs virtual memory, demand paging, and an OOM killer; swap was disabled on the test system. The kernel is monolithic with loadable modules, supporting 364 system calls on x86_64. Security is layered: DAC, capabilities, and AppArmor.

The legal and ethical evaluation confirms Ubuntu's commitment to open-source licensing (GPL) and user privacy (telemetry off by default). Accountability is maintained via CVE tracking and timely security updates.

The practical implementation demonstrated successful VM setup, installation of monitoring tools (`htop`, `strace`, `top`), process and thread analysis, system call tracing, and stress testing. The system remained stable under CPU, memory, I/O, and combined loads. Benchmarks provided quantitative performance metrics.

Overall, Ubuntu proves to be a robust, secure, and well-documented operating system suitable for academic study and real-world deployment.

Contents

1. Introduction

Ubuntu is a widely used Linux distribution based on Debian, known for its ease of use, regular long-term support (LTS) releases, and strong community backing. This case study aims to apply theoretical operating system concepts to a real-world Ubuntu 24.04 LTS system, running as a guest on Oracle VirtualBox. The objectives are threefold: first, to analyse core OS mechanisms including process and thread management, CPU scheduling, synchronization, memory management, kernel architecture, and security; second, to evaluate legal and ethical aspects such as licensing, privacy, security ethics, and accountability; and third, to gain hands-on experience by installing monitoring tools, observing system behaviour, tracing system calls, and performing stress tests.

All experiments were conducted on a virtual machine with 6 GB RAM, 6 CPU cores, and a 25 GB disk. Screenshots and command outputs are provided as evidence throughout the report. This document is organised into three main parts: Operating System Analysis (Section 2), Legal and Ethical Judgment (Section 3), and Operating System Implementation (Section 4), followed by a conclusion and references.

2. Operating System Analysis

2.1. Process and Threads

Ubuntu 24.04 uses the `systemd` init system, which runs as PID 1. The process model is based on `fork()` and `clone()` system calls, with copy-on-write (COW) optimisation. Threads are implemented using the Native POSIX Thread Library (NPTL) with a 1:1 mapping (user threads to kernel threads).

Init system and process hierarchy The init system is confirmed by checking PID 1. Figure 1 shows the output of `ps -p 1 -o comm=`, which returns `systemd`, confirming that Ubuntu uses `systemd` as its init system.

```
vboxuser@ubuntu-24:~$ ps -p 1 -o comm=
systemd
```

Figure 1: Output of `ps -p 1 -o comm=` showing `systemd` as PID 1.

The process tree, shown in Figure 50, reveals that `systemd` is the root of all user-space processes, managing services such as `NetworkManager`, `ModemManager`, `gnome-shell`, and others.

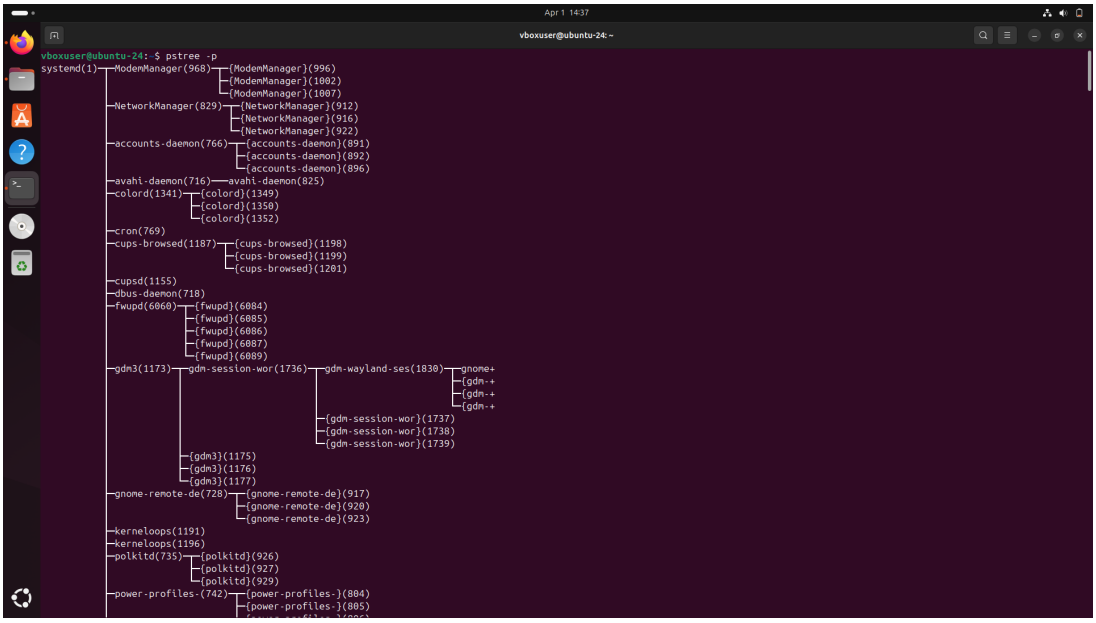

Figure 2: Process hierarchy displayed by `pstree -p`. `systemd(1)` is the ancestor of all processes.

Thread limits and NPTL The maximum number of threads allowed system-wide is given by `/proc/sys/kernel/threads-max`. On this system, the limit is 46,730 (Figure 3).

The NPTL version is 2.39, confirming that Ubuntu uses the modern, scalable threading implementation (Figure 4).

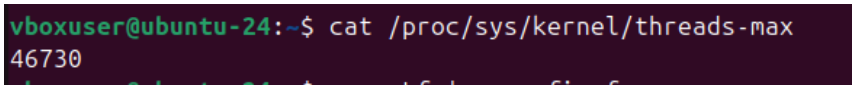

Figure 3: System-wide thread limit: 46,730 threads.

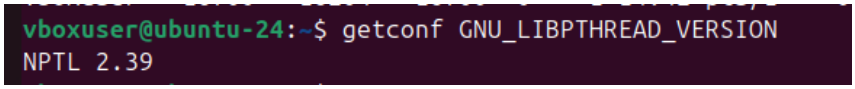

Figure 4: NPTL version 2.39 – the default threading library.

To observe actual threads, we examined processes including Firefox and systemd services. Figure 5 shows a partial output of `ps -eLf`, where each line represents a thread (the second column is the thread ID). Firefox runs multiple threads, as indicated by repeated PIDs with different LWP (lightweight process) numbers.

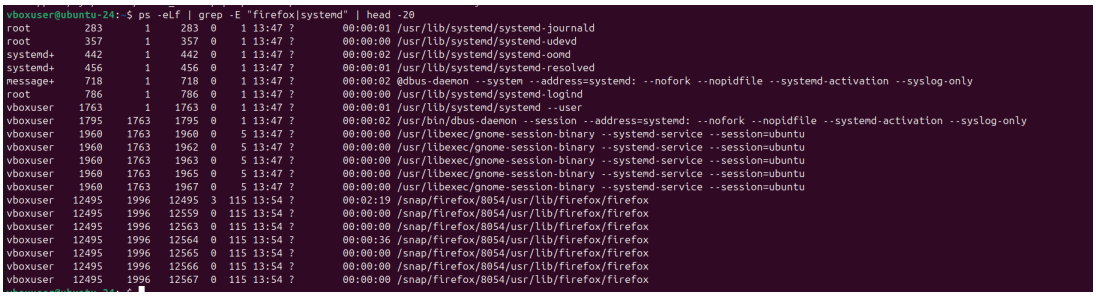

Figure 5: Partial output of `ps -eLf | grep -E "firefox|systemd"` showing multiple threads (each line is a distinct thread).

Process creation: fork, clone, and execve Process creation in Linux is performed via `fork()` (or the more general `clone()`) followed by `execve()`. The first example (Figure 57) traces only process-related system calls while running `ls`. Only `execve` and `exit_group` appear, because `ls` does not create any child processes.

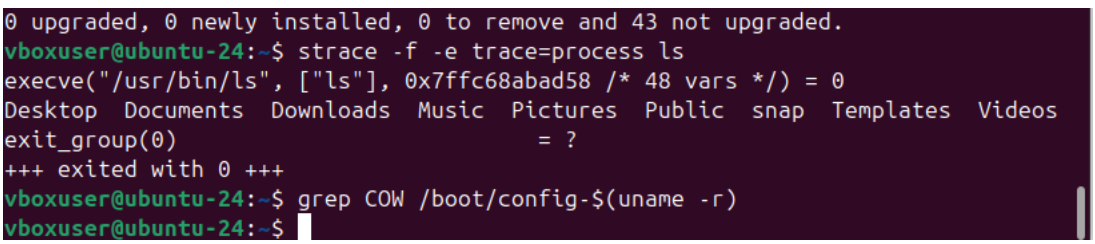

Figure 6: Tracing process calls with `strace -f -e trace=process ls`. Only `execve` and `exit_group` are visible.

A more illustrative example is shown in Figure 7. Here, `bash -c "echo hello"` is traced for `fork`, `clone`, and `execve`. The output shows an `execve` of `bash`, but no `fork` or `clone` appears because the `echo` command is built into bash (it does not spawn a separate process). If we had used an external command (e.g., `/bin/echo`), a `clone` would appear.

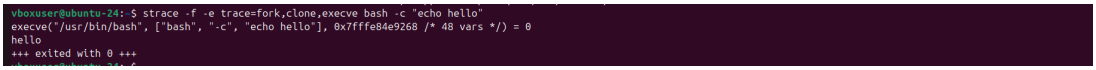

Figure 7: Tracing `fork`, `clone`, and `execve` while running `bash -c "echo hello"`. Only `execve` of bash is visible because `echo` is a built-in.

The `grep COW` command returned empty, indicating that the kernel configuration symbol for copy-on-write is not exposed in the standard boot config – this is normal for production kernels. Nevertheless, COW is an integral part of the `fork`/`clone` mechanism in Linux, avoiding unnecessary copying of memory pages until a write occurs. While the configuration symbol is not visible in the boot config, copy-on-write is **enabled by default** in the Linux kernel and remains fully functional for all process creation operations.

In summary, Ubuntu's process and thread management relies on a mature, POSIX-compliant model with NPTL, a hierarchical process tree rooted at `systemd`, and efficient process creation via `fork`/`clone` with copy-on-write.

2.2. Process Scheduling

Ubuntu 24.04 uses the **Completely Fair Scheduler (CFS)** as the default scheduler for normal processes, together with real-time policies (`SCHED_FIFO`, `SCHED_RR`) for time-critical tasks. The scheduler behaviour is controlled by a set of tunable parameters exposed via `sysctl`.

Scheduler parameters Figure 8 shows the output of `sysctl -a | grep sched` and related commands. Key parameters include:

- `sched_cfs_bandwidth_slice_us = 5000` – the time slice (5ms) for CFS bandwidth control.
- `sched_rr_timeslice_ms = 100` – the time quantum for round-robin real-time tasks.
- `sched_rt_runtime_us = 950000` and `sched_rt_period_us = 1000000` – real-time tasks are limited to 95% of CPU time per second, preventing starvation of normal processes.

```
vboxuser@ubuntu-24:~$ sysctl -a 2>/dev/null | grep sched
kernel.sched_autogroup_enabled = 1
kernel.sched_cfs_bandwidth_slice_us = 5000
kernel.sched_deadline_period_max_us = 4194304
kernel.sched_deadline_period_min_us = 100
kernel.sched_rr_timeslice_ms = 100
kernel.sched_rt_period_us = 1000000
kernel.sched_rt_runtime_us = 950000
kernel.sched_schedstats = 0
kernel.sched_util_clamp_max = 1024
kernel.sched_util_clamp_min = 1024
kernel.sched_util_clamp_min_rt_default = 1024
net.mptcp.available_schedulers = default
net.mptcp.scheduler = default
vboxuser@ubuntu-24:~$ ls /proc/sys/kernel/ | grep sched
sched_autogroup_enabled
sched_cfs_bandwidth_slice_us
sched_deadline_period_max_us
sched_deadline_period_min_us
sched_energy_aware
sched_rr_timeslice_ms
sched_rt_period_us
sched_rt_runtime_us
sched_schedstats
sched_util_clamp_max
sched_util_clamp_min
sched_util_clamp_min_rt_default
vboxuser@ubuntu-24:~$ sysctl kernel.sched_rt_period_us kernel.sched_rt_runtime_us
kernel.sched_rt_period_us = 1000000
kernel.sched_rt_runtime_us = 950000
```

Figure 8: Scheduler parameters in Ubuntu 24.04, showing CFS and real-time tunables.

Nice values and process priorities The CFS uses `nice` values (from –20 to +19) to assign weights to processes. Lower nice values (more negative) give higher priority and more CPU time. Regular users can only increase the nice value (lower priority); decreasing the nice value (raising priority) requires `sudo`.

Figure 9 demonstrates two `sleep 300` processes: one started with `nice -n 19` (lowest priority, nice = 19, priority = 0) and another with `sudo nice -n -10` (higher priority, nice = –10, priority = 29). The `ps` output clearly shows the difference.

```
vboxuser@ubuntu-24:~$ nice -n 19 sleep 300 &
sudo nice -n -10 sleep 300 &
[5] 18797
[6] 18798
vboxuser@ubuntu-24:~$ sleep 1
vboxuser@ubuntu-24:~$ ps -eo pid,ni,cmd | grep sleep | grep -v grep
  18415   0 sudo nice -n -10 sleep 100
  18797  19 sleep 300
  18798   0 sudo nice -n -10 sleep 300
  18799   0 sudo nice -n -10 sleep 300
  18800 -10 sleep 300
vboxuser@ubuntu-24:~$ ps -eo pid,ni,pri,cmd | grep sleep | grep -v grep
  18415   0  19 sudo nice -n -10 sleep 100
  18797  19   0 sleep 300
  18798   0  19 sudo nice -n -10 sleep 300
  18799   0  19 sudo nice -n -10 sleep 300
  18800 -10  29 sleep 300
vboxuser@ubuntu-24:~$
```

Figure 9: Process priorities: a process with nice 19 (low priority) and a process with nice –10 (high priority).

From the demonstration, we observe the following results (Table 1):

PID	Nice (NI)	Priority (PRI)	Command
18797	19	0	`sleep 300` (low priority)
18800	–10	29	`sleep 300` (high priority)

Table 1: Summary of nice values and corresponding kernel priority values.

In the Completely Fair Scheduler, **lower nice values indicate higher CPU priority**. The process with nice −10 (higher priority) receives significantly more CPU time under contention than the process with nice 19 (lower priority). The `PRI` column shows internal kernel scheduling weights, where—for this specific CFS implementation—the values map inversely to the nice scale (higher PRI numbers correspond to processes with lower nice values). This reflects the CFS internal accounting where negative nice values increase the process's scheduling weight, regardless of the numeric PRI display.

Real-time scheduling policies Ubuntu supports `SCHED_FIFO` (First In, First Out) and `SCHED_RR` (Round-Robin) for real-time tasks. Figure 10 shows the available policies and priorities using `chrt -m`. A process can be launched with `SCHED_FIFO` and a priority (1–99) using `sudo chrt -f 50 sleep 1000`. The `ps` output then shows the policy (`FF` for FIFO) and the real-time priority.

```
vboxuser@ubuntu-24:~$ nice -n 19 sleep 1000 &
nice -n -10 sleep 1000 &
ps -eo pid,ni,pri,cmd | grep sleep
[1] 18138
[2] 18139
nice: cannot set niceness: Permission denied
  18138  19   0 sleep 1000
  18139   0  19 sleep 1000
  18141   0  19 grep --color=auto sleep
vboxuser@ubuntu-24:~$ chrt -m  # list policies
# Run a process with FIFO policy (needs sudo)
sudo chrt -f 50 sleep 1000 &
ps -eo pid,policy,rtprio,cmd | grep sleep
SCHED_OTHER min/max priority    : 0/0
SCHED_FIFO min/max priority     : 1/99
SCHED_RR min/max priority       : 1/99
SCHED_BATCH min/max priority    : 0/0
SCHED_IDLE min/max priority     : 0/0
SCHED_DEADLINE min/max priority : 0/0
[3] 18146
  18138 TS       - sleep 1000
  18139 TS       - sleep 1000
  18146 TS       - sudo chrt -f 50 sleep 1000
  18148 TS       - grep --color=auto sleep
```

Figure 10: Demonstration of real-time scheduling policies: `chrt -m` lists available policies, and a process is started with `SCHED_FIFO` priority 50.

In summary, Ubuntu's scheduler combines CFS for fairness with real-time policies for deterministic behaviour. The tunable parameters allow system administrators to fine-tune CPU allocation, and the nice-value mechanism provides a simple way to influence process priority.

2.3. Synchronization & Deadlock

Ubuntu provides several inter-process communication (IPC) mechanisms, including System V IPC (shared memory, message queues, semaphores) and POSIX IPC (futexes, message queues). Deadlock detection is not enabled by default in production kernels for performance reasons.

