IEN-004 - Stage 3 — Linux Kernel Initialization

At the end of the previous stage, GRUB has successfully loaded two critical files into memory:

- the Linux kernel (`vmlinuz`)
- the initial RAM disk (`initrd` or `initramfs`)

It has also prepared the kernel command line, which may include parameters such as:

```
root=/dev/mapper/vg_root-lv_root ro rhgb quiet
```

or

```
root=UUID=0d6d0b68-8d95-42d6-bf7c-8a88f6b82d5e
```

or, as we encountered during our RHEL 5.2 recovery project:

```
acpi=no
```

At this point, everything the kernel needs has been placed into memory.

GRUB's work is complete.

The CPU now performs one of the most significant transitions during the entire startup sequence:

> **Execution is transferred from the bootloader to the Linux kernel.**

From this moment onward, Linux—not the firmware or GRUB—is responsible for bringing the system to life.

The Kernel Is Not Yet an Operating System

Many people think Linux "starts" the moment GRUB loads the kernel.

Technically, that is not true.

The kernel is only a binary image stored in memory.

It has not yet:

- initialized memory management,
- detected storage devices,
- mounted a filesystem,
- started services,
- or launched user-space processes.

Think of the kernel as the blueprint of a city before any roads, electricity, or buildings have been constructed.

The blueprint exists—but the city is not operational.

The first responsibility of the kernel is therefore to build its own operating environment.

From Compressed Image to Running Kernel

The kernel image stored in `/boot` is typically compressed.

On most systems, the file looks something like:

```
vmlinuz-5.14.0-503.el9.x86_64
```

Notice the name:

```
vmlinuz
```

The "z" indicates that the kernel image is compressed.

Why?

Because compressed kernels:

- require less disk space,
- reduce bootloader loading time,
- and minimize storage requirements.

Immediately after execution begins, the kernel performs its own decompression.

A simplified sequence looks like this:

```
GRUB
  |
  ▼
Load compressed kernel
  |
  ▼
Jump to kernel entry point
  |
  ▼
Kernel decompression
  |
  ▼
Kernel relocates itself
  |
  ▼
Kernel begins initialization
```

Only after decompression does the actual kernel begin executing.

Setting Up the CPU

The very first task of the kernel is preparing the processor.

Although the firmware has already initialized the CPU enough to execute code, Linux requires far more control.

The kernel configures features such as:

- processor operating mode,
- interrupt handling,
- exception handlers,
- floating-point support,
- cache management,
- CPU feature detection,
- multiple processor initialization (SMP).

Modern systems may contain dozens or even hundreds of CPU cores.

The kernel must detect every available processor and prepare them for scheduling.

Typical kernel messages include:

```
SMP: Total of 32 processors activated.
```

or

```
Detected 16 logical CPUs.
```

At this stage, only a very small portion of the kernel is active.

Memory Initialization

Once the processor is ready, the kernel turns its attention to memory.

RAM is the foundation upon which every process, filesystem cache, driver, and application depends.

The kernel must determine:

- how much memory is installed,
- which regions are usable,
- which regions are reserved,
- where the kernel itself resides,
- where future allocations can occur.

A simplified memory map might look like this:

```
+------------------------------+
| Kernel Image                 |
+------------------------------+
| Reserved Firmware Memory     |
+------------------------------+
| DMA Region                   |
+------------------------------+
| Available RAM                |
+------------------------------+
| High Memory                  |
+------------------------------+
```

From this point onward, Linux begins managing memory instead of relying on firmware.

This transition is fundamental because all future allocations—processes, drivers, caches, page tables—depend on the kernel's memory manager.

Engineering Insight

One of Linux's greatest strengths is that the firmware is only needed during the earliest stages of boot.

Once initialization is complete, the kernel takes full ownership of system resources.

From this point forward, Linux manages memory, processors, interrupts, and devices independently of the firmware.

Initializing Core Kernel Subsystems

With the CPU and memory prepared, the kernel begins activating its internal subsystems.

These subsystems provide the fundamental services required by every part of the operating system.

Examples include:

- Scheduler
- Virtual Memory Manager
- Interrupt Manager
- Timer Subsystem
- Process Manager
- Virtual Filesystem (VFS)
- Security Frameworks
- IPC Mechanisms

You can think of these as the "operating system inside the operating system."

Without them, Linux cannot execute processes or communicate with hardware.

