Programming Assignment 3
(PCI Device Driver)
CS614: Linux Kernel Programming
2022-23 - Semester II
Computer Science and Engineering
Indian Institute of Technology Kanpur
Due Date: 19 April 2023, 11:55pm

April 5, 2023

This programming assignment is designed to make you acquainted with the working principle of a PCI device along with its integration in an end-to-end manner. In this assignment, you need to write a device driver for a PCI device called `CryptoCard` and provides a user-space library through which an end-user can utilize the functionalities of `CryptoCard`. An end-user can use the device in all possible scenarios, i.e., single process, multi-process and multi-threaded scenarios.

CryptoCard: Overview and Specifications

Here, we provide an overview of the CryptoCard along with its specifications.

Overview

CryptoCard is a PCI device that encrypts/decrypts the data by using a specified key pair `a` and `b` and returns the encryption/decryption result to the end-user. CryptoCard uses a single PCI region, i.e., `BAR0`, and have one MB of input/output memory through which a user can communicate with the device. The device can perform encryption/decryption using MMIO or DMA. Both MMIO and DMA can be configured to raise an interrupt after the operation is finished by the device. In the case of non-interrupt mode, user needs to periodically check the status bit of the operation to get the result back. A list of services provided by the CryptoCard is enlisted bellow.

1. Encryption/Decryption with interrupt using MMIO
2. Encryption/Decryption without interrupt using MMIO
3. Encryption/Decryption with interrupt using DMA
4. Encryption/Decryption without interrupt using DMA
5. Set Key pair `a & b`
6. Status of the operation
7. Set length of the data
8. configure address for the data message (MMIO/DMA)

Apart from the PCI region, device has an internal buffer of 32 KB to store the intermediate results during an operation. This internal buffer is being used for both MMIO and DMA. Once an operation is triggered, device will not accept other request to change the address of data. Hence, user has to handle subsequent requests at driver or library level.
Note: MMIO/DMA and with/without Interrupt mode is a global configuration. For example, assume there are two processes A and B. A sets the configuration of CryptoCard to enable DMA mode with Interrupt. After that, if process B starts its operation but doesn't set configuration, the operation of B will be done using the recent configuration i.e. DMA mode with Interrupt (set by A). Also, when application starts and there is no explicit configuration done, then the default configuration of CryptoCard will be MMIO without interrupt.

Setting up the device for a VM

This CryptoCard device is implemented in a custom Qemu, and same has to be used for the assignment. There are two approaches to use the custom Qemu.

Approach 1: Use the static build: You can use the statically linked custom Qemu present in the `qemu-static` directory. This should be the preferred approach as it will save you the effort of building custom Qemu from source.

Approach 2: Build Custom Qemu from source: You can download/clone the custom Qemu source code from the link https://git.cse.iitk.ac.in/skmtr/CS730-Qemu-PA. To utilize the CryptoCard, you need to build the custom Qemu image and use the script *CS730-Qemu-PA-master/start_vm.sh* to run an operating system image. To build the custom Qemu image, perform the following steps,

- cd into CS730-Qemu-PA-master directory.
- $mkdir build
- $cd build
- $../configure –python=/usr/bin/python2.7 –target-list=x86_64-softmmu
- $make (Will take some time to complete.)

Starting your VM using custom Qemu

- Copy your VM's disk image file (with extension .qcow2 or .img) file from its original location (default `/var/lib/libvirt/images/`) to `qemu-static` directory. Let's assume that the name of this file is `"cs614.qcow2"`. You may skip this step if you want to use the existing VM image directly.
- Change permissions of this file to 0666 (i.e. rw_rw_rw_). $sudo chmod 0666 cs614.qcow2
- Edit the start_vm.sh file. Replace #disk_image= line by #disk_image=./cs614.qcow2
- Edit the last line of the start_vm.sh file. Replace "root=/dev/sda2" based on the root partition in the `cs614.qcow2` disk image.
- Now, start the VM using $./start_vm.sh. SSH into the VM (in another terminal) using the command `ssh -p 5555 <username>@localhost`

At this point, you can check the output of `lspci` command to find a device with `vendor:device` pair as `1234:deba`.

Note: If you are not able to reach this point in setup, contact us as soon as possible.

