Userland Logging in Novix OS

Logging is one of the most essential tools for debugging and monitoring any operating system.
Until now, **Novix OS** supported logging only inside the **kernel** — using functions like `LOG_INFO`, `LOG_ERR`, and so on.

In this chapter, we bring **logging capabilities to userland**!
Programs running in user space (like the shell or echo) can now send messages to the system log through a syscall.

Learning Goals

By the end of this chapter, you will understand:

- How to create a **user-level logging API** that connects to kernel logging
- How the **SYS_LOG** syscall works
- How the kernel safely handles and tags messages from user processes
- How user programs can log using the `LOG()` macro

Overview: From User to Kernel Log

When a user program calls `LOG("started...");`, this happens:

1. `LOG()` expands to a call to `ulog()` in **userland**
2. `ulog()` formats the message and performs a **syscall (SYS_LOG)**
3. The kernel's `sys_log()` function receives the message safely
4. The kernel prefixes the log with `[USER:PID=x]` and sends it to the virtio-emerg console

This mechanism allows both kernel and user processes to share the same centralized logging output.

1. Userland Logging Interface

We start by defining a **simple logging interface** that user programs can use.

user/include/log.h

```c
#pragma once

#include <stdarg.h>
#include <stddef.h>
#include <stdint.h>

void ulog(const char *fmt, ...);

// Simple macro for convenience
#define LOG(...) ulog(__VA_ARGS__)
```

Explanation:

- The `ulog()` function behaves similarly to `printf()` — it supports format strings and variadic arguments.
- The macro `LOG()` acts as a shorthand so user code can simply write:

```c
LOG("Shell started...");
```

2. User Logging Implementation

Now, let's implement `ulog()` to send log messages via a system call.

user/lib/log.c

```c
#include "../include/log.h"
#include "../include/syscall.h"
#include "../include/string.h"
#include "../include/common.h"
#include "../include/stdio.h"

void ulog(const char *fmt, ...) {
    char buf[256];
    va_list args;
    va_start(args, fmt);
    kvnprintf(buf, sizeof(buf), fmt, args);
    va_end(args);

    // Send to kernel via SYS_LOG
    syscall(SYS_LOG, (uint64_t)buf, 0, 0);
}
```

Key points:

- `kvnprintf()` formats the message into a temporary buffer.
- The syscall `SYS_LOG` sends the string to the kernel for processing.
- No direct UART output is done from user space (for safety and consistency).

3. Kernel Support for User Logs

In the kernel, we extend the syscall handler to include **SYS_LOG**.

kernel/syscall.c

```c
case SYS_LOG: {
    const char *msg = (const char *)f->regs.a0;  // syscall param
    long ret = sys_log(msg);                     // handle it
    f->regs.a0 = ret;                            // return code
    break;
}
```

Now, let's see the actual implementation of `sys_log()`.

sys_log() Implementation

```c
long sys_log(const char *msg) {
    if (!msg) return -1;

    enable_sum();

    char buf[256];
    strncpy(buf, msg, sizeof(buf) - 1);
    buf[sizeof(buf) - 1] = '\0';

    disable_sum();

    struct process *p = current_proc;
```

```
    const char *label = p->name;

    // Temporarily switch to kernel pagetable
    uintptr_t old_satp = switch_pagetable((uintptr_t)kernel_pagetable);

    virtio_emerg_log_ext("USER", p->pid, "INFO", label, "%s", buf);

    // Restore user page table
    restore_pagetable(old_satp);

    return 0;
}
```

Explanation:

- `enable_sum()` allows safe access to user memory.
- The message is copied into a **kernel buffer** to prevent user memory faults.
- The kernel logs the message with `virtio_emerg_log_ext()`, tagging it with:
  - Source type (`USER`)
  - Process ID (`PID`)
  - Log level (`INFO`)
  - Process name
- The pagetable switch ensures kernel log safety.

4. Logging Levels for Userland

In **include/debug.h**, we already defined kernel-level logging macros.
Now, we include variants for user-level logging that automatically switch page tables.

include/debug.h

```
#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__)
```

These call `log_user()`, which is implemented in the kernel and takes care of context switching.

kernel/user.c

```
void log_user(user_log_level_t level, const char *fmt, ...) {
    uintptr_t old_satp = switch_pagetable((uintptr_t)kernel_pagetable);

    va_list args;
    va_start(args, fmt);

    switch (level) {
        case LOG_LVL_INFO: virtio_emerg_log_va("INFO", NULL, fmt, args); break;
        case LOG_LVL_WARN: virtio_emerg_log_va("WARN", NULL, fmt, args); break;
        case LOG_LVL_ERR:  virtio_emerg_log_va("ERR", NULL, fmt, args); break;
        case LOG_LVL_DBG:  virtio_emerg_log_va("DBG", NULL, fmt, args); break;
    }

    va_end(args);
```

```
    restore_pagetable(old_satp);
}
```

This allows the kernel to display user log messages at different severity levels while maintaining isolation.

5. Using LOG() in User Programs

Let's modify our **shell** to log when it starts.

user/shell.c

```
int main() {
    char line[MAXLINE];
    char filename[MAXLINE + 8];
    char *title = "Novix RISC-V 64 OS, (c) NovixManiac, Shell version : 0.0.1\n";

    LOG("started...");

    cls();
    puts(title);

    while (1) {
        puts("$ ");
        ...
    }

    return 0;
}
```

This simple line will produce a formatted kernel log message that looks like:

`[USER:PID=2] [INFO] shell: started...`

If multiple user programs also log messages, each one is tagged with its PID and program name, allowing clear tracking of user-level events.

Design Reflection

Component	Purpose
`ulog()`	Formats and sends messages from userland
`SYS_LOG`	System call that bridges user → kernel logging
`sys_log()`	Safely handles message transfer and logs it
`LOG()`	User-friendly macro wrapper
Kernel pagetable switch	Prevents faults during kernel log writes

Security & Safety Notes

- User memory access is protected by **SUM enable/disable**.
- Kernel copies messages into an internal buffer before processing.
- Pagetable switching ensures that no invalid user pointers are dereferenced.
- Logs from userland are clearly separated from kernel logs for traceability.

Summary

In this chapter, we've achieved:

- Introduced the `SYS_LOG` syscall for userland logging
- Created `ulog()` and `LOG()` macros for user-friendly logging
- Implemented kernel-side log processing with safe copying and labeling
- Integrated logging into the shell
- Achieved unified kernel + user logging streams

With **userland logging**, Novix OS now supports complete system diagnostics — both the kernel and user programs can contribute to the same log stream, making debugging and tracing system behavior much easier.

A smile on my face — because Novix OS supports now userland logging!