Lecture 24: Device Driver

(Abstraction, Design, CUDA)

Xin Liu

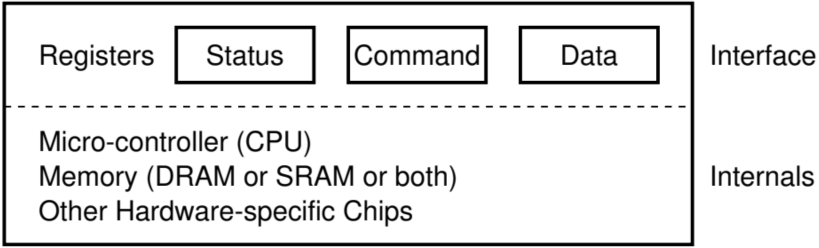
Florida State University
xliu15@fsu.edu

COP 4610 Operating Systems
https://xinliulab.github.io/FSU-COP4610-Operating-Systems/

Review of I/O Devices

- From the OS' perspective, an I/O device is essentially

  A Set of Registers and Protocols
- Regardless of what is connected behind them
  - Examples: serial port, keyboard, printer



A hardware-level abstraction for devices

- **Connecting Devices to the Computer:**
  - **Bus:** The sole external interface of the computer, connecting all devices.
    - The bus itself can be viewed as a device.
- **Managing the Bus:**
  - **Interrupt Controller:**
    - Coordinates device interrupts to manage CPU attention.
    - Avoids busy-waiting
  - **DMA:** Acts as a co-processor, assisting in data transfer for I/O devices.
- **Expanding Co-Processing:**
  - **GPU:** Another specialized co-processor.
  - Enables **heterogeneous computing** alongside the CPU.

Driver: A software-level abstraction for devices

- Each type of device has its own protocol and set of registers.
- Even for the same type of device, different models may have different registers and protocols.
- Directly exposing devices to the OS greatly increases the likelihood of errors.

Solution: Device Abstraction

To address this, we abstract all devices to communicate with the CPU in a unified manner as much as possible.

Outline

Today's Key Question:

How does the OS enable applications to access these devices?

- **What is a device driver?**
- **How do we abstract the device?**

Main Topics for Today:

- **Principles of Device Drivers**
- **Linux Device Driver Design**
- **GPU and CUDA**

Principles of Device Drivers

Main Functions of I/O Devices

Input and Output

- “Readable (read) and writable (write) byte sequences (streams or arrays)”
- Most common devices fit this model:
  - Character Device:
    - Terminal/Serial Port - Byte Stream
    - Printer - Byte Stream (e.g., PostScript files)
  - Block Device:
    - Hard Disk - Byte Array (Block Access)
  - GPU: Neither a Character Device Nor a Block Device
    - Byte Stream (Control) + Byte Array (Display)

How to Abstract I/O Devices

Since I/O devices are designed for input and output, we can abstract them using basic operations:

- Devices as Objects (Files) Supporting Various Operations
  - System call **read** - Reads data from a specified location on the device
  - System call **write** - Writes data to a specified location on the device
  - System call **ioctl** - Reads/Sets the status of the device
    - RTFM: `man 2 ioctl`

What is a Device Driver?

Translates system calls (read/write/ioctl/...) into interactions with device registers

- Essentially a piece of kernel code
- Human $\rightarrow$ Shell $\rightarrow$ System Calls $\rightarrow$ Driver $\rightarrow$ Devices
  - Our computer continuously performs abstraction and translation
- May block (e.g., waiting on a semaphore P operation, awakened by V operation in interrupt)

Examples of Objects in /dev/

```
$ cd /dev && ls
```

- **Note:** A driver is simply code:
  - It may not have a real device behind it.
  - It can simulate devices entirely through code.
- Examples:
  - `/dev/random`, `/dev/urandom` - Random number generators
    - Try: `head -c 512 /dev/urandom | hexdump -C`
    - May use hardware device for true random numbers, or software for pseudo-random numbers
  - `/dev/null` - "Null" device
    - Try: `yes > /dev/null` (Fake write success)
    - Try: `cat /dev/null` (No data to read)
    - Observe using `strace`
  - `/dev/zero` - "Zero" device
    - Try: `cat /dev/zero | head -c 512 | hexdump -C`

The Challenges of Device Drivers

I/O devices appear as a ”black box”

- Prone to Errors!
- Any code mistake simply causes it to ”not work”
- Device drivers: often the **lowest** quality code in Linux kernel

The Complexity of Device Drivers

- The computer industry includes countless devices, each with unique registers and protocols.
- Often, only the device’s programmers understand them, and even they may not fully.
- Since Vista, Microsoft, and now Linux, have been moving drivers to user space to prevent system crashes from driver errors.

Limitations of Byte Stream Abstraction

Devices involve not only data but also control

- Especially for additional functions and configurations of devices
- All extra features rely on `ioctl`
  - Arguments, returns, and semantics of `ioctl()` vary according to the device driver
  - Complex "hidden specifications"

Examples

- Printer settings: print quality, paper feed, duplex, card stock, cleaning, auto binding, ...
  - A printer worth tens of thousands is not that simple !
- Keyboard lighting effects, repeat rate, macro programming, ...
- Disk health, cache control, ...

