Linux Kernel Programming (CS614A), Spring 2023
Indian Institute of Technology Kanpur
Assignment 3

REPORT 3

Members: Sarthak Rout
Roll Number: 190772
Date: April 20, 2023

Contents

1 Design

- We have used `IOCTL` to send information to and fro from userland library to the kernel module. The kernel module parses the `IOCTL` call based on request type and manipulates the device memory region to support encryption/decryption and other auxiliary services.
- Initially, while the probe is called, we setup the device for MMIO and DMA requests.
- For MMIO case, we directly write to device registers and read from them using `readl /writel/readb/writeb` APIs. We copy each byte one by one to the device memory and read them after performing the operation.
- For the DMA case, instead of copying each byte, we copy that user data to the DMA buffer, whose address is directly passed to the device, which then operates on it directly.
- To handle interrupts, we write a simple interrupt handler which reads the `ISR` register and writes the same to the `ACK` register.
- To store per-process information, a hash table is used which stores key and config info, if set by the process. Hash table implementation is separated from the main module code.
- To support MMIO with the address mapped to user-space, we map the whole device memory once to the process virtual address space. Then, for multiple `map_card` requests, we use the same virtual address and increase it by appropriate offset and return that. At the end, when all requests have called `unmap`, we will call `munmap` on the whole virtual address region.

2 Implementation

- When the driver is inserted into the kernel, a kernel probe function is called which can be used to setup the device for use by the driver. We map the `BAR0` region of 1MB with `ioremap` for use by MMIO. We also setup the DMA handle and buffer structures along with the interrupt handler.
- Locks are used when the execution comes to IOCTL call. They are important to serialize access to the device as multiple requests can't use the device simultaneously. They also control access to the hash table which should only be used by the process which is going on to access the device.
- Before parsing the call, the hash table is initialised if not done and the per-process info is passed to helper functions that use the information for that particular process and call.
- While communicating with user via `IOCTL` calls, we use `copy_from_user` and `copy_to_user` APIs to safely copy data.
- The hash table is implemented as a array of linked lists with insertion done at the head of the linked list. The key is calculated by a hash function that uses the last 10 bits of the PID. the hash tables should be used with locks.
- For the userspace mapped MMIO case, we iterate over VMA regions to suitably find a region where we can remap a region of 1MB to the device memory. While remapping the region, we also change the page permission bits by walking the page table to get the PTE.
- At the end, before the module exits, we free up the kernel allocated data structures.

3 Test Strategies

- We write some basic test cases for all configurations: MMIO with/without interrupt, DMA with/without interrupt, alongwith multithreaded cases and userspace mapped MMIO.
- For the multithreaded case, we create a loop over 10 threads and choose different keys, configs and strings. We check all possible cases here execpt userspace mapped MMIO.
- We map the card to userspace multiple times and use MMIO to write to different buffers on device memory from userland. Then, we encrypt and decrypt to test the implementation.
- We run a loop over the same text and encrypt it for 5 times with different keys and then we decrypt the same text with the same keys are above in different order. Even if the order is not changed, the final decrypted result is the same as the original message. This implies that possibly the encryption/decryption operators are linear transformations.

4 Benchmark Results

With a shared memory file of 100 MB,

Benchmarks	Avg. CPU Utilisation
mmio	**98.3%**
mmio_interrupt	**100%**
dma	**103%**
dma_interrupt	**0.4%**
mmap	**95.1%**
mmap_interrupt	**74.1%**

4.1 Explanation

- In MMIO (with or without interrupt), we copy bytes one by one which results in high CPU utilisation.
- In DMA without interrupt, we have to keep on polling the DMA status register, which results in high CPU utilisation.
- Since `dma_interrupt` technique doesn't use CPU as it uses DMA engine and doesn't rely on the CPU polling the device regularly, it has lowest CPU utilisation.
- `mmap` and `mmap_interrupt` results in significant CPU utilisation. The interrupt variant uses less CPU showing that polling v/s interrupt is an important design decision and interrupt based methods use less CPU.