VirtIO and the Novix Logging System

In this chapter, we connect our OS to **virtual hardware** and build a **modern logging system** — similar to what Linux uses!
This allows Novix OS to communicate with devices through **VirtIO** and log kernel events in a structured, reliable way.

Overview

VirtIO is a standardized interface that allows virtual machines to communicate with virtual devices (like disks, consoles, and network cards) provided by **QEMU** or other hypervisors.

In this chapter, you'll learn:

- What VirtIO is and how it integrates with our kernel
- How to detect and initialize VirtIO devices
- How to use the VirtIO Console for **emergency logging**
- How Novix introduces a **multi-level logging system** (INFO, WARN, ERR, DBG)
- How user programs can log safely, even from user-space!

VirtIO in a Nutshell

VirtIO uses **Memory-Mapped I/O (MMIO)**. This means the CPU communicates with a device by reading and writing specific memory addresses — which are actually mapped to device registers.

Each VirtIO device (like console, disk, or network) has a unique MMIO base address and a set of registers that describe its type, vendor, and features.

For example:

```
#define VIRTIO_MMIO_BASE 0x10001000
#define VIRTIO_MAGIC     0x74726976   // "virt"
#define VIRTIO_DEV_ID_CONSOLE 3
```

When the kernel scans these addresses, it can identify connected devices and initialize them dynamically.

VirtIO Device Discovery

In `kernel/virtio.c`, we implement:

```
void virtio_bus_init_scan();
```

What It Does

- Scans through all possible VirtIO MMIO slots.
- Checks each one for the "virt" magic number.
- Identifies the **device type** (block, network, console, etc).
- Initializes the **VirtIO Console** when found.

Example log output:

```
Novix RISC-V 64 OS, (c) NovixManiac, Version 0.0.1

Booting ...
```

```
[trap_init] Trap vector initialized at : 0x80201f2c
[stack_init] STACK initialized, size = 128 KB, start = 0x804aaf90, end = 0x804caf90
[ram_init]   RAM   initialized, size =  64 MB, start = 0x804cb000, end = 0x844cb000 total pages = 16384
[heap_init]  HEAP  initialized, size =  32 MB, start = 0x844cb000, end = 0x864cb000
[paging_init] kernel page table initialized at : 0x804cb000
[virtio-scan] Scanning for virtio devices...
[virtio-scan] Device found at 0x10001000: ID=1 Vendor=0x554d4551
  → Network device
[virtio-scan] Device found at 0x10002000: ID=2 Vendor=0x554d4551
  → Block device
[virtio-scan] Device found at 0x10007000: ID=3 Vendor=0x554d4551
  → Console device
[virtio_emerge_init] Initialized at 0x10007000
[virtio-scan] Device found at 0x10008000: ID=16 Vendor=0x554d4551
  → GPU device
[virtio-debug] Scan completed.
```

Once detected, the **VirtIO Console** becomes our **primary logging interface**.

The Emergency Logging System

Logging starts even before the full kernel is operational.
For that, we use the **VirtIO Emergency Console**, implemented in `virtio-emerg.c`.

Function: `virtio_emerg_init()`

```c
void virtio_emerg_init(volatile uint32_t *regs);
```

This sets up the base address of the "emergency write" register inside the console device.

When initialized, we can directly send characters and strings to the QEMU console output.

Function: `emerg_putc()`

```c
void emerg_putc(char c);
```

Sends a single character to the emergency console.

Function: `emerg_puts()`

```c
void emerg_puts(const char *s);
```

Sends a full string, one character at a time.

Function: `emerg_printf()`

```c
void emerg_printf(const char *fmt, ...);
```

Formats text like `printf()`, then sends it to the console — allowing color-coded or formatted logs even before the UART is ready!