Device Complexity: More Than Just Byte Streams

Functions beyond the "Byte Stream"

- Devices are far more complex than simple input/output streams.
- Many features require specific configurations, often handled through `ioctl` calls.
- This complexity is just the tip of the iceberg.

The Terminal: A Special Device in UNIX

- The Shell acts as the interface between the user and the OS.
- The **terminal** is the I/O device behind this interface.
- Terminals support text-based input and output, enabling users to interact with the OS.
- They handle complex features like:
  - Input modes, signal handling, and line editing
  - Terminal settings, which can be configured dynamically
    - **RTFM:** `tty`, `stty`, ...

The Terminal Usage

```
$ tty
$ echo hello > /dev/pts/1
```

- Find out which system call “recognizes” the terminal.

```
#include <stdio.h>

int main() {
    printf("Hello, World\n");
}
```

- Tip: Compare using the following two comandes:

```
$ strace ./a.out
```

```
fstat(1, st_mode=S_IFCHR|0620, st_rdev=makedev(0x88,
0x1), ...) = 0
```

```
$ strace ./a.out > /dev/null
```

```
ioctl(1, TCGETS, 0x7fffc7a27ef0) = -1 ENOTTY
(Inappropriate ioctl for device)
```

More On Terminal

Terminal as a Line Buffer

- Terminal uses **line buffering**: output is stored until a newline ('\n') or 'fflush(stdout);' is called.
- Files, on the other hand, use **full buffering**, meaning data is stored until the buffer is full or manually flushed.
- Buffering improves performance by reducing the number of slow system calls.
- If an error occurs before flushing, buffered data may not be displayed.

```c
#include <stdio.h>

int main() {
    printf("Hello
   World!");

    int *p;
    p = NULL;
    *p = 1;
}
```

```c
#include <stdio.h>

int main() {
    printf("Hello
   World!\n");

    int *p;
    p = NULL;
    *p = 1;
}
```

```c
#include <stdio.h>

int main() {
    printf("Hello
   World!");
    fflush(stdout);
    int *p;
    p = NULL;
    *p = 1;
}
```

Understanding Terminal Buffering

- Consider the following code:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
    for (int i = 0; i < 2; i++) {
        fork();
        printf("Hello World!\n");
    }
    for (int i = 0; i < 2; i++) {
        wait(NULL);
    }
}
```

- When we run `./a.out`, we get 6 `"Hello World!"`
- When we run `./a.out | cat`, we get 8 `"Hello World!"`

Understanding Fork

- fork() duplicates everything in the process, including the output buffers!
- When we run `./a.out | cat`, we get 8 `"Hello World!"`
  - It's connected to a pipe, not directly to the terminal.
  - The buffer does not flush after each newline; it waits until it's full or the program ends.
  - When fork() is called, the parent process's buffer (which may contain `"Hello World!"`) is duplicated in the child process.
  - Both parent and child processes may flush the same buffered data, leading to duplicated outputs.
  - Result: Buffer duplication causes additional outputs; 8 "Hello World!" are printed.

Principles of Device Driver Design

How can we use a computer to launch a nuclear missile?
Have A Try: launcher

1. Define the Driver's Purpose

Key Questions to Define:

- What operations will the driver support? (e.g., read, write)
- How will the device interact with the user or system?
- Example: Our nuclear missile code triggers an ASCII art when the password is received.

2. Register Device Number and Class

- **Major Number:** Specifies the device driver type.
  - Tells the OS which driver to use for this type of device.
  - Assigned by `alloc_chrdev_region` and stored in `dev_major`.
- **Minor Number:** Differentiates device instances.
  - Allows the driver to handle multiple devices of the same type.
  - Specified by `MKDEV(dev_major, i)`, creating instances like '/dev/nuke0' (Minor 0) and '/dev/nuke1' (Minor 1).

```
    // allocate device range and  create major number
    alloc_chrdev_region(&dev, 0, 1, "nuke");
    dev_major = MAJOR(dev);

    launcher_class = class_create(THIS_MODULE, "nuke");
    cdev.owner = THIS_MODULE;
    for (i = 0; i < NUM_DEV; i++) {
        // register device
        cdev_init(&devs[i].cdev, &fops);
        cdev_add(&devs[i].cdev, MKDEV(dev_major, i), 1);
        device_create(launcher_class, NULL, MKDEV(
   dev_major, i), NULL, "nuke%d", i);
    }
```

3. Define File Operations Interface

File Operations:

- Define how users will interact with the device:
  - **read:** Retrieves data from the device.
  - **write:** Sends data to the device.
  - **open/release:** Opens and closes access to the device.

```
static ssize_t launcher_read(struct file *, char __user
    *, size_t, loff_t *);
static ssize_t launcher_write(struct file *, const char
    __user *, size_t, loff_t *);

static struct file_operations fops = {
    .owner = THIS_MODULE,
    .read = launcher_read,
    .write = launcher_write,
};
```

Core Functions Based on file_operations:

- Each function provides specific device behavior:
  - **launcher_read:** Returns "This is dangerous!" when read.
  - **launcher_write:** Checks input for specific pattern to trigger ASCII art.