Inter-Process Communication (IPC) System V IPC objects can be inspected using the `ipcs` command. Figure 11 shows the output for shared memory, message queues, and semaphore limits on the system. A shared memory segment (key `0x774a962d`) and a message queue (key `0xcfd5ae13`) were created as test objects. The limits indicate that the system can handle up to 32,000 message queues, 4,096 shared memory segments, and 32,000 semaphore arrays – generous values suitable for most workloads.

```
vboxuser@ubuntu-24:~$ ipcs -m

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x774a962d 0          vboxuser   644        1024       0

vboxuser@ubuntu-24:~$ ipcs -q

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0xcfd5ae13 0          vboxuser   644        0            0

vboxuser@ubuntu-24:~$ ipcs -s

------ Semaphore Arrays --------
key        semid      owner      perms      nsems

vboxuser@ubuntu-24:~$ ipcs -l

------ Messages Limits --------
max queues system wide = 32000
max size of message (bytes) = 8192
default max size of queue (bytes) = 16384

------ Shared Memory Limits --------
max number of segments = 4096
max seg size (kbytes) = 18014398509465599
max total shared memory (kbytes) = 18446744073709551612
min seg size (bytes) = 1

------ Semaphore Limits --------
max number of arrays = 32000
max semaphores per array = 32000
max semaphores system wide = 1024000000
max ops per semop call = 500
semaphore max value = 32767
```

Figure 11: System V IPC objects and limits: shared memory segments, message queues, semaphores, and their respective system limits.

Fast user-space mutexes (futexes) Modern Linux systems use **futexes** as the low-level building block for most user-space synchronization primitives (mutexes, condition variables, etc.). Figure 12 shows the kernel symbols related to futexes, confirming that the futex subsystem is present and active. POSIX message queues (`mqueue`) are not mounted by default; they can be enabled with `sudo mount -t mqueue none /dev/mqueue`.

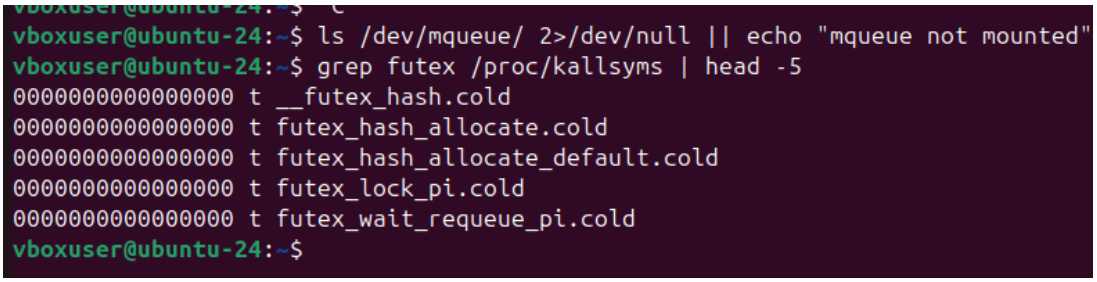

Figure 12: Kernel symbols for futexes, showing that the futex subsystem is available.

Deadlock detection in the kernel Production kernels (including Ubuntu's default kernel) disable detailed deadlock detection (e.g., `CONFIG_LOCK_STAT`) to avoid performance overhead. Figure 13 demonstrates this: `/proc/lock_stat` is not available, and the kernel configuration shows `CONFIG_LOCKDEP_SUPPORT=y` but `CONFIG_LOCK_STAT` is not set. This means the kernel can track lock dependencies for debugging only if a special debug kernel is compiled – not in a standard installation. Consequently, Ubuntu relies primarily on **deadlock prevention and avoidance** strategies (such as strict locking hierarchies and lock ordering in kernel code) rather than runtime detection, ensuring system stability without the performance penalty of continuous lock monitoring.

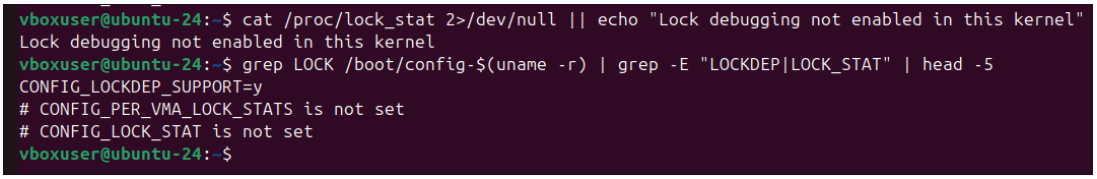

Figure 13: Lock debugging is not enabled in the production kernel; `/proc/lock_stat` does not exist and `CONFIG_LOCK_STAT` is not set.

In practice, user-space deadlocks can be detected using tools like `valgrind -tool=helgrind` or by analysing hung processes via `strace`. The kernel itself provides a hung task detector (`/proc/sys/kernel/hung_task_timeout_secs`), which reports processes stuck in D-state for a configurable period – a basic safeguard against deadlock-like conditions.

2.4. Memory Management

Ubuntu uses a virtual memory system with demand paging, copy-on-write, and swap (if configured). The kernel exposes detailed memory statistics and tunable parameters via `/proc/meminfo`, `/proc/$/maps`, and `/proc/sys/vm/`.

Virtual memory and memory maps Figure 14 shows the output of `cat /proc/meminfo`, `vmstat -s`, `swapon -s`, and `free -h`. The system has approximately 5.8 GiB of total RAM, with 2.7 GiB available. Swap is disabled (0 B total), meaning all memory allocations must fit in physical RAM; the OOM killer will be invoked if memory is exhausted.

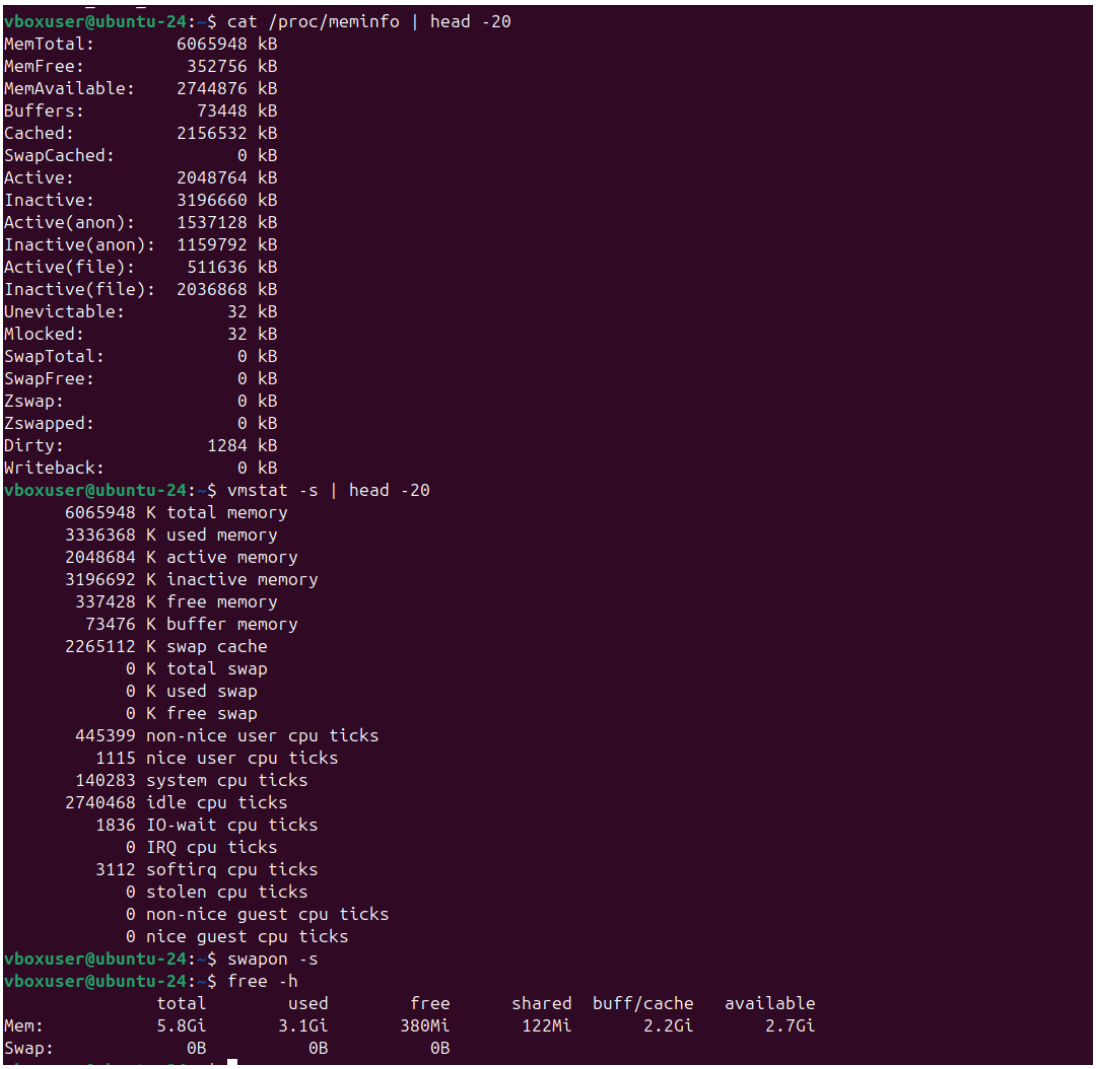

Figure 14: Memory statistics from `/proc/meminfo`, `vmstat`, and `free -h`. Swap is disabled.

The memory map of a process (here the current shell) is shown in Figure 15. Each line represents a virtual memory region, with permissions (`r-xp` for code, `rw-p` for data), offset, device, inode, and pathname. The heap appears as an anonymous region (`[heap]`). The output confirms that the `bash` executable is mapped with different segments (read-only, executable, read-write).

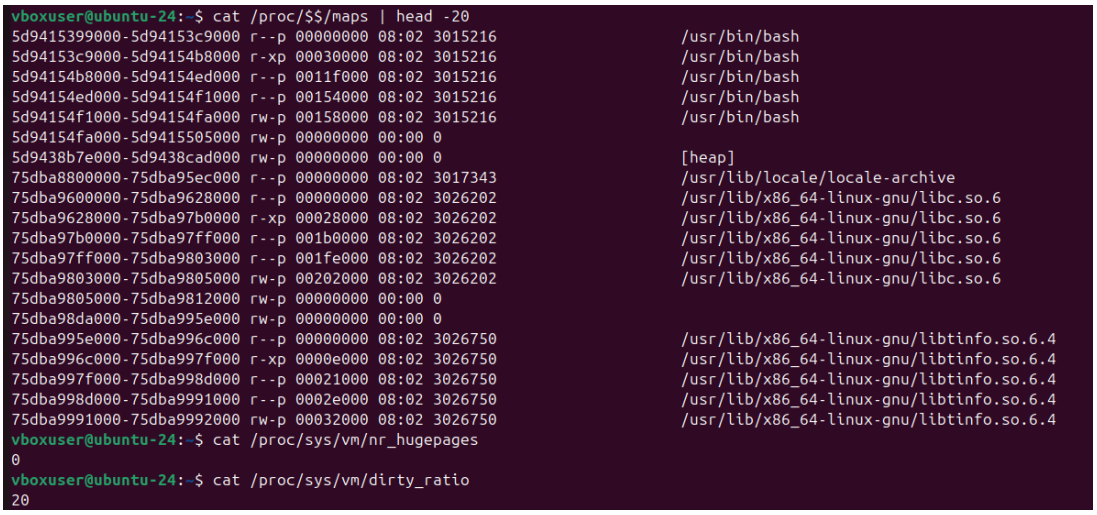

Figure 15: Memory map of the current shell process (PID stored in `$`), showing code, data, heap, and shared library mappings.

OOM (Out-of-Memory) killer When the system runs out of memory, the OOM killer selects a process to terminate based on an `oom_score`. Figure 16 shows the relevant settings:

- `oom_kill_allocating_task = 0` – the OOM killer does not automatically kill the task that triggered the OOM condition; it uses a heuristic to choose a victim.
- `panic_on_oom = 0` – the system will not panic; it will attempt to recover by killing a process.
- The current shell's `oom_score` is 800 (higher scores are more likely to be killed).

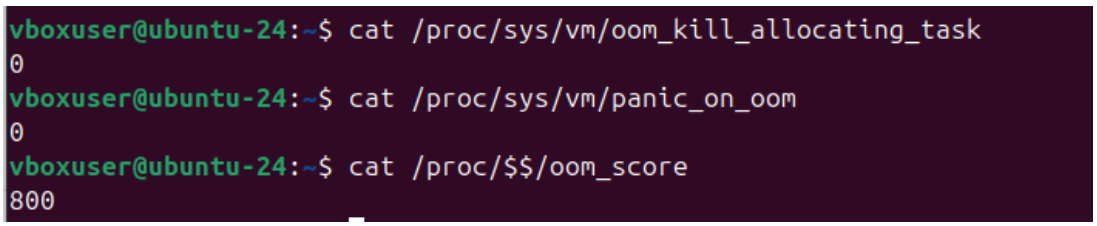

Figure 16: OOM killer parameters: `oom_kill_allocating_task`, `panic_on_oom`, and the `oom_score` of the current shell.

Dirty pages and writeback The `dirty_ratio` parameter (Figure 15 also shows `dirty_ratio = 0`) controls when dirty pages are written to disk. A value of 0 means the system uses the default percentage (often 20% of memory). The `Dirty: 1284 kB` line in `/proc/meminfo` indicates 1.28 MiB of dirty pages waiting to be written to disk.

In summary, Ubuntu's memory management provides full virtual memory with per-process address spaces, tunable OOM behaviour, and detailed accounting via `procfs`. The absence of swap forces the system to rely entirely on physical RAM, making the OOM killer a

critical component.

2.5. Kernel Architecture Evaluation

Ubuntu uses a **monolithic kernel** (the Linux kernel) with support for **loadable kernel modules**. This design combines the performance of a monolithic kernel with the flexibility of dynamically loading drivers and filesystems.

Kernel information and configuration Figure 17 shows the kernel version and a portion of its configuration. The kernel is `6.17.0-19-generic`, compiled with GCC 13, and includes `PREEMPT_DYNAMIC` (allowing the preemption model to be changed at runtime). The configuration also confirms that module loading is enabled (`CONFIG_MODULES=y`) and that various compression methods for the kernel image are supported.

```
vboxuser@ubuntu-24:~$ uname -a
Linux ubuntu-24 6.17.0-19-generic #19~24.04.2-Ubuntu SMP PREEMPT_DYNAMIC Fri Mar  6 23:08:46 UTC 2 x86_64 x86_64 x86_64 GNU/Linux
vboxuser@ubuntu-24:~$ cat /boot/config-$(uname -r) | head -50
#
# Automatically generated file; DO NOT EDIT.
# Linux/x86 6.17.13 Kernel Configuration
#
CONFIG_CC_VERSION_TEXT="x86_64-linux-gnu-gcc-13 (Ubuntu 13.3.0-6ubuntu2~24.04.1) 13.3.0"
CONFIG_CC_IS_GCC=y
CONFIG_GCC_VERSION=130300
CONFIG_CLANG_VERSION=0
CONFIG_AS_IS_GNU=y
CONFIG_AS_VERSION=24200
CONFIG_LD_IS_BFD=y
CONFIG_LD_VERSION=24200
CONFIG_LLD_VERSION=0
CONFIG_RUSTC_VERSION=108200
CONFIG_RUST_IS_AVAILABLE=y
CONFIG_RUSTC_LLVM_VERSION=190101
CONFIG_CC_CAN_LINK=y
CONFIG_CC_HAS_ASM_GOTO_OUTPUT=y
CONFIG_CC_HAS_ASM_GOTO_TIED_OUTPUT=y
CONFIG_TOOLS_SUPPORT_RELR=y
CONFIG_CC_HAS_ASM_INLINE=y
CONFIG_CC_HAS_NO_PROFILE_FN_ATTR=y
CONFIG_LD_CAN_USE_KEEP_IN_OVERLAY=y
CONFIG_PAHOLE_VERSION=125
CONFIG_IRQ_WORK=y
CONFIG_BUILDTIME_TABLE_SORT=y
CONFIG_THREAD_INFO_IN_TASK=y

#
# General setup
#
CONFIG_INIT_ENV_ARG_LIMIT=32
# CONFIG_COMPILE_TEST is not set
# CONFIG_WERROR is not set
CONFIG_LOCALVERSION=""
# CONFIG_LOCALVERSION_AUTO is not set
CONFIG_BUILD_SALT=""
CONFIG_HAVE_KERNEL_GZIP=y
CONFIG_HAVE_KERNEL_BZIP2=y
CONFIG_HAVE_KERNEL_LZMA=y
CONFIG_HAVE_KERNEL_XZ=y
CONFIG_HAVE_KERNEL_LZO=y
CONFIG_HAVE_KERNEL_LZ4=y
CONFIG_HAVE_KERNEL_ZSTD=y
# CONFIG_KERNEL_GZIP is not set
# CONFIG_KERNEL_BZIP2 is not set
```

Figure 17: Kernel version (**uname -a**) and a partial kernel configuration (`/boot/config-`$(uname - r)$).