High-Level Logging Macros

Once VirtIO is initialized, we can use simple macros for kernel logging:

```
#define LOG_INFO(...)  virtio_emerg_log("INFO", NULL, __VA_ARGS__)
#define LOG_WARN(...)  virtio_emerg_log("WARN", NULL, __VA_ARGS__)
#define LOG_ERR(...)   virtio_emerg_log("ERR",  NULL, __VA_ARGS__)
#define LOG_DEBUG(...) virtio_emerg_log("DBG",  NULL, __VA_ARGS__)
```

These print messages like:

```
[INFO]   RAM initialized, 64 MB detected
[WARN]   Low memory: 3 pages remaining
[ERR]    Page fault at VA=0x0000000000000000
[DBG]    Scheduler tick = 142
```

The system automatically prefixes messages with a log level for clarity.

VirtIO Log Formatting Functions

`virtio_emerg_log()`

Formats a line with the given log level and message.

`virtio_emerg_log_va()`

Variant that accepts a `va_list` — used internally for variable argument forwarding.

`virtio_emerg_log_ext()`

Adds even more metadata to the log, such as source module, process ID, and label.

Example output:

```
[USER:PID=2] [INFO] shell: initialized environment
```

User-Space Logging Support

User programs can also write logs using special macros that route through `log_user()`.

In `include/user.h` we define:

```
#define LOG_USER_INFO(...) log_user(LOG_LVL_INFO, __VA_ARGS__)
#define LOG_USER_WARN(...) log_user(LOG_LVL_WARN, __VA_ARGS__)
#define LOG_USER_ERR(...)  log_user(LOG_LVL_ERR, __VA_ARGS__)
#define LOG_USER_DBG(...)  log_user(LOG_LVL_DBG, __VA_ARGS__)
```

Function: `log_user()`

This is the bridge between user-space and kernel logging.

```
void log_user(user_log_level_t level, const char *fmt, ...);
```

It performs these steps safely:

1. Temporarily switches back to the **kernel page table**.
2. Logs the message via VirtIO (avoiding page faults).
3. Restores the user's page table afterward.

This ensures **secure, crash-free logging** from user-space!

Copying Data Between Kernel and User Space

The function `copy_to_user()` safely copies memory between spaces:

```
int copy_to_user(pagetable_t pt, void *dst_user, void *src_kernel, size_t len);
```

For every byte copied, it translates the virtual address to a physical address with `walkaddr()` — ensuring user memory is valid.
Debug logs show each copy operation, which helps in development.

VirtIO Console Test

Inside `virtio_emerg_test()`, we can verify that writing to the emergency console works:

```
void virtio_emerg_test(volatile uint32_t *regs);
```

It writes directly to the device's `emerg_wr` register:

```
Hi
[virtio_console_emerge_test] emergency-write done
```

This confirms that the console is operational before we use it for kernel logging.

Integration in `kernel_main()`

Now, all components come together beautifully:

```
void kernel_main(void) {
    bss_init();
    uart_cls();
    uart_printf("Novix RISC-V 64 OS, (c) NovixManiac, Version 0.0.1\n\n");
    uart_puts("Booting ...\n");

    trap_init();
    stack_init();
    ram_init();
    heap_init();
    paging_init();
    date_time_test();

    // Initialize VirtIO bus and console
    virtio_bus_init_scan();

    // Test logging
    log_test();

    // Load idle process
    size_t fs;
    void *idle_elf_file = tarfs_lookup("idle.elf", &fs, 1);
    if (!idle_elf_file) {
        PANIC("[kernel_main] idle.elf not found!");
    }

    for (;;);
}
```

Example Output

```
[virtio-scan] Scanning for virtio devices...
[virtio-scan] Device found at 0x10007000: ID=3 Vendor=0x554d4551
```

```
  → Console device
[virtio_emerge_init] Initialized at 0x10007000
[INFO] Heap initialized, 32 MB available
[DBG] Kernel boot complete
```

Summary

By the end of this chapter, Novix OS now has:

- A **VirtIO driver** for QEMU virtual devices
- A **reliable emergency console** for early logging
- A **multi-level logging API** for kernel and user-space
- Safe copying between user and kernel memory
- Unified log output format — just like modern Unix kernels

Logging is no longer just `printf()` — it's structured, extensible, and hardware-backed!

`*A smile on my face - and hopefully on the students' too.*`