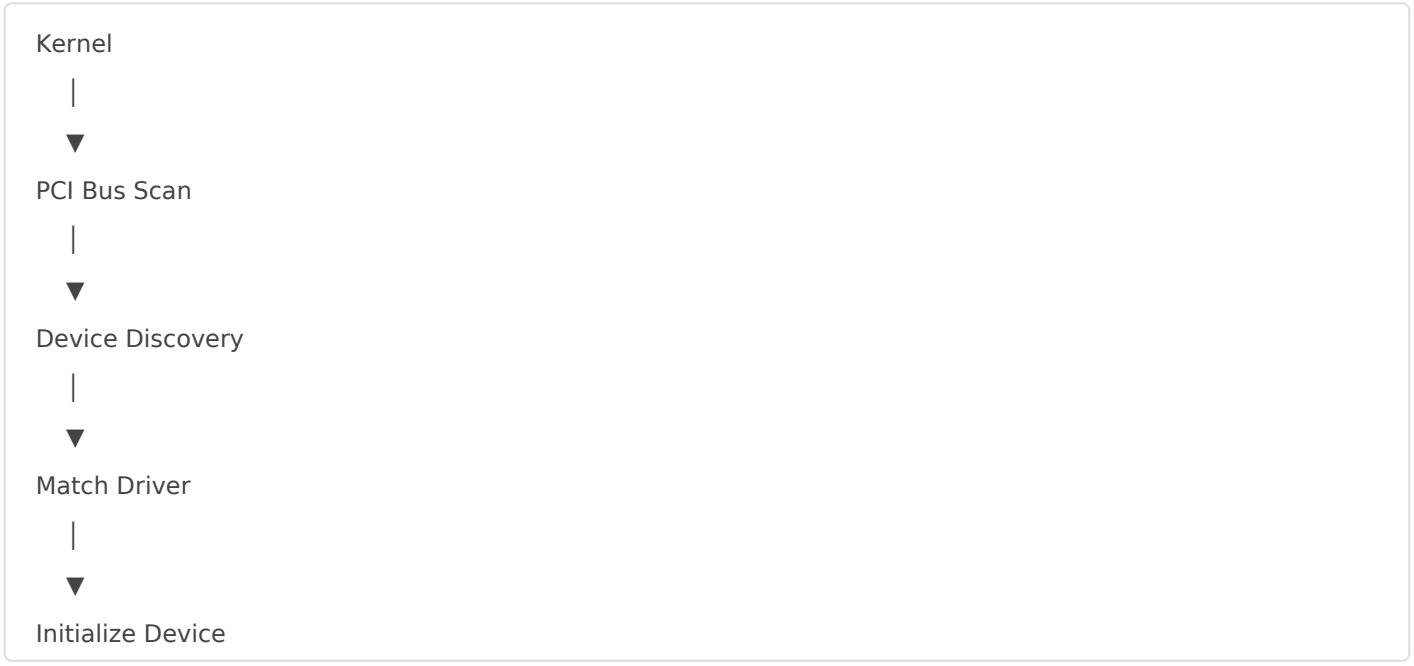
Device Driver Initialization

Once the kernel infrastructure is operational, Linux begins initializing device drivers.

Drivers are responsible for communicating with hardware such as:

- SATA controllers,
- NVMe controllers,
- RAID controllers,
- Fibre Channel HBAs,
- USB controllers,
- Ethernet adapters,
- GPUs,
- storage controllers.

A simplified sequence is shown below:



This process is essential.

If the kernel lacks the driver required for the storage controller containing the root filesystem, the boot process cannot continue.

The Kernel Still Cannot Access the Root Filesystem

This surprises many engineers.

At this point the kernel has:

- initialized memory,
- initialized processors,
- activated internal subsystems,
- detected hardware,
- loaded some drivers,

yet it **still cannot mount the root filesystem**.

Why?

Because the storage stack may not yet be complete.

Enterprise Linux systems commonly use technologies such as:

- LVM
- Software RAID
- Multipath
- Fibre Channel SAN
- iSCSI
- NVMe over Fabrics

The kernel alone often does not have enough information or modules to activate these storage layers.

It therefore requires assistance from another component:

> **The Initial RAM Disk (initrd/initramfs).**

This is precisely why the bootloader loaded the initrd alongside the kernel.

The kernel's next step is to unpack that temporary root filesystem and execute its `/init` program.

Common Kernel Initialization Failures

Many boot failures occur during kernel initialization.

Some of the most common include:

Symptom	Likely Cause
`Kernel panic - not syncing`	Fatal kernel error
`Unable to mount root fs`	Root filesystem cannot be located
`No working init found`	`/init` or `/sbin/init` missing
`Unknown-block(0,0)`	Storage device unavailable

Symptom	Likely Cause
System freezes after "Loading kernel..."	Early kernel initialization failure

A crucial point to remember is that a **kernel panic** does **not** always mean the kernel itself is defective.

In many cases, the kernel is functioning correctly but cannot proceed because it lacks access to the storage or userspace required for the next stage.

Understanding exactly **where** the panic occurs is therefore far more valuable than the panic message alone.

Engineering Insight

One of the most misunderstood boot errors is:

```
Kernel panic - not syncing:
VFS: Unable to mount root fs
```

Although the message contains the words *Kernel panic*, the underlying problem is frequently **not the kernel binary itself**. More commonly, the kernel cannot locate or access the root filesystem because the required storage driver, LVM activation, RAID assembly, or filesystem module is unavailable. Treating every kernel panic as a "kernel problem" often leads engineers in the wrong direction.

Summary

By the end of this stage, the Linux kernel has transformed from a compressed image on disk into a running operating system core. It has initialized the CPU, taken control of memory management, activated its internal subsystems, and begun communicating with hardware through device drivers.

However, the system is still incomplete. The kernel knows **how** to manage hardware, but it may not yet know **where** the real root filesystem resides.

To bridge that gap, the kernel hands control to the **initial RAM disk (`initrd` / `initramfs`)**, which will load any remaining drivers, activate storage technologies such as LVM or RAID, and prepare the real root filesystem for mounting.

Revision #1
Created 4 July 2026 04:15:32 by Sulistio Priatmojo
Updated 4 July 2026 04:16:10 by Sulistio Priatmojo