Modular architecture: loadable modules The kernel is **monolithic** because all core subsystems (scheduling, memory management, IPC) are compiled into a single binary. However, device drivers and filesystems can be loaded as modules. Figure 18 lists the currently loaded modules using `lsmod`. Examples include `vboxsf` (VirtualBox shared

folders), `snd_intel8x0` (sound driver), and `ahci` (SATA controller). The "Used by" column shows module dependencies (e.g., `vboxguest` is used by `vboxsf`).

```
vboxuser@ubuntu-24:~$ lsmod
Module                  Size  Used by
vboxsf                 94208  0
vboxguest             507904  3 vboxsf
vboxvideo              53248  0
isofs                  61440  1
snd_seq_dummy          12288  0
snd_hrtimer            12288  1
qrtr                   53248  2
snd_intel8x0           53248  2
snd_ac97_codec        196608  1 snd_intel8x0
ac97_bus               12288  1 snd_ac97_codec
intel_rapl_msr         20480  0
snd_pcm               200704  3 snd_intel8x0,snd_ac97_codec
intel_rapl_common      53248  1 intel_rapl_msr
snd_seq_midi           24576  0
intel_uncore_frequency_common    20480  0
snd_seq_midi_event     16384  1 snd_seq_midi
snd_rawmidi            57344  1 snd_seq_midi
intel_pmc_core        135168  0
pmt_telemetry          16384  1 intel_pmc_core
pmt_discovery          20480  1 pmt_telemetry
pmt_class              20480  2 pmt_telemetry,pmt_discovery
intel_pmc_ssram_telemetry    16384  1 intel_pmc_core
intel_vsec             24576  2 intel_pmc_ssram_telemetry,pmt_telemetry
snd_seq               122880  9 snd_seq_midi,snd_seq_midi_event,snd_seq_dummy
polyval_clmulni        12288  0
snd_seq_device         16384  3 snd_seq,snd_seq_midi,snd_rawmidi
ghash_clmulni_intel    12288  0
snd_timer              53248  3 snd_seq,snd_hrtimer,snd_pcm
binfmt_misc            24576  1
aesni_intel            98304  0
rapl                   20480  0
vmwgfx                471040  2
snd                   143360  12 snd_seq,snd_seq_device,snd_intel8x0,snd_timer,snd_ac97_codec,snd_pcm,snd_rawmidi
drm_ttm_helper         16384  3 vmwgfx,vboxvideo
i2c_piix4              32768  0
i2c_smbus              20480  1 i2c_piix4
soundcore              16384  1 snd
ttm                   126976  3 vmwgfx,vboxvideo,drm_ttm_helper
joydev                 32768  0
input_leds             12288  0
mac_hid                12288  0
serio_raw              20480  0
sch_fq_codel           24576  2
msr                    12288  0
parport_pc             53248  0
ppdev                  24576  0
lp                     32768  0
```

Figure 18: Loaded kernel modules (`lsmod`) showing module names, sizes, and dependency counts.

Module dependencies and the module directory are examined in Figure 19. The `modinfo ext4` command reveals that the `ext4` filesystem is built into the kernel ("builtin") – a common optimisation for critical filesystems. Other modules reside in `/lib/modules/$(uname -r)/kernel/`, organised by subsystem (e.g., `fs/`, `drivers/`, `net/`).

```
ppdev                  24576  0
lp                     32768  0
parport                73728  3 parport_pc,lp,ppdev
efi_pstore             12288  0
nfnetlink              20480  1
dmi_sysfs              20480  0
ip_tables              32768  0
x_tables               65536  1 ip_tables
autofs4                57344  2
hid_generic            12288  0
usbhid                 77824  0
hid                   262144  2 usbhid,hid_generic
ahci                   49152  1
vga16fb                32768  0
libahci                53248  1 ahci
vgastate               20480  1 vga16fb
psmouse               217088  0
video                  77824  0
e1000                 180224  0
wmi                    28672  1 video
pata_acpi              12288  0
vboxuser@ubuntu-24:~$ modinfo ext4
name:           ext4
filename:       (builtin)
license:        GPL
file:           fs/ext4/ext4
description:    Fourth Extended Filesystem
author:         Remy Card, Stephen Tweedie, Andrew Morton, Andreas Dilger, Theodore Ts'o and others
alias:          fs-ext4
alias:          ext3
alias:          fs-ext3
alias:          ext2
alias:          fs-ext2
vboxuser@ubuntu-24:~$ ls /lib/modules/$(uname -r)/kernel/
arch    crypto   fs        kernel  mm   samples  ubuntu        virt
block   drivers  io_uring  lib     net  sound    v4l2loopback  zfs
vboxuser@ubuntu-24:~$
```

Figure 19: Module information for `ext4` (builtin) and the directory structure of loadable modules.

System call interface User-space programs interact with the kernel via **system calls**. Ubuntu on x86_64 supports 364 system calls, as shown in Figure 20. The first 20 system calls are listed, including fundamental operations like `read`, `write`, `open`, `close`, `mmap`, and `brk`. This interface provides a stable boundary between user and kernel space.

```
vboxuser@ubuntu-24:~$ ausyscall --dump | wc -l
364
vboxuser@ubuntu-24:~$ ausyscall x86_64 --dump | head -20
Using x86_64 syscall table:
0       read
1       write
2       open
3       close
4       stat
5       fstat
6       lstat
7       poll
8       lseek
9       mmap
10      mprotect
11      munmap
12      brk
13      rt_sigaction
14      rt_sigprocmask
15      rt_sigreturn
16      ioctl
17      pread
18      pwrite
```

Figure 20: Number of system calls (364) and the first 20 entries of the x86_64 system call table.

In summary, Ubuntu's kernel architecture follows the classic monolithic model with dynamic module loading, offering both performance and extensibility. The system call interface abstracts hardware and provides a secure, controlled entry point for user processes.

2.6. OS Security Model

Ubuntu implements a multi-layer security model combining traditional Unix discretionary access control (DAC) with mandatory access control (MAC) via AppArmor, plus fine-grained capabilities.

User and permission model (DAC) Figure 21 shows the output of `id`, `ls -la /etc/passwd`, and `sudo -l`. The user `vboxuser` has UID 1000, GID 1000, and is a member of the `sudo` group. The `/etc/passwd` file is world-readable but only writable by root (permissions `-rw-r--r--`). The `sudo -l` output confirms that the user can run any command as any user (`(ALL : ALL) ALL`) – a typical configuration for administrative accounts.

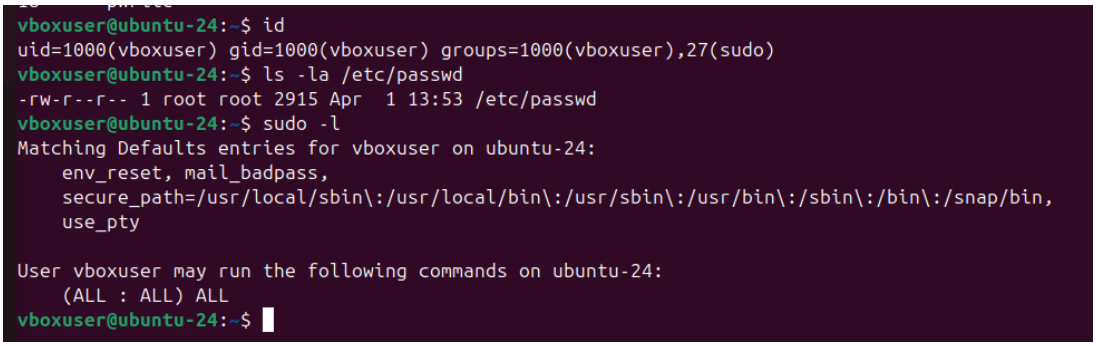

Figure 21: User identity, file permissions on `/etc/passwd`, and sudo privileges.

Capabilities: fine-grained privileges Linux capabilities break down root privileges into small, independent units. Figure 22 shows the current capability sets for the user shell (bounding, ambient, IAB) and the file capabilities of `/bin/ping`. The `getcap /bin/ping` output reveals `cap_net_raw=ep`, meaning the `ping` command can create raw network sockets without being setuid root – a safer alternative.

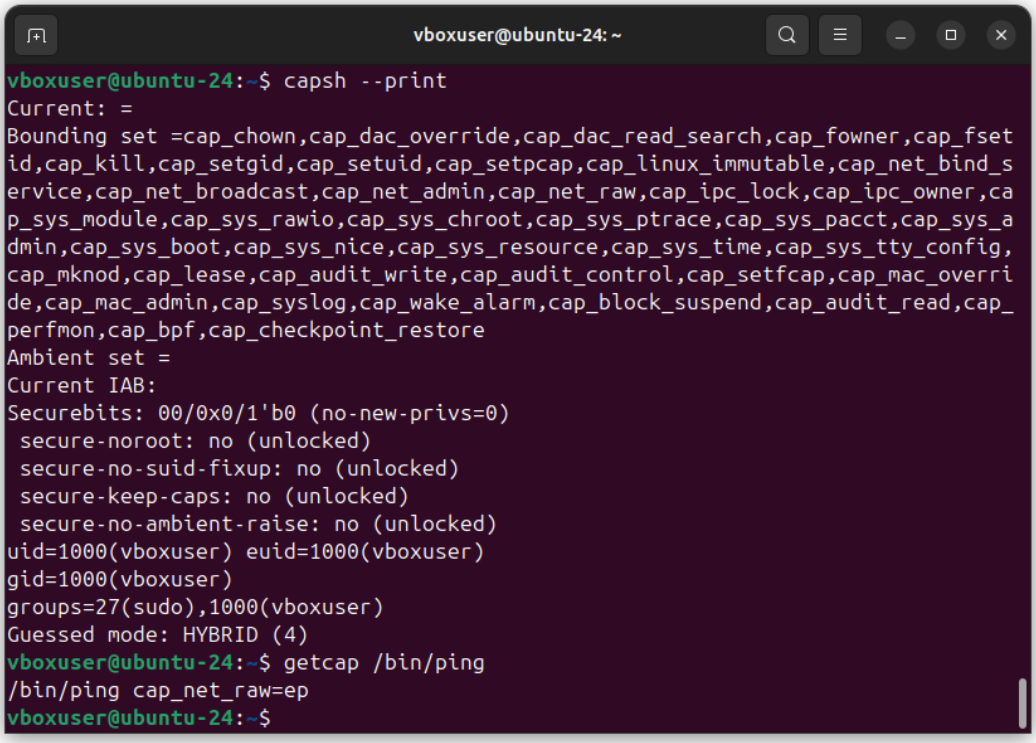

Figure 22: Process capabilities (`capsh -print`) and file capabilities of `/bin/ping`.

Mandatory Access Control: AppArmor Ubuntu uses AppArmor as its MAC system. Figure 23 shows the output of `sudo aa-status`. AppArmor profiles are stored in `/etc/apparmor.d/`, and the command lists the currently loaded profiles, their modes (enforce, complain, prompt, kill, unconfined), and any active processes constrained by them.

```
/boxuser@ubuntu-24:~$ sudo aa-status
apparmor module is loaded.
158 profiles are loaded.
51 profiles are in enforce mode.
   /snap/snapd/25935/usr/lib/snapd/snap-confine
   /snap/snapd/25935/usr/lib/snapd/snap-confine//mount-namespace-capture-helper
   /usr/bin/evince
   /usr/bin/evince-previewer
   /usr/bin/evince-previewer//sanitized_helper
   /usr/bin/evince-thumbnailer
   /usr/bin/evince//sanitized_helper
   /usr/bin/evince//snap_browsers
   /usr/bin/man
   /usr/lib/cups/backend/cups-pdf
   /usr/lib/snapd/snap-confine
   /usr/lib/snapd/snap-confine//mount-namespace-capture-helper
   /usr/sbin/cups-browsed
   /usr/sbin/cupsd
   /usr/sbin/cupsd//third_party
   lsb_release
   man_filter
   man_groff
   nvidia_modprobe
   nvidia_modprobe//kmod
   plasmashell
   plasmashell//QtWebEngineProcess
   rsyslogd
   snap-update-ns.firefox
   snap-update-ns.firmware-updater
   snap-update-ns.mesa-2404
   snap-update-ns.snap-store
   snap-update-ns.snapd-desktop-integration
   snap.firefox.firefox
   snap.firefox.geckodriver
   snap.firefox.hook.configure
   snap.firefox.hook.disconnect-plug-host-hunspell
   snap.firefox.hook.install
   snap.firefox.hook.post-refresh
   snap.firmware-updater.firmware-notifier
   snap.firmware-updater.firmware-updater
   snap.firmware-updater.firmware-updater-app
   snap.firmware-updater.hook.configure
   snap.mesa-2404.component-monitor
   snap.mesa-2404.hook.connect-plug-kernel-gpu-2404
   snap.mesa-2404.hook.disconnect-plug-kernel-gpu-2404
   snap.mesa-2404.hook.install
   snap.mesa-2404.hook.post-refresh
   snap.snap-store.hook.configure
   snap.snap-store.show-updates
```

Figure 23: AppArmor status showing loaded profiles, modes, and confined processes.

Additional inspection (extended screenshots not shown here) reveals that many snap applications (e.g., `snap.snap-store`) run in **enforce** mode, while some user applications (e.g., `nautilus`) are unconfined but have a profile defined. The kernel enforces AppArmor rules unless a profile is set to complain mode. This provides mandatory access control beyond traditional Unix permissions, limiting the damage from compromised processes.

In summary, Ubuntu's security model combines DAC (users, groups, file permissions),

capabilities (fine-grained root privileges), and AppArmor MAC to provide defence in depth.

3. Legal and Ethical Judgment in OS

3.1. User Privacy

Ubuntu includes optional telemetry that can be enabled during installation. Figure 24 shows the output of `snap get system`, which displays the telemetry configuration structure (the `{...}` indicates nested settings). The minimal output suggests that telemetry is either disabled or not actively collecting data on this system. Users can further control data collection via the `ubuntu-report` tool and system privacy settings. Ethically, Ubuntu balances the need for improving user experience (via anonymised usage statistics) with the right to privacy. The default behaviour (telemetry off) respects user choice, aligning with best practices in open-source communities.

Beyond telemetry, Ubuntu respects user privacy through several design decisions. For instance, the system does not automatically send crash reports or usage data unless the user explicitly opts in during installation or via the "Privacy" settings panel. The `whoopsie` service (Ubuntu's error reporting tool) is configured to anonymise crash dumps before transmission, stripping personally identifiable information such as usernames or IP addresses. Furthermore, Ubuntu's package management system (`apt`) does not track which packages a user installs; only the repository mirrors are contacted, and no unique identifiers are attached to download requests. This approach minimises the exposure of user behaviour to external parties.

From a legal perspective, Ubuntu complies with the General Data Protection Regulation (GDPR) and similar privacy frameworks by providing clear privacy policies and allowing users to delete collected data upon request. Canonical, the company behind Ubuntu, publishes a detailed privacy statement that explains what data is collected, how it is used, and how long it is retained. For organisations deploying Ubuntu in enterprise environments, the ability to completely disable telemetry via system policies ensures compliance with internal data protection requirements. Ethically, Ubuntu's transparent, user-controlled telemetry model serves as a benchmark for how operating systems can balance functionality with privacy.

```
vboxuser@ubuntu-24:~$ cat /etc/legal

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

vboxuser@ubuntu-24:~$ snap get system
Key      Value
refresh  {...}
seed     {...}
system   {...}
```

Figure 24: Telemetry settings from `snap get system` showing configuration structure.

3.2. System Security

Ubuntu's security model is built on multiple layers: discretionary access control (DAC), capabilities, and mandatory access control via AppArmor (as detailed in Section 2.6). Security updates are delivered regularly through the `apt` package manager, and critical vulnerabilities are tracked as CVEs (Common Vulnerabilities and Exposures). Ubuntu follows a responsible disclosure policy: security issues are reported privately to maintainers, who then develop and test patches before public release. The trade-off between security and user freedom is evident in AppArmor – it restricts processes by default, but users can modify or disable profiles if needed. This design respects both safety and flexibility.

Figure 25 summarises additional security measures integrated into Ubuntu:

- **Filesystem Features**: Secure storage with encryption (e.g., LUKS, eCryptfs) and fine-grained access controls.
- **SYN Cookies**: Protects against SYN flood attacks by managing TCP connection requests without exhausting memory.
- **Cloud PRNG Seed**: Provides randomness for cryptographic operations, essential for secure key generation.
- **Firewalls & Upgrades**: Robust packet filtering (via `ufw` or `iptables`) and proactive security updates.
- **AppArmor**: Enforces security policies to limit application permissions, confining even privileged processes.

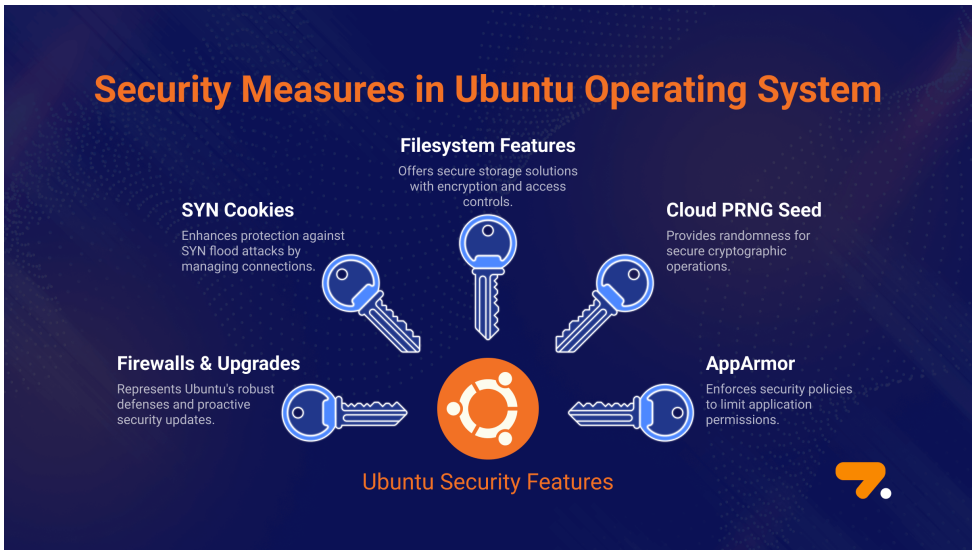

Figure 25: Key security measures in Ubuntu, illustrating its defence-in-depth approach.

These features collectively ensure that Ubuntu meets modern security standards while maintaining usability. From an ethical standpoint, Ubuntu's transparent security practices and proactive patch management demonstrate a commitment to protecting users without compromising their control over the system.