```
static ssize_t launcher_write(struct file *file, const
    char __user *buf, size_t count, loff_t *offset) {
....
if (strncmp(databuf, "\x01\x14\x05\x14", 4) == 0) {
     printk(KERN_INFO "nuke: correct password entered.\n")
    ;
     const char *EXPLODE[] = {
       //Your ASCII art
     };
     int i;
     for (i = 0; i < sizeof(EXPLODE) / sizeof(EXPLODE[0]);
      i++) {
       printk("\033[01;31m%s\033[0m\n", EXPLODE[i]);
     }
   } else {
```

5. Initialization and Cleanup

- `module_init`: Registers device and sets up file operations.
- `module_exit`: Unregisters device and cleans up resources.

```
static void __exit launcher_exit(void) {
    device_destroy(launcher_class, MKDEV(dev_major,
    0));
    unregister_chrdev_region(MKDEV(dev_major, 0),
    MINORMASK);
    class_unregister(launcher_class);
    class_destroy(launcher_class);
}

module_init(launcher_init);
module_exit(launcher_exit);
```

Device Driver Design Summary

Key Steps in Device Driver Design:

- Define purpose and main functions.
- Register device (major and minor numbers, class).
- Set up file operations interface (read, write, ioctl).
- Implement core functions for device-specific logic.
- Initialize resources and provide cleanup routines.

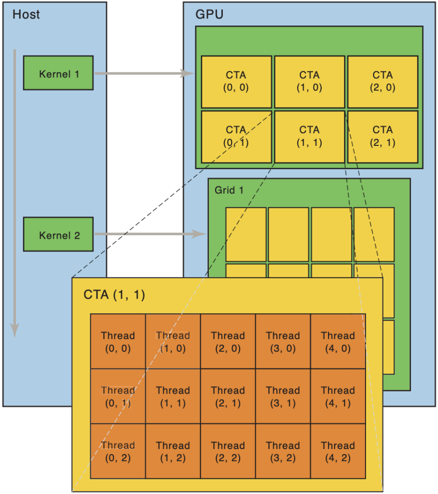
Install the Driver

- `make`
- Load module: `sudo insmod nuke.ko`
- Check '/dev' for device files ('/dev/nuke0', '/dev/nuke1') using `ls -l /dev/nuke*`
- Set the devices to be readable and writable with `sudo chmod 666 /dev/nuke0`
- Test with `echo "COP4610" | sudo tee /dev/nuke0`
- Observe logs with `dmesg | tail -n 20`
- Unload the module with `sudo rmmod nuke`

CUDA: Programming for GPUs

Single Instruction, Multiple Threads

- Many threads execute the same instruction
- Each thread has some thread-local data (e.g., ID)
- Highly optimized design
  - One Program Counter (PC), multiple data elements
  - Successor to VLIW and SIMD architectures



Reference: Parallel Thread Execution ISA Application Guide

- CUDA uses an instruction set architecture (ISA) for parallel thread execution.
- The code is compiled into SASS (machine code).
  - Use `cuobjdump --dump-ptx` or `--dump-sass` to view assembly code.

Essential Tools in the CUDA Toolchain

- **gcc** → **nvcc**: NVIDIA CUDA Compiler, used to compile CUDA code.
- **binutils** → **cuobjdump**: Disassembler for CUDA binaries.
- **gdb** → **cuda-gdb**: Debugger for CUDA code.
  - Allows debugging directly on the GPU!
- **perf** → **nvprof**: Performance profiling tool to analyze GPU code.
- ...

CUDA brings a complete set of tools to the GPU, similar to what we have for CPUs.

GPU Driver Overview

GPU Drivers are Complex

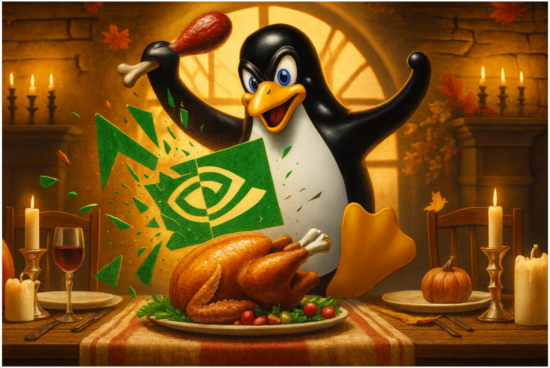
- Complete Toolchain
  - Just-in-time (JIT) compilation
  - Profiler
  - ...
- API Implementation
  - `cudaMemcpy`, `cudaMalloc`, ...
  - Kernel execution
  - Mostly implemented via `ioctl`
- Device Compatibility

Note: NVIDIA open-sourced its driver in 2022!

- Before that ... Video

Takeaways

- What are device drivers?
  - Translate `read/write/ioctl` calls into device-understandable protocols
- Device Driver Design Principles



Happy Thanksgiving!