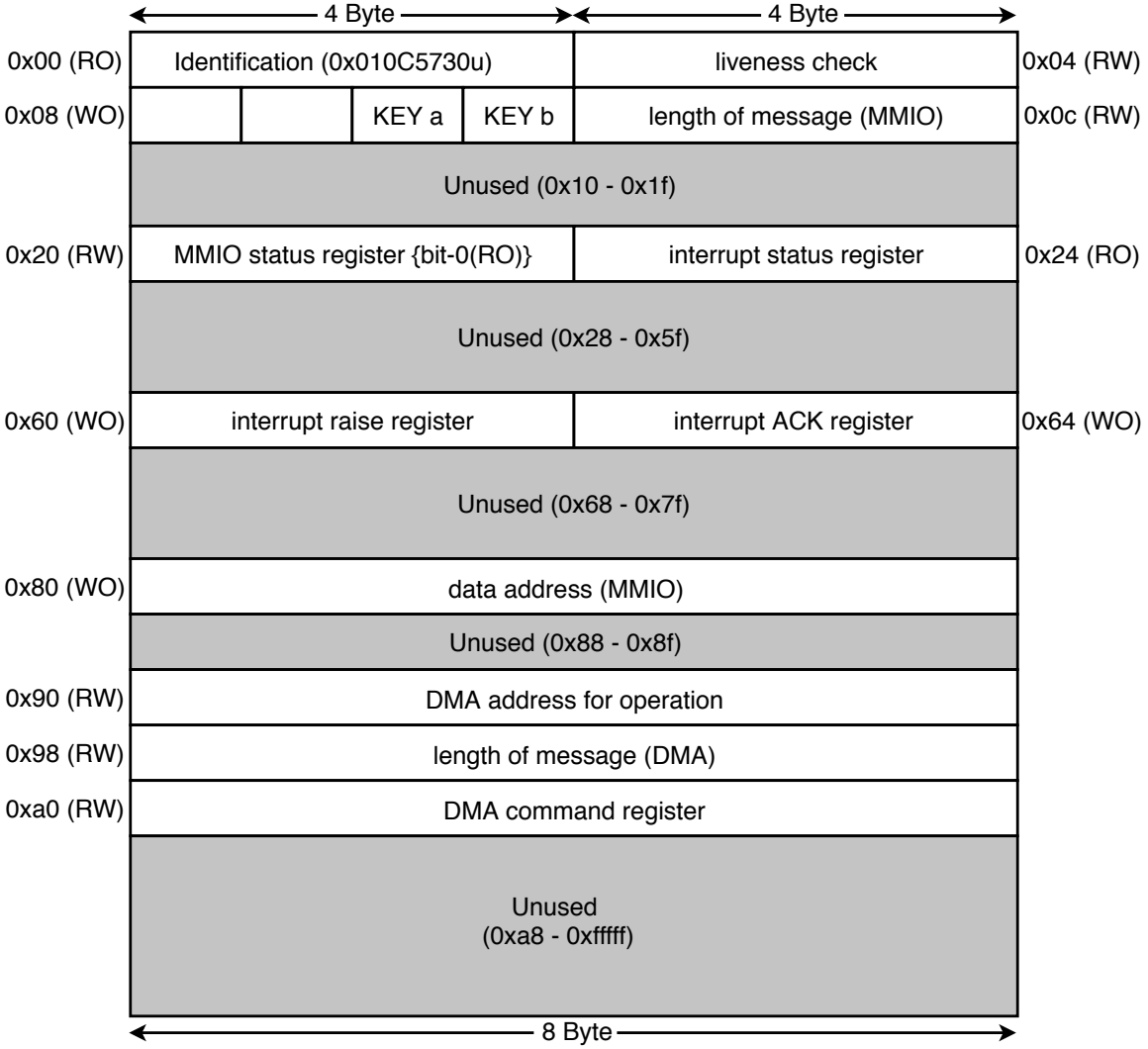

Figure 1: CryptoCard memory region.

Device Specification

Figure 1 shows the one MB of input/output memory layout that is used to communicate with the CryptoCard. As mentioned previously, the 1MB region can be accessed using MMIO through the PCI BAR0. Device has registers of two sizes —*(i)* 4 byte, and *(i)* 8 byte. Device input/output memory below address 0x80 can only accommodate 4 byte while addresses above 0x80 can accommodate 8 byte of data at a time. As shown in Figure 1, operation on device registers or memory regions can be performed on three ways, i.e., (i) read only (RO), (ii) write only (WO), and both read & write (RW). Some of the memory regions of the CryptoCard is unused, that can be utilized by the device driver if needed.

The description of the rest of the memory region (registers) are as follows:

- **identification**: This holds the identification of the device and can only be read. On read, it always returns a static value `0x010C5730u`.
- **liveness check**: To check whether the device is live or not. You can write some value (4 byte) on it. When you read the value, this will return the complement of earlier provided value, which indicates that the `CryptoCard` is live.
- **KEY**: This is a write only register and holds the key for encryption/decryption operation. Both

MMIO and DMA utilizes this for the corresponding operation. It holds both components of the key, i.e. `"a"` at byte1 and `"b"` at byte 0.

- **length of message (MMIO)**: It represents the data size on which an encryption/decryption operation has to be performed through MMIO.
- **MMIO status register**: It gives status of the operation for MMIO and can also be used to configure operation type and interrupt. A user can read from and write to this register except the bit 0, which is read only and used to denote the status of operation. Apart from status bit, user can write to other bits. The bit corresponding to operations are shown in table 1:

Bit position	Meaning
0	Read returns 0 if device is free else returns 1
1	Set value to 0 for encryption and to 1 is decryption
7	Set value to 1 to raise interrupt after finishing operation and write 0x001 to interrupt status register on completion

Table 1: Details of MMIO status register Bit Positions

For example:
If the user want to perform decryption using MMIO and raise interrupt, he/she has to write `0x80|0x02` to MMIO status register.