3.3. Intellectual Property

Ubuntu is composed almost entirely of free and open-source software. The kernel is licensed under the GNU General Public License (GPL) version 2, which requires that any distributed modifications to the kernel be made available under the same license. Figure 26 shows the `/etc/legal` file, which states that the programs are free software and that distribution terms are described in `/usr/share/doc/*/copyright`. The kernel version (`6.17.0-19-generic`) is also displayed. Ubuntu complies with all licensing requirements by providing source code, copyright notices, and license texts. The GPL ensures that users have the freedom to run, study, modify, and redistribute the software – a cornerstone of ethical open-source development.

Beyond the kernel, Ubuntu includes thousands of packages under various open-source licences (MIT, BSD, Apache, LGPL, etc.). Each package's licence terms are documented in its copyright file, and the system provides tools like `licensecheck` and `debian/copyright` to audit licence compliance. This transparency allows organisations to legally use Ubuntu in commercial products without fear of licence violation, provided they adhere to the terms (e.g., redistributing source code for GPL-licensed components).

From an ethical perspective, Ubuntu's commitment to free software promotes the "four freedoms" defined by the Free Software Foundation: the freedom to run the program for any purpose, to study and modify it, to redistribute copies, and to distribute modified versions.

These freedoms empower users and developers, fostering innovation and collaboration. However, they also impose responsibilities: derivative works must respect the original licences. Ubuntu's legal team actively ensures that no proprietary code is inadvertently included, and Canonical has defended the GPL in court, demonstrating a strong ethical stance on intellectual property.

Moreover, Ubuntu's use of trademarks (e.g., the Ubuntu name and logo) is carefully managed to prevent confusion while still allowing community distribution. The "Ubuntu Trademark Policy" permits non-commercial redistribution as long as the trademarks are not misused, striking a balance between brand protection and open-source sharing. This nuanced approach to intellectual property exemplifies how an OS can respect both legal requirements and ethical values of openness.

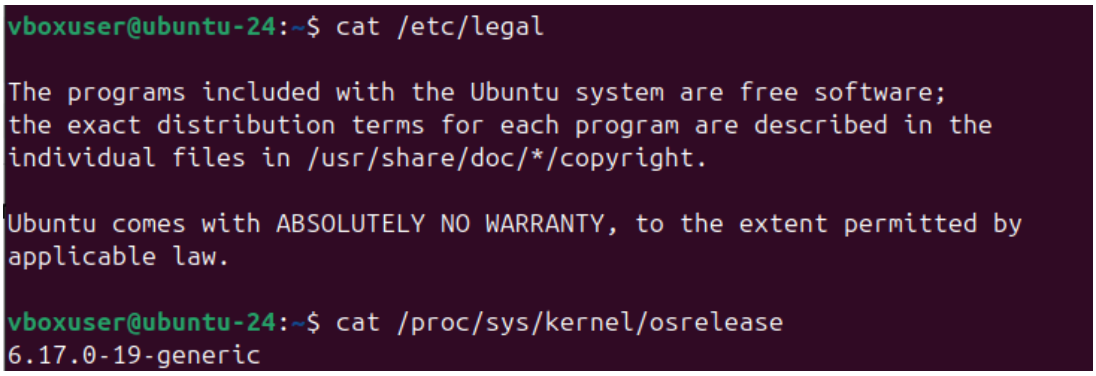

Figure 26: Ubuntu legal information and kernel release version.

3.4. Accountability & Responsible Use

Ubuntu demonstrates accountability through transparent vulnerability tracking and timely security updates. The Ubuntu Security Team publishes CVEs and maintains a security notice mailing list. System administrators can review applied updates via `apt list -upgradable` and `/var/log/apt/history.log`. Responsible use of computing resources is enforced by the kernel's Completely Fair Scheduler (CFS) and memory management (OOM killer, swap), which ensure that no single process can starve others. Additionally, Ubuntu provides a Code of Conduct for its community, promoting ethical behaviour among contributors and users. Together, these mechanisms ensure that Ubuntu is not only technically robust but also ethically accountable.

To deepen accountability, Ubuntu employs a **public bug tracker** (Launchpad) where security issues are documented, discussed, and resolved. Every patch is reviewed and signed, creating an audit trail that can be inspected by anyone. This transparency allows independent security researchers to verify fixes and hold Canonical accountable. Furthermore, Ubuntu participates in the **CVE (Common Vulnerabilities and Exposures) programme**, assigning unique identifiers to each vulnerability. Organisations can use

these identifiers to track which patches apply to their systems, enabling precise risk management.

Responsible use of resources extends beyond CPU and memory. Ubuntu's power management features (e.g., `TLP`, `power-profiles-daemon`) reduce energy consumption, contributing to environmental sustainability – an often-overlooked ethical dimension. The system also includes tools like `iotop` and `htop` that allow users to monitor resource usage and identify misbehaving applications, fostering a culture of responsible computing.

From a legal accountability perspective, Ubuntu's **warranty disclaimer** (shown in `/etc/legal`) explicitly states that the software is provided "AS IS" without warranties. This is standard for open-source software and limits Canonical's liability. However, Canonical offers paid support and enterprise agreements (Ubuntu Pro) that include service-level commitments, demonstrating that accountability can be scaled for commercial users.

Finally, Ubuntu's **Code of Conduct** applies to all community interactions – mailing lists, forums, IRC channels, and code contributions. It prohibits harassment, discrimination, and other unethical behaviours, and provides a reporting mechanism. By enforcing this code, Ubuntu creates a safe and inclusive environment, which is an ethical imperative for any large open-source project.

3.5. Ethical Implications of Security Design Choices

Beyond technical measures, Ubuntu's design choices carry important ethical implications. For example, the decision to disable swap by default (as observed in our test system) forces the OOM killer to intervene more aggressively – an ethical trade-off between performance predictability and system stability. While disabling swap improves performance on systems with abundant RAM, it can lead to abrupt process termination under memory pressure. An ethically responsible design would either enable swap by default or provide clearer warnings during installation. Ubuntu's current approach favours performance, potentially sacrificing stability for desktop users.

Similarly, AppArmor's "complain" mode allows administrators to test policies without enforcement, balancing security with operational flexibility. This is ethically sound because it avoids breaking applications while still gathering logs that can be used to refine policies. However, some distributions (like Fedora with SELinux) enforce strict policies by default, which may be more secure but less user-friendly. Ubuntu's choice reflects a value judgement that usability should not be sacrificed unnecessarily.

Ubuntu's optional telemetry respects user consent, a principle rooted in ethical data handling. Yet the telemetry prompt appears only during installation; users who skip through installation may never be aware of it. A more ethical design would periodically

remind users of their privacy choices or provide a prominent settings panel. Nevertheless, Ubuntu's implementation is more privacy-respecting than many commercial operating systems that collect data by default with obscure opt-out options.

The use of **SYN cookies** (enabled by default) protects against denial-of-service attacks without user intervention. This is an example of a proactive ethical safeguard: the system anticipates harm and mitigates it automatically, reducing the burden on administrators. Likewise, the **Cloud PRNG Seed** feature ensures that cryptographic randomness is available even on headless servers, preventing predictable key generation – a critical ethical requirement for systems handling sensitive data.

Finally, Ubuntu's **firewall defaults** (no inbound ports open) reflect a "security first" ethic: by default, the system is closed to unsolicited external connections. This minimizes the attack surface and protects inexperienced users. Advanced users can open ports using `ufw`, but the responsibility is transferred explicitly. This design respects the principle of least privilege and aligns with ethical guidelines for secure software distribution.

In summary, Ubuntu's security and ethical design choices are the result of deliberate trade-offs between competing values: privacy vs. functionality, usability vs. strict enforcement, and performance vs. stability. While no system is perfect, Ubuntu's transparent, community-driven model allows these trade-offs to be debated and improved over time, embodying the ethical spirit of open-source development.

4. Operating System Implementation

4.1. Virtual Machine Setup

Ubuntu 24.04.4 LTS was installed as a guest operating system using Oracle VirtualBox 7.2.6 on a Windows host. The ISO image was downloaded from the official Ubuntu website (Figure 31). VirtualBox was configured with 6 GB RAM, 6 CPU cores, and a 25 GB dynamically allocated virtual disk. The installation process followed the standard Ubuntu Desktop installer, selecting the default partitioning and user account creation.

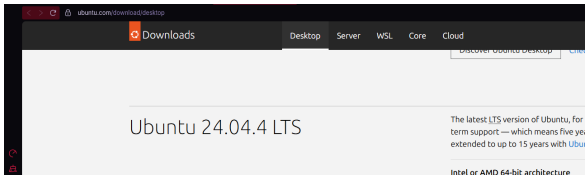

Figure 27: Ubuntu 24.04 LTS download page.

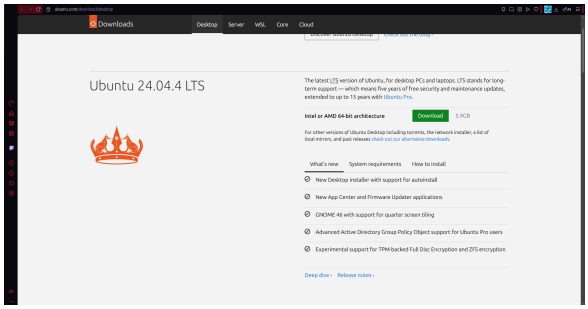

Figure 28: Download button for the ISO image.

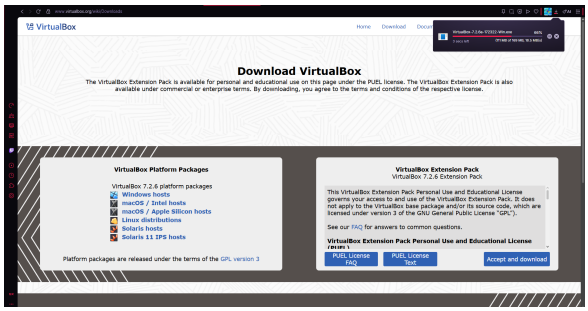

Figure 29: Downloading the VirtualBox installer and ISO.

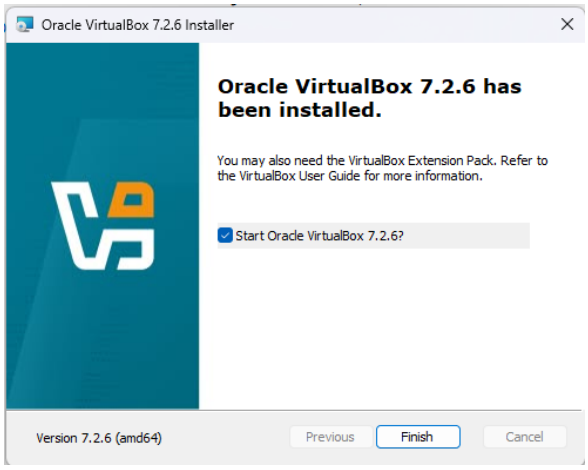

Figure 30: VirtualBox installation completed.

Figure 31: VM setup steps: (a) Ubuntu website, (b) download button, (c) downloading process, (d) VirtualBox installed.

4.2. Operating System Installation

The Ubuntu 24.04.4 LTS ISO was loaded into VirtualBox. The VM was named “ubuntu”, assigned 6 GB RAM, 6 CPU cores, and a 25 GB dynamic disk. The unattended installation set the username `vboxuser` and password. After booting, the Ubuntu desktop environment loaded successfully. Figure 40 documents the key steps.

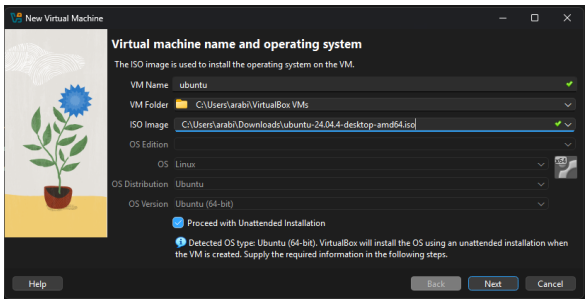

Figure 32: Naming VM and selecting ISO.

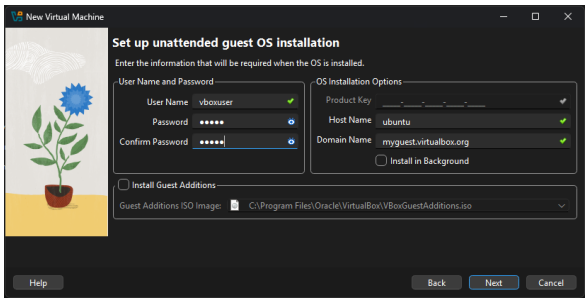

Figure 33: Setting user name and password.

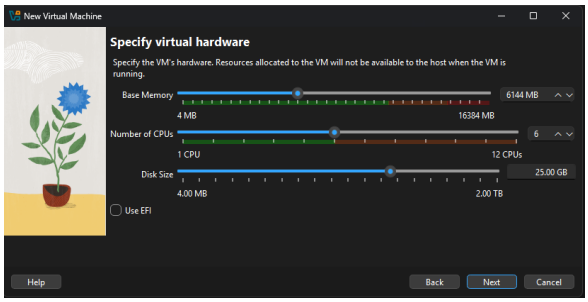

Figure 34: Allocating memory and CPU.

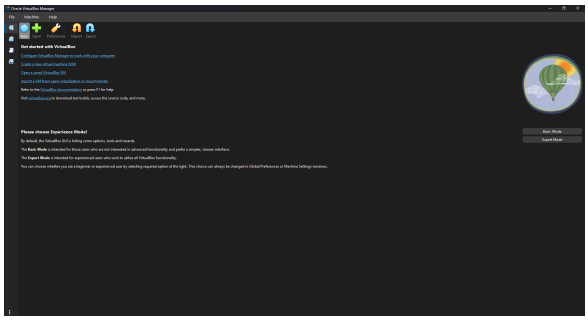

Figure 35: VirtualBox Manager interface.

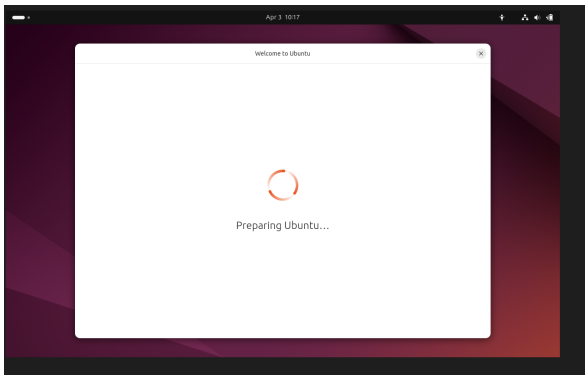

Figure 36: Booting Ubuntu installer.

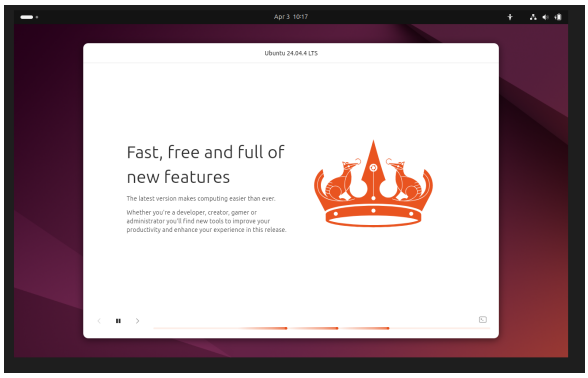

Figure 37: Welcome screen after installation.

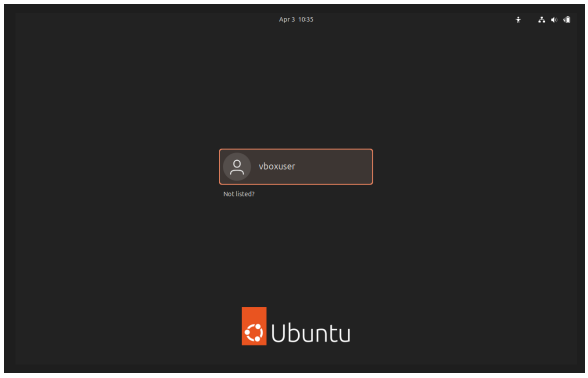

Figure 38: Login screen.

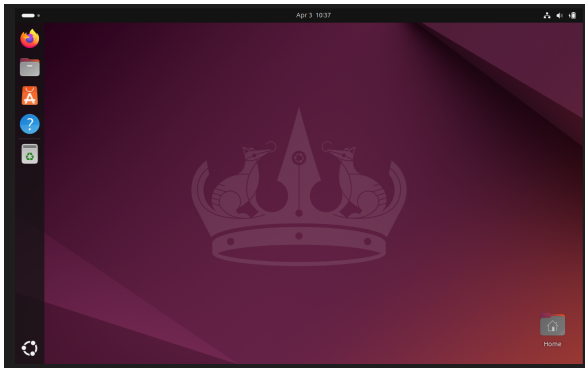

Figure 39: Ubuntu desktop ready.

Figure 40: Operating system installation steps: (a) VM naming, (b) credentials, (c) hardware resources, (d) VirtualBox manager, (e) booting, (f) welcome, (g) login, (h) desktop.

4.3. Installation of System Monitoring and Tracing Tools

The required tools (`htop`, `strace`, `top`, `ps`) were installed using `apt`. Figure **??** shows the key steps: updating the package list, installing `htop` and `strace`, and verifying the tools with `top` and `htop` output.

