CS614 Assignment 3
Name: Nibir Baruah (19807545)

DESIGN AND IMPLEMENTATION DETAILS

To crypto_card device takes help of an implemented character device using the IOCTL system call facilities. The appropriate flag is passed to the IOCTL along with the required parameters to distinguish between different cases. To handle multi-threaded/multi-process test cases, hash map of size 1000(Number of hash buckets) is maintained which will store the per thread/process key and config details. The per process structure is of type $struct\ node$. There can be three types of inserts to this hash map. The first type($INIT\_INSERT$) is when any process enters the ioctl handler for the very first time. This will set the dma, interrupt and key set flags to 0; KEY_INSERT and $CONFIG\_INSERT$ is the type of insert which is called when setting the key and config settings respectively. The expectation is that the first insert will already create a structure in the hash map, and the latter two calls has to just set the required values in that structure. All the modifications to this hash map occurs in critical sections of semaphore locks. Below the different components of the driver are explained:

- **Setting the keys:** When the user calls $set\_key()$ library function, the keys are passed along with the IOCTL call. At the kernel side, the passed keys from the user space are wrote to the $struct\ node$ structure for that process and also to the specified location in the device memory(BAR0). Here, a flag variable is also set in the per process data structure to indicate that the keys have been set. This information will be useful during encryption and decryption.

- **Setting the config details:** In $set\_config()$ function, the config type and value is passed to the IOCTL call. The IOCTL call handler will write these details(setting DMA on/off and interrupts on/off) to the hash map.

- **Encryption and Decryption:** The $encrypt()/decrypt()$ functions are called by the user to encrypt/decrypt any text passed appropriately to the device. Here intially we check whether the key set flag has been set this for this process. If it has been set, we first write the stored keys in the hash map to the specified memory in the device (As it may have overwritten by the keys of some other process due to multi-threading). Then the user passed $isMapped$ flag and the DMA and interrupt settings are checked. Accordingly the handler functions are called with appropriate parameters(message text and length).

- **MMIO Handling (w/o user space mapping):**Here, the passed user space arguments are first copied to a kernel data structure. Then the message length is set in the device memory and the status bits(this is done differently for encryption and decryption) are set. The message is copied from the kernel buffer to the unused device memory and this unused memory offset is written in the MMIO data address offset of the device memory region. After this, if we use polling a while loop, runs, otherwise a semaphore $down\_interuptible()$ function is called, to wait for the completion of the encryption/decryption process.

- **DMA Handling (w/o user space mapping):**Here all the things are same as in MMIO handling, except the data need not be copied. In the MMIO data address, we write the DMA address that we passed to the $dma\_alloc\_coherent()$ function in the device probe function.

- **Map card and unmap card:**When the user calls $map\_card()$, we first get an user address to suggest to the mmap system call. This address is obtained by substracting some apropriate value from the $vma- > vm\_start$ of the very first VMA. This is done so that no merging happens with any other VMA(This was required for calling the $io\_remap\_pfn\_range()$ function). This address is then passed to the mmap system call. Then we do an IOCTL call to map this virtual adress to the required device memory region. $io\_remap\_pfn\_range()$ is called to do the mapping and the pte entries are also properly set with the required permission bits. $unmap\_card()$ basically just calls the munmap system call to unmap the allocated memory area. The case for multiple map_card calls is also handled here by using a count reference, and doing the munmap system call when the $map\_card()$ and $munmap\_card()$ calls match exactly. We maintain a global user address in the library file to indicate the current start address. This is updated accordingly by the $map\_card()$ calls by adding the size.

- **MMIO Handling (with user space mapping):** This case is similar to the MMIO handling case without user space mapping except that in the MMIO data address offset we write the difference between the passed user space virtual address in the encrypt/decrypt function and the globally maintained base address variable which was set during the map card IOCTL call.

- **Interrupt handling:** There is a common interrupt handler function that just reads the value at the interrupt raise register of the device and writes it to interrupt ACK register. It also calls the semaphore $up()$ function to wake up the sleeping/waiting encryption/decryption process.

TEST STRATEGIES

The single threaded test cases were simple and made by using $set\_key()$ and $set\_config()$ with the different options and key types. Also multiple levels of encryption were done on a given text, followed by the same number of decryption with the key order reversed. This generated the original text. For the multi-threading checking, 10 threads were spawned with each running encryption/decryption on different key and config settings. For the $map_card()$ and $unmap_card()$ testing, multiple calls were overlapped and each user address given different strings to encrypt and decrypt.

BENCHMARK RESULTS

Executable	Average CPU Utilization
mmio	99.8%
mmio_interrupt	99.7%
dma	99.8%
dma_interrupt	0.5%
mmap	105%
mmap_interrupt	77%