- **interrupt status register (ISR)**: It contains values corresponding to MMIO/DMA. `0x001` indicates MMIO and `0x100` indicates DMA.
- **interrupt raise register**: Can be used to raise interrupt manually for testing.
- **interrupt ACK register**: Writing to this register clears an interrupt. You should write the value present in ISR to clear an interrupt. This is needed to stop generating interrupts.
- **data address**: The stored value contains the address of the data to perform Encryption/Decryption and the result of Encryption/Decryption is put back to the same buffer. This happens only after bit 0 in the MMIO status register (refer table 1) is cleared. Note that writing to this address triggers the Encryption/Decryption process through MMIO.
- **DMA address for operation**: Address to perform Encryption/Decryption and store the result.
- **length of message (DMA)**: It represents the data size on which an encryption/decryption operation has to be performed through DMA.
- **DMA command register**: It controls the DMA operation and uses a bitwise OR operation to combined multiple command. The bit corresponding to operations are shown in table 2:

Bit position	Meaning
0	By setting it to 1 triggers the operation through DMA
1	Set value to 0 for encryption and to 1 is decryption
2	Set value to 1 to raise interrupt after finishing operation and write 0x100 to interrupt status register on completion

Table 2: Details of DMA command register Bit Positions

Note that, the unused address range starting from `0xA8` to `0xFFFFF` can be used to host the buffer in MMIO mode. This should not be used by the software when performing DMA.

Task to be done

Till now you were getting familiarized with the functionalities and specification of `CryptoCard`. The real task of the assignment is to design and implement build below mentioned two core components,

- Device driver for `CryptoCard`
- An user-space library interface to the device

Device Driver

Your device driver should implement all the functionalities mentioned in the overview section. All these functionalities should be configurable from the user-space through any of the known methods (e.g., chardev etc.). Apart from these specified functionalities, you should also provide a functionality to map device memory into user-space and read/write data directly.
You are free to follow any good coding style for the device driver implementation as no skeleton code is provided by us. We will be using your user-space library for evaluation. We will evaluate expected functionalities using multiple test programs with different scenarios like a single process, multi-process, etc. Preferably, split the driver code into multiple files and build a single kernel object file.

Library

In this part, you have to implement a shared library using the provided skeleton code in the `template` directory.

Go through the `README` file to understand the directory structure and build process. It contains one test case or example program that uses your library for performing encryption/decryption. You need to complete the definition of the library functions according to your device driver. You can not modify the signature of a library functions.

Testing Scenarios (80 Marks)

We will evaluate two scenarios—(i) Single process and (ii) multi-process/multi-threaded.
Test cases for both the scenarios are discussed bellow.

Single Process Test-cases (60 marks)

Single process test-cases are as follows:

1. Encryption/Decryption through MMIO without interrupt. (10 marks)
2. Encryption/Decryption through MMIO with interrupt. (10 marks)
3. Encryption/Decryption through DMA without interrupt. (10 marks)
4. Encryption/Decryption through DMA with interrupt. (10 marks)
5. Encryption/Decryption through MMIO where the device memory region is mapped to the user-space. User will write data directly to the mapped memory and triggers the operation. (20 marks)

Multi-process/multi-threaded test cases (20 marks)

1. Encryption/Decryption through MMIO without interrupt. (5 marks)
2. Encryption/Decryption through MMIO with interrupt. (5 marks)
3. Encryption/Decryption through DMA without interrupt. (5 marks)
4. Encryption/Decryption through DMA with interrupt. (5 marks)

Assignment Report (20 Marks)

Along with creating library and device driver, you also need to create a report. There is no fixed structure of report. Some of the things that can be included in it are:

1. Design
2. Implementation details
3. Test strategies (How did you test your solution)
4. Benchmark Results.

More about Benchmark Results

You have to submit the CPU utilization report using benchmarks with DMA and MMIO. We have provided the user-level code (`template/benchmark-test/`) which should be used to record the benchmarks.

1. Run the make command in the root "template" directory to generate the executables such as dma, mmio, mmap etc.
2. Execute the command "./create-dev FILE_SIZE_IN_MB". This will create an in memory file in /dev/shm folder. For example: "./create-dev 100" this command will create a file size of 100MB. If the command-line argument is not provided. Then the program will create a 10 MB file by default.
3. There will be 6 executables (dma, dma_interrupt, mmio, mmio_interrupt, mmap, mmap_interrupt). You have to execute each one and report the CPU utilization numbers along with your submission.
4. You can use SAR/iostat tool to record the CPU utilization.

Deliverable:

1. Assignment report.
2. Source code of `library` and `device driver` along with the makefile for the driver code. You should put drivers code inside the directory `drivers` of the library skeleton code along with its makefile. Your makefile should generate drivers kernel module with the name `CryptoCard_mod.ko`. During the evaluation, automated script will search only for the driver module with the name as mentioned above.
3. Submit the tarball of your assignment named as your roll number in Canvas. You can use the Makefile provided in the library code to prepare tarball. The procedure for preparing tar file using makefile is listed in the `README` file.
   (Check whether script adds project report also into the tarball. Maybe provide specific instructions for directory structure.)