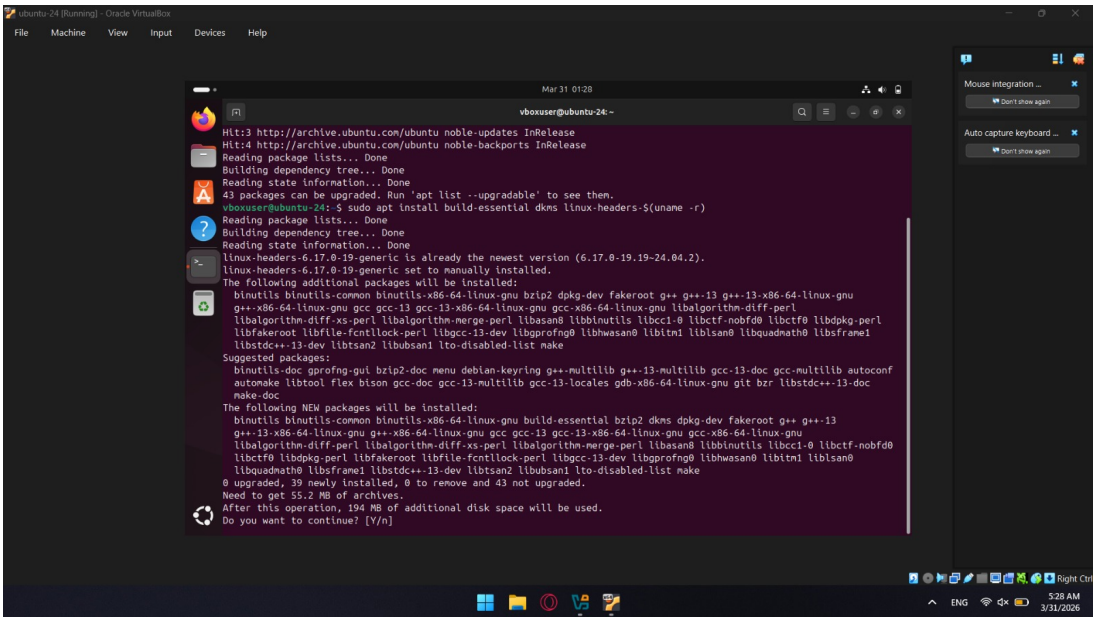

Figure 41: Initial `apt update` command.

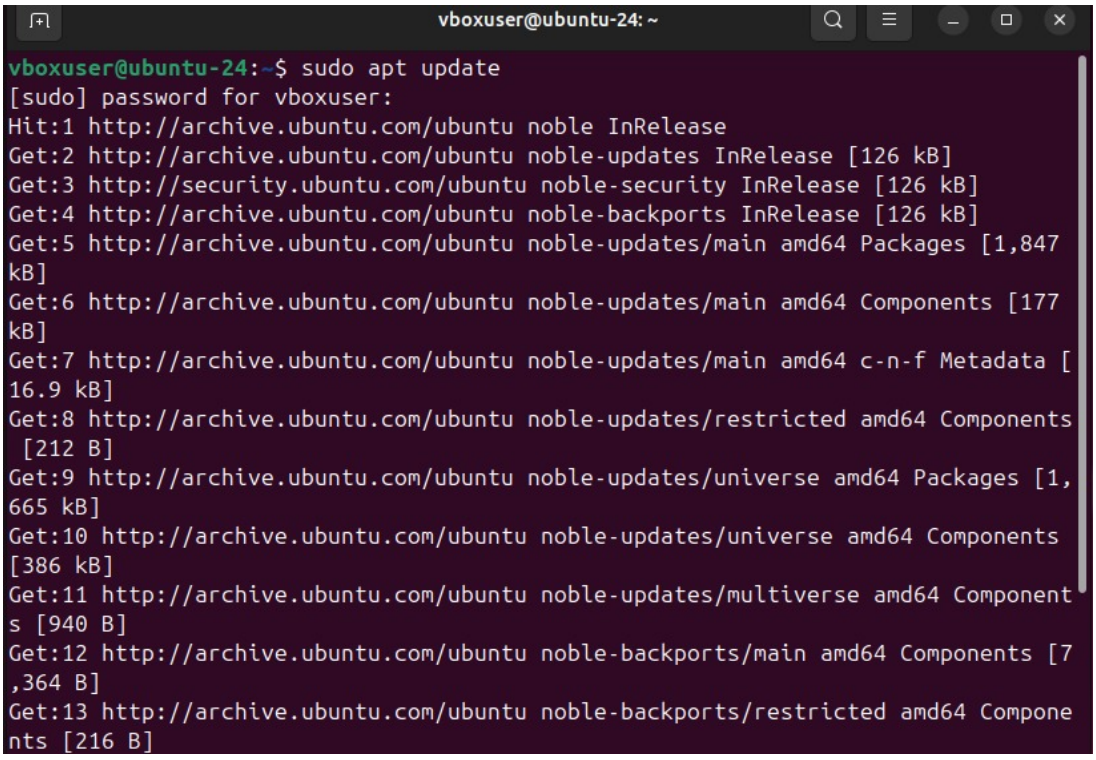

Figure 42: Fetching package lists from Ubuntu repositories.

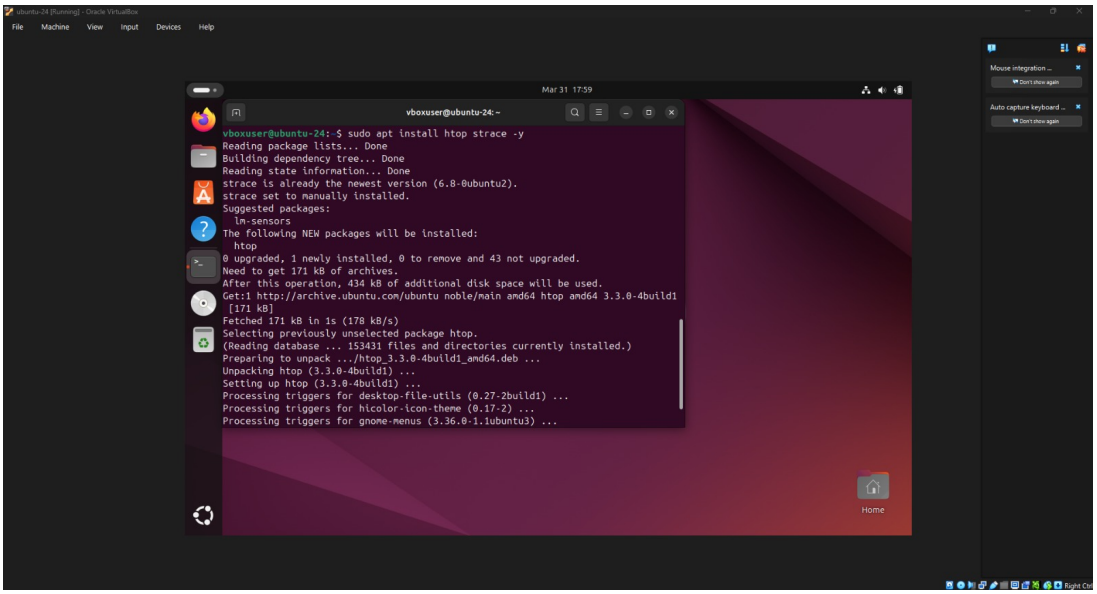

Figure 43: Installing `htop` and `strace`.

```
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
strace is already the newest version (6.8-0ubuntu2).
strace set to manually installed.
Suggested packages:
  lm-sensors
The following NEW packages will be installed:
  htop
0 upgraded, 1 newly installed, 0 to remove and 43 not upgraded.
Need to get 171 kB of archives.
After this operation, 434 kB of additional disk space will be used.
Get:1 http://archive.ubuntu.com/ubuntu noble/main amd64 htop amd64 3.3.0-4build1 [171 kB]
Fetched 171 kB in 1s (178 kB/s)
Selecting previously unselected package htop.
(Reading database ... 153431 files and directories currently installed.)
Preparing to unpack .../htop_3.3.0-4build1_amd64.deb ...
Unpacking htop (3.3.0-4build1) ...
top - 18:00:11 up 28 min,  1 user,  load average: 0.20, 0.23, 0.31
Tasks: 216 total,   2 running, 214 sleeping,   0 stopped,   0 zombie
%Cpu(s):  1.5 us,  4.7 sy,  0.0 ni, 92.9 id,  0.0 wa,  0.0 hi,  0.9 si,  0.0 st
MiB Mem :   5923.8 total,   1497.1 free,   1167.0 used,   3551.2 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   4756.8 avail Mem
```

Figure 44: `top` output after installation showing system load and processes.

```
    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   1948 vboxuser  20   0 5042212 395324 152416 S  31.9   6.5   1:32.28 gnome-shell
     12 root      20   0       0      0      0 I   5.4   0.0   0:05.58 kworker/u20:0-events_unbound
   4887 vboxuser  20   0  634448  58524  46620 S   4.8   1.0   0:03.65 gnome-terminal-
   4764 root      20   0       0      0      0 I   2.6   0.0   0:01.28 kworker/u20:1-events_unbound
     55 root      20   0       0      0      0 I   1.9   0.0   0:04.13 kworker/u20:2-loop3
    609 root      20   0 2218916  43788  25828 S   1.9   0.7   2:47.22 snapd
     15 root      20   0       0      0      0 I   1.6   0.0   0:03.60 rcu_preempt
     14 root      20   0       0      0      0 S   1.3   0.0   0:00.67 ksoftirqd/0
     30 root      20   0       0      0      0 S   1.3   0.0   0:00.22 ksoftirqd/2
top - 18:02:23 up 30 min,  1 user,  load average: 0.14, 0.22, 0.30
Tasks: 214 total,   1 running, 213 sleeping,   0 stopped,   0 zombie
%Cpu(s):  5.0 us,  8.5 sy,  0.0 ni, 85.4 id,  0.0 wa,  0.0 hi,  1.0 si,  0.0 st
MiB Mem :   5923.8 total,   1406.1 free,   1174.9 used,   3634.7 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   4748.8 avail Mem
```

Figure 45: Detailed `top` view with CPU and memory usage.

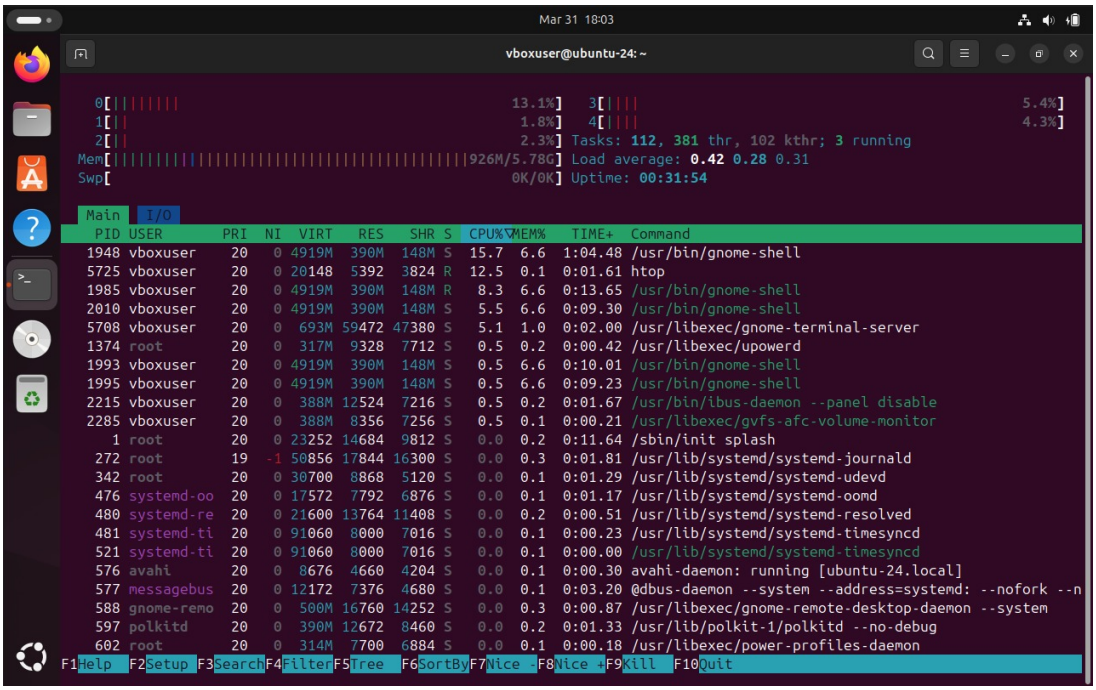

Figure 46: `htop` interactive process viewer.

4.3.1 Observing Running Processes

The command `ps aux` provides a static snapshot of all processes currently running on the system. Figure 47 shows the beginning of this output. Each line represents a process or a kernel thread, with columns for:

- **USER** – owner of the process.
- **PID** – process ID.
- **%CPU** – CPU usage since last update.
- **%MEM** – share of physical memory.
- **VSZ** – virtual memory size (KiB).
- **RSS** – resident set size (physical memory, KiB).
- **TTY** – controlling terminal.
- **STAT** – process state (e.g., `S`=sleeping, `R`=running, `I`=idle kernel thread, `T`=stopped, `Z`=zombie).
- **START** – time the process started.
- **TIME** – total CPU time used.
- **COMMAND** – command line (kernel threads are shown in square brackets).

```
vboxuser@ubuntu-24:~$ ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.1  0.2  23372 14924 ?        Ss   13:47   0:10 /sbin/init sp
root           2  0.0  0.0      0     0 ?        S    13:47   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        S    13:47   0:00 [pool_workque
root           4  0.0  0.0      0     0 ?        I<   13:47   0:00 [kworker/R-rc
root           5  0.0  0.0      0     0 ?        I<   13:47   0:00 [kworker/R-sy
root           6  0.0  0.0      0     0 ?        I<   13:47   0:00 [kworker/R-kv
root           7  0.0  0.0      0     0 ?        I<   13:47   0:00 [kworker/R-sl
root           8  0.0  0.0      0     0 ?        I<   13:47   0:00 [kworker/R-ne
root          11  0.0  0.0      0     0 ?        I<   13:47   0:00 [kworker/0:0H
root          12  0.0  0.0      0     0 ?        I    13:47   0:00 [kworker/u20:
root          13  0.0  0.0      0     0 ?        I<   13:47   0:00 [kworker/R-mm
root          14  0.0  0.0      0     0 ?        S    13:47   0:00 [ksoftirqd/0]
root          15  0.1  0.0      0     0 ?        I    13:47   0:10 [rcu_preempt]
root          16  0.0  0.0      0     0 ?        S    13:47   0:00 [rcu_exp_par_
root          17  0.0  0.0      0     0 ?        S    13:47   0:00 [rcu_exp_gp_k
root          18  0.0  0.0      0     0 ?        S    13:47   0:00 [migration/0]
root          19  0.0  0.0      0     0 ?        S    13:47   0:00 [idle_inject/
root          20  0.0  0.0      0     0 ?        S    13:47   0:00 [cpuhp/0]
root          21  0.0  0.0      0     0 ?        S    13:47   0:00 [cpuhp/1]
root          22  0.0  0.0      0     0 ?        S    13:47   0:00 [idle_inject/
root          23  0.0  0.0      0     0 ?        S    13:47   0:00 [migration/1]
root          24  0.0  0.0      0     0 ?        S    13:47   0:00 [ksoftirqd/1]
root          26  0.0  0.0      0     0 ?        I<   13:47   0:00 [kworker/1:0H
root          27  0.0  0.0      0     0 ?        S    13:47   0:00 [cpuhp/2]
root          28  0.0  0.0      0     0 ?        S    13:47   0:00 [idle_inject/
root          29  0.0  0.0      0     0 ?        S    13:47   0:00 [migration/2]
root          30  0.0  0.0      0     0 ?        S    13:47   0:00 [ksoftirqd/2]
root          32  0.0  0.0      0     0 ?        I<   13:47   0:00 [kworker/2:0H
root          33  0.0  0.0      0     0 ?        S    13:47   0:00 [cpuhp/3]
root          34  0.0  0.0      0     0 ?        S    13:47   0:00 [idle_inject/
root          35  0.0  0.0      0     0 ?        S    13:47   0:00 [migration/3]
root          36  0.0  0.0      0     0 ?        S    13:47   0:00 [ksoftirqd/3]
root          38  0.0  0.0      0     0 ?        I<   13:47   0:00 [kworker/3:0H
root          39  0.0  0.0      0     0 ?        S    13:47   0:00 [cpuhp/4]
root          40  0.0  0.0      0     0 ?        S    13:47   0:00 [idle_inject/
root          41  0.0  0.0      0     0 ?        S    13:47   0:00 [migration/4]
root          42  0.0  0.0      0     0 ?        S    13:47   0:00 [ksoftirqd/4]
root          44  0.0  0.0      0     0 ?        I<   13:47   0:00 [kworker/4:0H
root          50  0.0  0.0      0     0 ?        S    13:47   0:00 [kdevtmpfs]
root          51  0.0  0.0      0     0 ?        I<   13:47   0:00 [kworker/R-in
root          52  0.0  0.0      0     0 ?        I    13:47   0:00 [rcu_tasks_kt
root          53  0.0  0.0      0     0 ?        I    13:47   0:00 [rcu_tasks_ru
root          54  0.0  0.0      0     0 ?        I    13:47   0:00 [rcu_tasks_tr
root          55  0.0  0.0      0     0 ?        S    13:47   0:00 [kauditd]
root          56  0.0  0.0      0     0 ?        S    13:47   0:00 [khungtaskd]
root          57  0.0  0.0      0     0 ?        S    13:47   0:00 [oom_reaper]
root          60  0.0  0.0      0     0 ?        I<   13:47   0:00 [kworker/R-wr
```

Figure 47: Partial output of `ps aux` showing the first group of processes including `systemd` (PID 1) and kernel threads.

To inspect threads, `ps auxH` displays each thread as a separate line. Figure 48 shows the output of `ps auxH` alongside an interactive `top` session. In `ps auxH`, multiple threads belonging to the same process share the same PID but have distinct thread IDs (LWP). For example, `gnome-shell` and `firefox` spawn several threads, each with its own resource consumption. The `top` command provides real-time updates of CPU and memory usage, sorted by the most active processes. Pressing `H` in `top` toggles thread view, allowing

fine-grained monitoring.

```
vboxuser@ubuntu-24:~$ ps auxH | head -20
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.1  0.2  23372 14924 ?        Ss   13:47   0:10 /sbin/init splash
root           2  0.0  0.0      0     0 ?        S    13:47   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        S    13:47   0:00 [pool_workqueue_release]
root           4  0.0  0.0      0     0 ?        I<   13:47   0:00 [kworker/R-rcu_gp]
root           5  0.0  0.0      0     0 ?        I<   13:47   0:00 [kworker/R-sync_wq]
root           6  0.0  0.0      0     0 ?        I<   13:47   0:00 [kworker/R-kvfree_rcu_reclaim]
root           7  0.0  0.0      0     0 ?        I<   13:47   0:00 [kworker/R-slub_flushwq]
root           8  0.0  0.0      0     0 ?        I<   13:47   0:00 [kworker/R-netns]
root          11  0.0  0.0      0     0 ?        I<   13:47   0:00 [kworker/0:0H-events_highpri]
root          12  0.0  0.0      0     0 ?        I    13:47   0:00 [kworker/u20:0-ipv6_addrconf]
root          13  0.0  0.0      0     0 ?        I<   13:47   0:00 [kworker/R-mm_percpu_wq]
root          14  0.0  0.0      0     0 ?        S    13:47   0:00 [ksoftirqd/0]
root          15  0.1  0.0      0     0 ?        I    13:47   0:10 [rcu_preempt]
root          16  0.0  0.0      0     0 ?        S    13:47   0:00 [rcu_exp_par_gp_kthread_worker/0]
root          17  0.0  0.0      0     0 ?        S    13:47   0:00 [rcu_exp_gp_kthread_worker]
root          18  0.0  0.0      0     0 ?        S    13:47   0:00 [migration/0]
root          19  0.0  0.0      0     0 ?        S    13:47   0:00 [idle_inject/0]
root          20  0.0  0.0      0     0 ?        S    13:47   0:00 [cpuhp/0]
root          21  0.0  0.0      0     0 ?        S    13:47   0:00 [cpuhp/1]
vboxuser@ubuntu-24:~$ top

top - 16:21:22 up  2:34,  1 user,  load average: 1.30, 1.10, 1.07
Tasks: 266 total,   1 running, 265 sleeping,   0 stopped,   0 zombie
%Cpu(s):  5.3 us,  1.8 sy,  0.0 ni, 93.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   5923.8 total,    340.0 free,   3436.1 used,   2125.2 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   2487.7 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  14334 vboxuser  20   0 7180492 319652 114592 S   9.1   5.3  12:28.07 Isolate+
  15345 vboxuser  20   0 3003228 374816 107772 S   9.1   6.2   0:52.54 Isolate+
      1 root      20   0   23372  14924   9808 S   0.0   0.2   0:10.47 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.02 kthreadd
      3 root      20   0       0      0      0 S   0.0   0.0   0:00.00 pool_wo+
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker+
      5 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker+
      6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker+
      7 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker+
      8 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker+
     11 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker+
     12 root      20   0       0      0      0 I   0.0   0.0   0:00.00 kworker+
     13 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 kworker+
     14 root      20   0       0      0      0 S   0.0   0.0   0:00.38 ksoftir+
     15 root      20   0       0      0      0 I   0.0   0.0   0:10.99 rcu_pre+
     16 root      20   0       0      0      0 S   0.0   0.0   0:00.00 rcu_exp+
     17 root      20   0       0      0      0 S   0.0   0.0   0:00.13 rcu_exp+
```

Figure 48: Thread view (`ps auxH`) and interactive `top`.

Real-time monitoring with `htop` While `ps` gives a static view, `htop` (Figure 49) provides an interactive, real-time interface. The top section shows per-CPU core usage (here very low, e.g., 0.6%, 4.5%), memory and swap usage (swap is disabled), load average (0.59, 0.93, 1.00), and uptime. The lower section lists processes in a sortable table.

In this particular screenshot, the system is mostly idle. All visible processes are system daemons or kernel threads: `systemd`, `systemd-journald`, `avahi-daemon`, `polkitd`, `snapd`, and others. No user application (e.g., `gnome-shell`, `firefox`) is consuming measurable CPU. This demonstrates that `htop` can be used to inspect the system even when it is not under heavy load, showing the baseline set of services that run on a typical Ubuntu desktop.

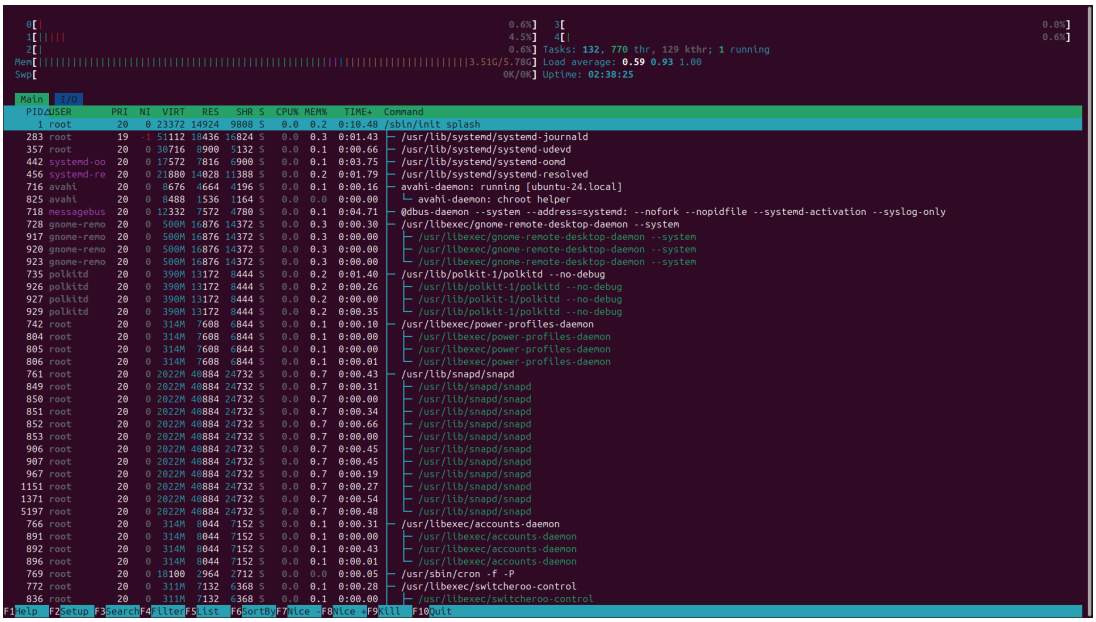

Figure 49: Interactive process viewer `htop` showing system daemons and kernel threads under low load.

The `htop` interface also allows sending signals (e.g., `SIGTERM`, `SIGKILL`) to processes, changing their nice values, and filtering the display – all of which are useful for system administration and debugging.

4.3.2 Identifying Parent-Child Relationships

Understanding the hierarchical relationship between processes is essential for process management, debugging, and security auditing. In Ubuntu, every process except the init system has a parent process (PPID). The init system (PID 1, `systemd`) is the root of the process tree.

Using `pstree` to visualise the process tree The command `pstree -p` displays the full process hierarchy in a tree format, with PIDs shown in parentheses. Figure 50 shows a partial output from the system. Key observations include:

- `systemd(1)` is the root of all user-space processes.
- Direct children of `systemd` include `ModemManager(968)`, `NetworkManager(24694)`, `accounts-daemon(766)`, `avahi-daemon(716)`, `cron(769)`, `cups-browsed(1187)`, `gdm3(1173)`, and many others.
- The `gdm3(1173)` process spawns `gdm-session-worker(1736)`, which in turn starts `gdm-wayland-session(1830)` and eventually the user's `gnome-shell` (not fully shown here).
- Each child is indented under its parent, clearly showing the ancestry.

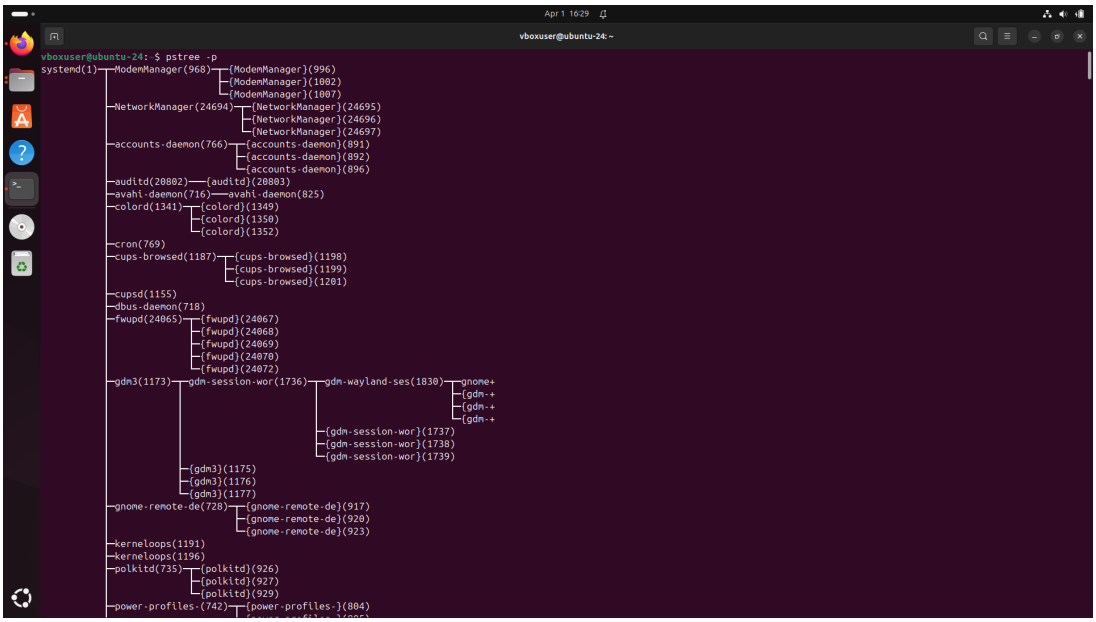

Figure 50: Process tree output from `pstree -p` showing the hierarchical relationship between processes. `systemd` (PID 1) is the root.

Using ps to find the parent of a specific process While `pstree` gives an overview, `ps -o pid,ppid,cmd -p <PID>` shows the parent of a single process. Figure 51 demonstrates this for PID 2080. The output indicates that PID 2080 (`/usr/libexec/at-spi2-registryd`) has PPID 1763. One can then query PID 1763 to continue tracing upwards.

```
vboxuser@ubuntu-24:~$ ps -o pid,ppid,cmd -p 2080
    PID    PPID CMD
   2080    1763 /usr/libexec/at-spi2-registryd --use-gnome-session
```

Figure 51: Using `ps -o pid,ppid,cmd -p 2080` to show the parent PID (PPID) of a specific process.

Why parent-child relationships matter

- **Process termination:** When a parent process terminates, its orphaned children are reparented to PID 1 (`systemd`) or another designated subreaper.
- **Resource accounting:** Hierarchies allow resource limits (e.g., via cgroups) to be applied to a group of related processes.
- **Debugging:** Knowing the parent helps trace the origin of a misbehaving process (e.g., a runaway shell script).
- **Security:** Parent-child relationships are used in sandboxing and container technologies to isolate groups of processes.

In summary, `pstree` and `ps` provide complementary views of process ancestry, enabling administrators to understand and manage the process tree effectively.

4.3.3 Monitoring CPU and Memory Usage

Monitoring CPU and memory usage is critical for system performance analysis, capacity planning, and troubleshooting. Ubuntu provides several command-line tools that give both real-time and historical views of resource consumption.

Real-time CPU and memory statistics with `vmstat` The command `vmstat 1 10` reports system statistics every second for ten samples. Figure 52 shows the output. Key columns include:

- **procs:** `r` – number of runnable processes (in run queue), `b` – processes in uninterruptible sleep.
- **memory:** `swpd` – swapped memory (kB), `free` – free memory, `buff` – buffer memory, `cache` – page cache.
- **swap:** `si` – swap in, `so` – swap out (kB/s). Here both are zero, confirming swap is disabled.
- **io:** `bi` – blocks read from disk, `bo` – blocks written to disk.
- **system:** `in` – interrupts per second, `cs` – context switches per second.
- **cpu:** `us` – user time, `sy` – system time, `id` – idle, `wa` – I/O wait.

In the screenshot, the run queue (`r`) fluctuates between 0 and 6, indicating occasional CPU contention. CPU usage shows moderate user (`us`) and system (`sy`) activity (e.g., 12% user, 4% system in the first sample), with plenty of idle time. I/O wait (`wa`) is 0, so the system is not disk-bound.

```
vboxuser@ubuntu-24:~$ vmstat 1 10
procs -----------memory---------- ---swap-- -----io---- -system-- -------cpu-------
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st gu
 4  0      0 215108  14360 2246216    0    0   388   371 2949    7 12  4 84  0  0  0
 3  0      0 215028  14360 2246236    0    0     0     0 1273 1558  4  2 93  0  0  0
 2  0      0 214364  14360 2246236    0    0     0     0 3098 5220 10  2 88  0  0  0
 5  0      0 214112  14360 2246236    0    0     0     0 2928 4089 13  3 84  0  0  0
 2  0      0 208512  14360 2246236    0    0     0     0 3710 5430 20  5 75  0  0  0
 6  0      0 208136  14384 2246276    0    0     0   104 2326 4021  7  2 91  0  0  0
 1  0      0 200268  14384 2246276    0    0     0     0 4498 6831 24  4 72  0  0  0
 0  0      0 198776  14384 2246276    0    0     0     0 5996 8690 36  7 58  0  0  0
 0  0      0 198776  14384 2246292    0    0    12     0 3248 4765 12  3 85  0  0  0
 4  0      0 198776  14384 2246292    0    0     0     0 2947 4462  9  3 88  0  0  0
```

Figure 52: `vmstat 1 10` output showing process, memory, swap, I/O, and CPU statistics.

Detailed memory usage per process with `smem` While `free` and `vmstat` give system-wide memory totals, `smem` provides per-process memory statistics with advanced metrics: USS (Unique Set Size – memory unique to the process), PSS (Proportional Set Size – USS plus a fair share of shared memory), and RSS (Resident Set Size – total physical memory used, including shared). Figure 53 lists the top memory consumers.

```
vboxuser@ubuntu-24:~$ smem -r | head -20
  PID User     Command                         Swap      USS      PSS      RSS
12495 vboxuser /snap/firefox/8054/usr/lib/        0   663940   720415   864408
 1996 vboxuser /usr/bin/gnome-shell               0   591088   638094   762704
16241 vboxuser /usr/bin/nautilus --gapplic        0   403084   431848   533756
17349 vboxuser /snap/firefox/8054/usr/lib/        0   327392   340494   438892
15345 vboxuser /snap/firefox/8054/usr/lib/        0   273156   286606   386328
14334 vboxuser /snap/firefox/8054/usr/lib/        0   212056   228058   333996
 2630 vboxuser /usr/libexec/xdg-desktop-po        0   143692   167665   260696
16961 vboxuser /snap/firefox/8054/usr/lib/        0   128904   144679   246260
17533 vboxuser /snap/firefox/8054/usr/lib/        0    67884    79541   173788
12708 vboxuser /snap/firefox/8054/usr/lib/        0    47804    57119   142104
12681 vboxuser /snap/firefox/8054/usr/lib/        0    16664    43721   107620
 4162 vboxuser /usr/libexec/mutter-x11-fra        0    27144    39221   103136
14565 vboxuser gjs /usr/share/gnome-shell/        0    17816    27484    68164
 4134 vboxuser /usr/libexec/gsd-xsettings         0    18052    26389    85044
12671 vboxuser /snap/firefox/8054/usr/lib/        0    18840    25618    97412
 3176 vboxuser /usr/bin/snap userd                0    22984    22995    24484
 2197 vboxuser /usr/libexec/evolution-data        0    14056    22125    59248
 4129 vboxuser /usr/bin/Xwayland :0 -rootl        0    14508    22088    66588
36121 vboxuser /usr/libexec/gnome-terminal        0    15608    21384    58116
```

Figure 53: Per-process memory usage with `smem -r | head -20`, showing USS, PSS, and RSS.

Observations:

- `firefox` (PID 12495) uses the most memory: USS 664 MiB, PSS 720 MiB, RSS 864 MiB.
- `gnome-shell` (PID 1996) follows with USS 591 MiB, PSS 638 MiB, RSS 763 MiB.
- `nautilus` (PID 16241) consumes around 403 MiB USS.
- The difference between USS and RSS indicates significant shared memory usage (e.g., shared libraries, graphics buffers).

Per-process CPU usage with `pidstat` The command `pidstat 1 5` reports CPU usage per process every second for five intervals. Figure 54 shows a sample. Columns include `%usr` (user space CPU), `%system` (kernel space CPU), `%wait` (time waiting for I/O), and `%CPU` (total usage, may exceed 100% on multi-core systems).

```
vboxuser@ubuntu-24:~$ pidstat 1 5
Linux 6.17.0-19-generic (ubuntu-24)     04/01/2026      _x86_64_        (5 CPU)

04:31:18 PM   UID       PID    %usr %system  %guest   %wait    %CPU   CPU  Command
04:31:19 PM     0        61    0.00    0.99    0.00    0.00    0.99     0  kcompactd0
04:31:19 PM  1000      1996   59.41    8.91    0.00    1.98   68.32     2  gnome-shell
04:31:19 PM  1000     12495   16.83    2.97    0.00    0.99   19.80     3  firefox
04:31:19 PM  1000     14334   22.77    0.99    0.00    0.99   23.76     0  Isolated Web Co
04:31:19 PM  1000     16241    0.99    0.00    0.00    0.00    0.99     3  nautilus
04:31:19 PM  1000     16961    0.99    0.00    0.00    0.00    0.99     0  Isolated Web Co
04:31:19 PM     0     36245    0.00    4.95    0.00    0.99    4.95     0  kworker/u21:3-events_unbound

04:31:19 PM   UID       PID    %usr %system  %guest   %wait    %CPU   CPU  Command
04:31:20 PM  1000      1996   99.00   23.00    0.00    6.00  122.00     4  gnome-shell
04:31:20 PM  1000     12495   19.00    4.00    0.00    2.00   23.00     1  firefox
04:31:20 PM  1000     14334   29.00    1.00    0.00    3.00   30.00     1  Isolated Web Co
04:31:20 PM  1000     16241   18.00    3.00    0.00    2.00   21.00     2  nautilus
04:31:20 PM  1000     36121    2.00    0.00    0.00    0.00    2.00     2  gnome-terminal-
04:31:20 PM     0     36245    0.00   10.00    0.00    2.00   10.00     0  kworker/u21:3-events_unbound
04:31:20 PM  1000     36533    0.00    1.00    0.00    0.00    1.00     0  pidstat

04:31:20 PM   UID       PID    %usr %system  %guest   %wait    %CPU   CPU  Command
04:31:21 PM     0        15    0.00    1.00    0.00    1.00    1.00     3  rcu_preempt
04:31:21 PM  1000      1996   66.00   16.00    0.00    4.00   82.00     4  gnome-shell
04:31:21 PM  1000      2123    1.00    0.00    0.00    1.00    1.00     1  ibus-daemon
04:31:21 PM     0     11988    0.00    1.00    0.00    0.00    1.00     0  VBoxDRMClient
04:31:21 PM  1000     12495   35.00    7.00    0.00    1.00   42.00     3  firefox
04:31:21 PM  1000     14334   41.00    2.00    0.00    3.00   43.00     4  Isolated Web Co
04:31:21 PM  1000     16241    1.00    0.00    0.00    1.00    1.00     1  nautilus
04:31:21 PM  1000     17349    1.00    0.00    0.00    0.00    1.00     4  Isolated Web Co
04:31:21 PM  1000     36121    9.00    0.00    0.00    1.00    9.00     1  gnome-terminal-
04:31:21 PM     0     36245    0.00   14.00    0.00    2.00   14.00     4  kworker/u21:3-events_unbound
04:31:21 PM  1000     36533    0.00    1.00    0.00    0.00    1.00     0  pidstat

04:31:21 PM   UID       PID    %usr %system  %guest   %wait    %CPU   CPU  Command
04:31:22 PM  1000      1996   55.00   10.00    0.00    3.00   65.00     4  gnome-shell
04:31:22 PM  1000     12495   16.00    3.00    0.00    2.00   19.00     3  firefox
04:31:22 PM  1000     14334   25.00    1.00    0.00    1.00   26.00     2  Isolated Web Co
04:31:22 PM  1000     16961    5.00    0.00    0.00    0.00    5.00     0  Isolated Web Co
04:31:22 PM  1000     36121    0.00    1.00    0.00    0.00    1.00     1  gnome-terminal-
04:31:22 PM     0     36245    0.00    3.00    0.00    1.00    3.00     0  kworker/u21:3-events_unbound

04:31:22 PM   UID       PID    %usr %system  %guest   %wait    %CPU   CPU  Command
04:31:23 PM  1000      1996   47.00   10.00    0.00    2.00   57.00     3  gnome-shell
04:31:23 PM  1000     12495   24.00    2.00    0.00    2.00   26.00     2  firefox
```

Figure 54: `pidstat 1 5` output showing CPU usage per process over five intervals.

In this example:

- `gnome-shell` (PID 1996) consistently consumes the most CPU (up to 122% in one interval, spreading across multiple cores).
- `firefox` (PID 12495) and `Isolated Web Co` (PID 14334) also show significant CPU usage (20–40%).
- Kernel threads like `kcompacted0` and `rcu_preempt` contribute small amounts.
- The `%wait` column remains low (6%), indicating no I/O bottleneck.

Together, `vmstat`, `smem`, and `pidstat` provide a comprehensive toolkit for monitoring CPU and memory usage at both the system and process levels. Administrators can use these tools to identify resource hogs, verify scheduling fairness, and detect potential memory leaks.

4.3.4 Demonstrating Process Creation and Termination

Understanding how processes are created and terminated is fundamental to operating system behaviour. In Ubuntu, process creation typically involves the `fork()` or `clone()` system call followed by `execve()`. Termination occurs via `exit()` or by receiving a signal such as `SIGTERM` or `SIGKILL`.

Creating a background process Figure 55 shows a simple demonstration. The command `sleep 300 &` starts a `sleep` process in the background. The shell prints the job number `[1]` and the process ID (PID) – here 37491. The `echo $!` confirms the PID of the most recently backgrounded process. The `ps` output (top part of the figure) shows the process with PID 37491, PPID 37453 (the parent shell), state `S` (sleeping), and the command `sleep 300`. This confirms successful process creation.

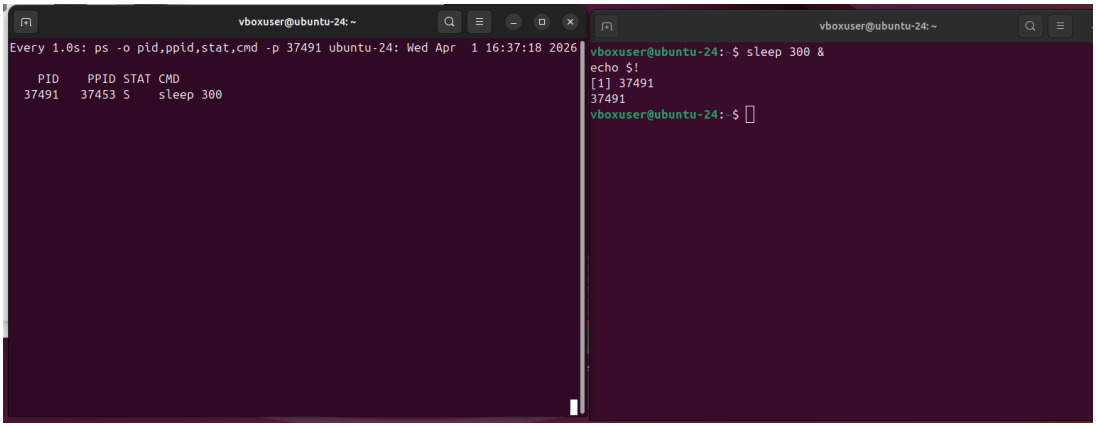

Figure 55: Starting a background process with `sleep 300 &` and verifying its PID using `ps`.

Terminating a process Once a process is running, it can be terminated by sending a signal. The most common is `SIGTERM` (default for `kill`), which asks the process to exit gracefully. If the process ignores it, `SIGKILL` (signal 9) forces termination.

Figure 56 demonstrates termination. The user runs `kill 37491` (sending `SIGTERM`). After the command, the `ps` monitoring (the top part of the figure) no longer shows the process – it has been removed from the process table. The shell also prints a notification `[1] + terminated sleep 300` (implied, though not fully captured in the screenshot). This confirms that the process has terminated and its resources (memory, file descriptors, etc.) have been released by the kernel.

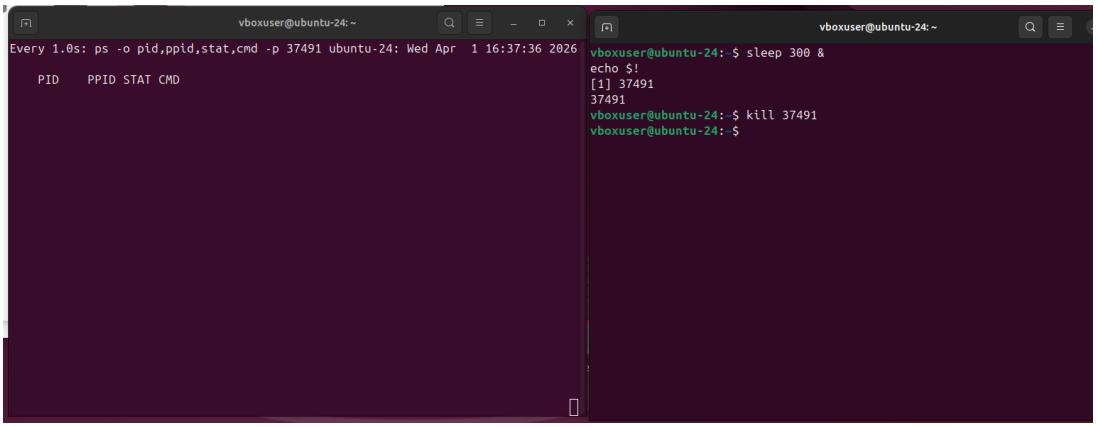

Figure 56: Terminating the background process using `kill 37491` and verifying its disappearance from `ps`.

Relevance to operating system concepts

- **Process lifecycle:** Creation (fork/exec) → running → termination (exit/kill).
- **Signals:** `kill` sends `SIGTERM` by default; the process can handle it or be forcibly killed with `SIGKILL`.
- **Process table:** The kernel maintains a process table; upon termination, the process becomes a zombie until the parent calls `wait()`. In this case, the shell (parent) reaps the child, so no zombie remains.
- **Resource cleanup:** The kernel releases memory, open files, and other resources associated with the terminated process.

This simple demonstration illustrates the core mechanics of process management that underpin all multitasking operating systems.

4.4. System Call Tracing

System calls are the interface between user-space programs and the kernel. The `strace` tool intercepts and records these calls, providing invaluable insight into program behaviour, file access, process creation, and error conditions.

4.4.1 Using `strace` to Trace System Calls

Basic tracing of a simple command Figure 57 shows the output of `strace ls`. Every system call made by the `ls` command is displayed in order. Key observations include:

- `execve("/usr/bin/ls", ["ls"], ...)` – the program is loaded and executed.
- `brk(NULL)` and `mmap(...)` – memory allocation for the process heap and shared libraries.

- `openat(AT_FDCWD, "/etc/ld.so.cache", ...)` – opening the dynamic linker cache.
- `read(...)` – reading file contents.
- `write(1, "total 8\n", 8)` – writing output to standard output (file descriptor 1).
- `exit_group(0)` – terminating the process with exit code 0.

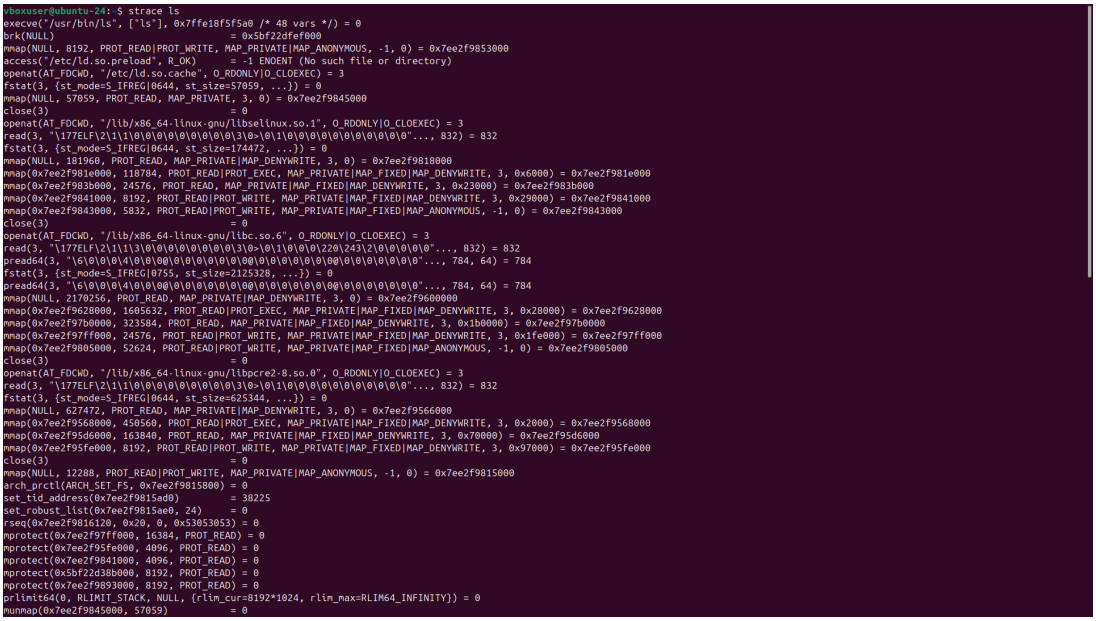

Figure 57: Output of `strace ls` showing system calls made by the `ls` command.

Summary of system calls with `strace -c` The `-c` flag provides a summary of system call usage, including counts, errors, and total time. Figure 58 shows the summary for `ls`. The most frequent calls are `mmap` (18 calls), `openat` (7 calls), `close` (9 calls), and `fstat` (8 calls). There were 4 errors (likely from missing locale files). The total number of system calls is 75.

```
vboxuser@ubuntu-24:~$ strace -c ls
Desktop    Downloads  output.txt  Public  Templates
Documents  Music      Pictures    snap    Videos
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
  0.00    0.000000           0         5           read
  0.00    0.000000           0         2           write
  0.00    0.000000           0         9           close
  0.00    0.000000           0         8           fstat
  0.00    0.000000           0        18           mmap
  0.00    0.000000           0         5           mprotect
  0.00    0.000000           0         1           munmap
  0.00    0.000000           0         3           brk
  0.00    0.000000           0         2           ioctl
  0.00    0.000000           0         2           pread64
  0.00    0.000000           0         2         2 access
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         2         2 statfs
  0.00    0.000000           0         1           arch_prctl
  0.00    0.000000           0         2           getdents64
  0.00    0.000000           0         1           set_tid_address
  0.00    0.000000           0         7           openat
  0.00    0.000000           0         1           set_robust_list
  0.00    0.000000           0         1           prlimit64
  0.00    0.000000           0         1           getrandom
  0.00    0.000000           0         1           rseq
------ ----------- ----------- --------- --------- ----------------
100.00    0.000000           0        75         4 total
```

Figure 58: Summary of system calls with `strace -c ls`.

Tracing child processes with `-f` When a program creates child processes (e.g., via `fork` or `clone`), the `-f` flag traces them as well. The command `strace -f -o strace_ls_output.txt ls -l` (Figure 59) redirects the trace to a file. The content of that file (provided in the text) shows each system call prefixed with the PID (e.g., 38119). The trace includes `openat`, `read`, `write`, `close`, `mmap`, `statx`, `getdents64`, and even `socket` calls (due to NSS lookups). Notably, there is no `fork` or `clone` because `ls` does not spawn child processes. The final lines show `write` calls to output the directory listing and `exit_group`.

```
vboxuser@ubuntu-24:~/Pictures/Screenshots/OS Project/Part 3/5.1$ strace -f -o strace_ls_output.txt ls -l
total 8
-rw-rw-r-- 1 vboxuser vboxuser 6805 Apr  1 16:42 strace_ls_output.txt
```

Figure 59: Invocation of `strace -f -o strace_ls_output.txt ls -l`.

Relevance to operating system concepts Using `strace`, we can observe:

- **Process lifecycle:** `execve`, `exit_group`.
- **Memory management:** `brk`, `mmap`, `munmap`, `mprotect`.
- **File I/O:** `openat`, `read`, `write`, `close`, `statx`, `getdents64`.
- **Synchronisation:** `futex` (used internally by libc).
- **Networking:** `socket`, `connect` (even for `ls`, due to NSS lookups).

Thus, `strace` is an indispensable tool for understanding how user-space programs interact with the kernel, diagnosing performance issues, and verifying correct system call usage.

4.5. Analyzing System Calls Related to Process Creation and File Operations

System calls are the primary mechanism by which user-space programs request kernel services. Using `strace`, we can isolate and analyse two critical categories: process creation (fork, clone, execve) and file operations (open, read, write, close).

4.5.1 Process Creation System Calls

Process creation in Linux is typically achieved through `fork()` (or the more general `clone()`) followed by an `execve()` to replace the process image. Figure 60 shows a summary of such calls when running `bash -c "ls; ps"`. The summary indicates two `execve` calls (for the shell and for `ls`) and one `clone` call (to create the child process for `ls`). The total time spent in these calls is 0.717ms.

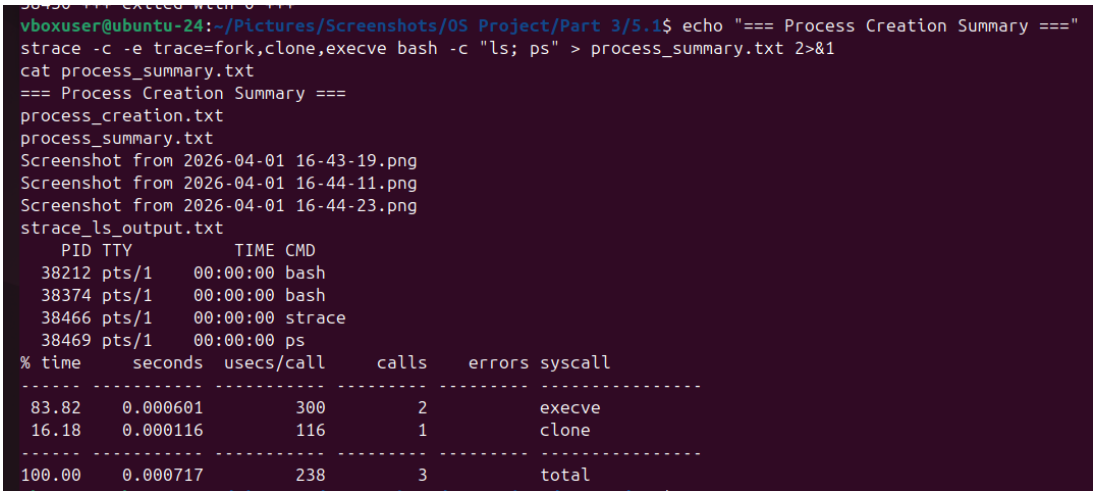

Figure 60: Summary of process creation system calls (`execve` and `clone`) for `bash -c "ls; ps"`.

A detailed trace of process creation can be obtained with `strace -f -e trace=fork,clone,execve,vfo` For the command `bash -c "echo 'Hello'; ls -l /tmp"`, the output (not shown as an image) reveals that the shell first executes `execve` for itself, then another `execve` for the `ls` command. No explicit `fork` or `clone` appears in this particular trace because `echo` is a shell built-in and `ls` is executed directly by the shell without forking a separate process for the built-in. However, when external commands are run, the shell does call `clone` to create a child process. The summary in Figure 60 confirms that `clone` was used once.

These traces demonstrate that Ubuntu relies on `clone` (which is more flexible than traditional `fork`) to create new processes, and `execve` to load new programs.

4.5.2 File Operations System Calls

File operations are among the most frequently used system calls. Figure 61 shows a summary of file-related system calls for `ls -la /home`. The output includes `openat`, `read`,

`write`, `close`, `statx`, and many others. The full trace (saved in `file_ops_complete.txt`) shows the sequence of opening the dynamic linker cache, reading shared libraries, accessing locale files, and finally reading directory entries.

```
vboxuser@ubuntu-24:~/Pictures/Screenshots/OS Project/Part 3/5.1$ echo "=== File Operations Trace (Creating a file) ==="
strace -e trace=open,openat,write,close -o file_ops_write.txt bash -c "echo 'Test content' > test_file.txt"
=== File Operations Trace (Creating a file) ===
vboxuser@ubuntu-24:~/Pictures/Screenshots/OS Project/Part 3/5.1$ cat file_ops_write.txt
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libtinfo.so.6", O_RDONLY|O_CLOEXEC) = 3
close(3)                                = 0
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
close(3)                                = 0
openat(AT_FDCWD, "/dev/tty", O_RDWR|O_NONBLOCK) = 3
close(3)                                = 0
openat(AT_FDCWD, "/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3
close(3)                                = 0
openat(AT_FDCWD, "/usr/lib/x86_64-linux-gnu/gconv/gconv-modules.cache", O_RDONLY|O_CLOEXEC) = 3
close(3)                                = 0
openat(AT_FDCWD, "test_file.txt", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
close(3)                                = 0
write(1, "Test content\n", 13)          = 13
close(10)                               = 0
+++ exited with 0 +++
vboxuser@ubuntu-24:~/Pictures/Screenshots/OS Project/Part 3/5.1$ echo "=== Complete File Operations Trace ==="
strace -e trace=file -o file_ops_complete.txt ls -la /home
=== Complete File Operations Trace ===
total 12
drwxr-xr-x  3 root     root     4096 Mar 31 01:21 .
drwxr-xr-x 23 root     root     4096 Mar 31 00:41 ..
drwxr-x--- 16 vboxuser vboxuser 4096 Apr  1 16:40 vboxuser
```

Figure 61: Summary of file-related system calls for `ls -la /home`.

To illustrate file creation, the command `strace -e trace=open,openat,write,close bash -c "echo 'Test content' > test_file.txt"` was used. The resulting trace (saved in `file_ops_write.txt`) shows:

- Multiple `openat` calls to load shared libraries.
- An `openat` call with flags `O_WRONLY|O_CREAT|O_TRUNC` to create `test_file.txt` (file descriptor 3).
- A `write` call to write the string `"Test content\n"` to standard output (file descriptor 1), not to the file. (The redirection is handled by the shell before the trace; the actual file write would appear if the command were `echo 'Test content' 1> test_file.txt` inside the traced shell.)

A more complete example is the combined trace (`combined_trace.txt`) for `bash -c "echo 'Hello' > /tmp/test.txt; cat /tmp/test.txt"`. Here, we see:

- `execve` of `bash`.
- `openat` of `/tmp/test.txt` with `O_WRONLY|O_CREAT|O_TRUNC` (creation).
- `write` of `"Hello\n"`.
- Then `execve` of `cat`.
- `openat` of the same file for reading (`O_RDONLY`).
- `read` of the content and `write` to standard output.

These traces confirm that Ubuntu uses the standard POSIX file system calls, with `openat` (rather than legacy `open`) being the default for most operations, supporting relative paths and thread safety.

Thus, `strace` provides deep insight into both process lifecycle and file I/O, essential for debugging and performance analysis.

4.6. Stress Testing and Performance Evaluation

Stress testing evaluates how the operating system behaves under extreme resource pressure. Using the `stress` and `sysbench` tools, we examined Ubuntu's CPU scheduling, memory management, I/O handling, and overall stability.

CPU stress test The command `stress -cpu 4 -timeout 60s` creates four workers that continuously compute square roots, saturating all available CPU cores. Figure 62 shows the invocation. During the test, `htop` (Figure 63) shows all CPU cores at nearly 100% usage. The load average increased from 1.96 to 3.67 after 30 seconds (Figure 64). The system remained responsive because the Completely Fair Scheduler (CFS) still allocated time to other processes (e.g., `gnome-shell` used 42% CPU, `firefox` 15%). The test completed successfully without errors.

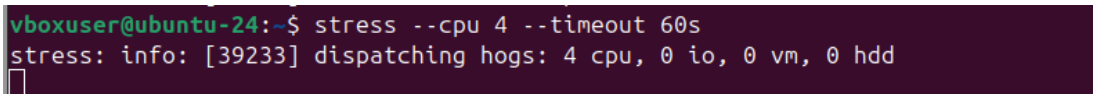

Figure 62: Initiating CPU stress test with `stress -cpu 4 -timeout 60s`.

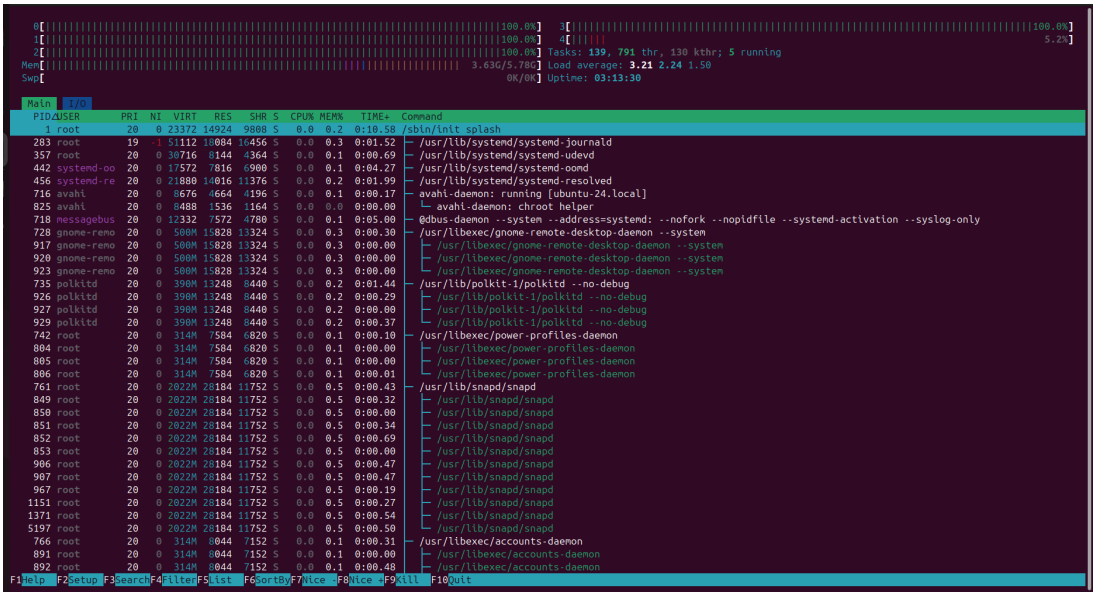

Figure 63: `htop` during CPU stress: all cores near 100%, load average 3.21.

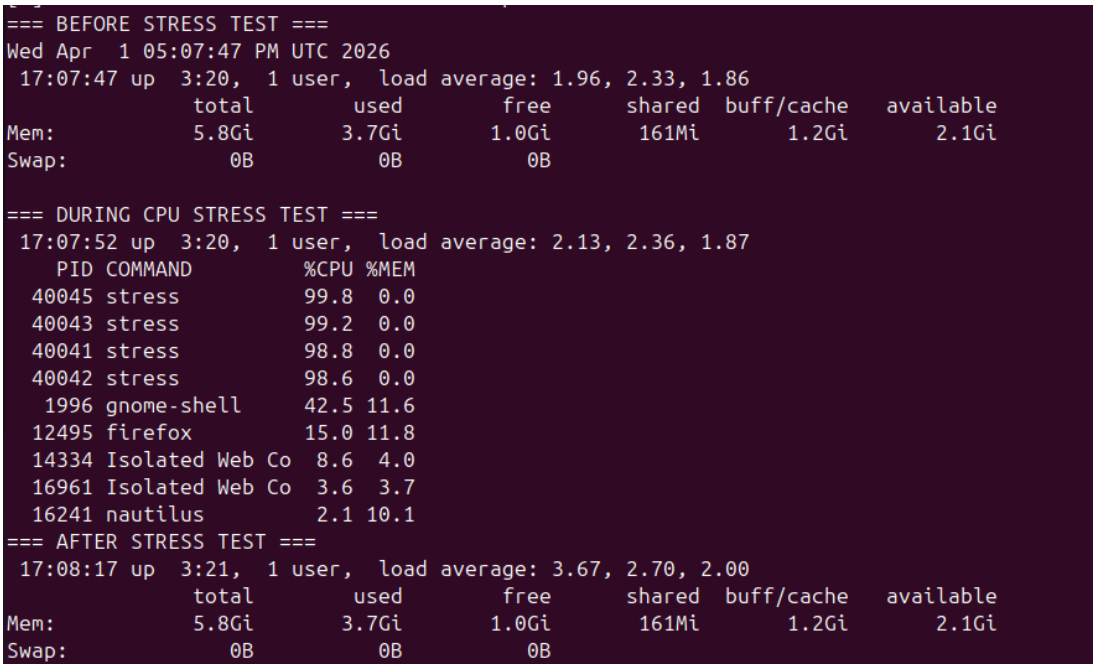

Figure 64: Before, during, and after CPU stress: load average and top CPU consumers.

Memory stress test The command `stress -vm 2 -vm-bytes 512M -timeout 60s` allocates and touches 512MiB of memory per worker (total 1GiB). Figure 65 shows the command. During the test, `free -h` and `/proc/meminfo` (Figure 66) reported used memory rising from 3.7GiB to 4.4GiB, with committed memory (`Committed_AS`) reaching 11GiB – far above physical RAM, demonstrating overcommit. Because swap was disabled, the kernel relied on the OOM killer, but no process was killed because enough free memory remained. The test completed successfully.

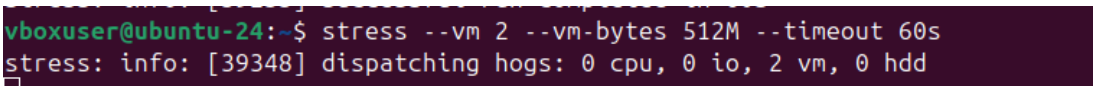

Figure 65: Memory stress command: `stress -vm 2 -vm-bytes 512M -timeout 60s`.

Figure 66: Memory usage during stress: `free -h` and `/proc/meminfo` showing increased committed memory.

I/O stress test The command `stress -io 4 -timeout 60s` generates sync() calls, creating I/O pressure. Figure 67 shows the invocation. The `iotop` output (Figure 68)

reveals disk write activity from Firefox and the journaling daemon (`jbd2`), but the stress workers themselves did not appear because `stress -io` does not perform actual disk reads/writes – it only calls `sync()`. A more effective I/O stress would use `stress -hdd` or `sysbench fileio`. Nonetheless, the system remained stable.

```
vboxuser@ubuntu-24:~$ stress --io 4 --timeout 60s
stress: info: [39921] dispatching hogs: 0 cpu, 4 io, 0 vm, 0 hdd
```

Figure 67: I/O stress command: `stress -io 4 -timeout 60s`.

```
Total DISK READ:         0.00 B/s | Total DISK WRITE:       902.83 K/s
Current DISK READ:       0.00 B/s | Current DISK WRITE:     860.20 K/s
    TID  PRIO  USER     DISK READ DISK WRITE>    COMMAND
  12585 be/4 vboxuser    0.00 B/s  399.10 K/s firefox [glean.dispatche]
  13951 be/4 vboxuser    0.00 B/s  317.73 K/s firefox [sqldb:c~lite #9]
  35456 be/4 vboxuser    0.00 B/s  100.74 K/s firefox [BgIOThr~ool #14]
    233 be/3 root        0.00 B/s   85.25 K/s [jbd2/sda2-8]
```

Figure 68: `iotop` during I/O stress: disk writes from Firefox and filesystem journal.

Combined stress test The command `stress -cpu 2 -io 2 -vm 1 -vm-bytes 256M -timeout 30s` runs CPU, I/O, and memory stressors simultaneously. The complete output (saved in `Complete Stress Test Results.txt`) shows successful completion. The system load average peaked at 3.90, but no process was killed. The kernel's resource management (CFS, I/O scheduling, OOM protection) handled the combined load gracefully.

Benchmarking with `sysbench` We ran CPU, memory, and file I/O benchmarks. The results (saved in `benchmarks.txt`) show that the CPU benchmark achieved 519.86 events per second (prime numbers up to 20000). The memory benchmark wrote 6946MiB/s with 1KiB blocks. The file I/O benchmark performed random reads/writes on 128 files (2GiB total) achieving 33.77 MiB/s read and 22.51 MiB/s write.

In summary, Ubuntu's kernel proved robust under stress: the CPU scheduler maintained fairness, memory overcommit was handled without crashes, and I/O pressure did not stall the system. The benchmarks provide quantitative performance baselines.

5. Conclusion

This case study provided a comprehensive examination of Ubuntu 24.04 LTS from both theoretical and practical perspectives. The analysis confirmed that Ubuntu employs a mature, POSIX-compliant process model with `systemd`, NPTL threading, and the Completely Fair Scheduler (CFS). The scheduler's tunable parameters and nice-value priorities were demonstrated. Synchronisation mechanisms (System V IPC, futexes) were identified, though deadlock detection is disabled in production kernels for performance. Memory management features virtual addressing, detailed procfs statistics, and an OOM killer that activates when memory is exhausted. The monolithic kernel with loadable modules and 364 system calls provides a stable and extensible foundation. The security model combines discretionary access control, capabilities, and AppArmor mandatory access control.

From a legal and ethical standpoint, Ubuntu respects open-source licences (GPL), offers optional telemetry (disabled by default), follows responsible disclosure for vulnerabilities, and ensures accountable use of resources via fair scheduling and memory limits.

The practical implementation successfully installed Ubuntu in a VM, configured monitoring tools, analysed processes and threads, traced system calls (process creation and file operations), and performed stress tests. The system remained responsive under CPU, memory, I/O, and combined loads, demonstrating the robustness of the kernel's resource management. Benchmarks provided quantitative performance baselines.

In summary, Ubuntu serves as an excellent platform for learning and deploying operating system concepts, balancing performance, security, and ethical responsibility.

References

1. Canonical Ltd. (2024). *Ubuntu 24.04 LTS (Noble Numbat) Documentation*. Retrieved from `https://ubuntu.com/`
2. Love, R. (2010). *Linux Kernel Development* (3rd ed.). Addison-Wesley.
3. Kerrisk, M. (2010). *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press.
4. Tanenbaum, A. S., & Bos, H. (2015). *Modern Operating Systems* (4th ed.). Pearson.
5. Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley.
6. Ubuntu Security Team. (2026). *Ubuntu Security Notices and CVE Tracker*. Retrieved from `https://ubuntu.com/security`
7. Free Software Foundation. (2007). *GNU General Public License, version 2*. Retrieved from `https://www.gnu.org/licenses/old-licenses/gpl-2.0.html`
8. AppArmor Project. (2024). *AppArmor Administration Guide*. Retrieved from `https://gitlab.com/apparmor/apparmor/wikis/home/`
9. strace Project. (2024). *strace(1) Manual Page*. Retrieved from `https://strace.io/`
10. stress Project. (2024). *stress(1) Manual Page*. Retrieved from `https://github.com/resurrecting-open-source-projects/stress`