Assignment-3

Name: Qin Peiran Student ID: 119010249

Part 1: Environment

- My project adopts the following environment:
  - version of OS:

```
qpr@ubuntu:~/Desktop/Assignment_1_119010249$ cat /etc/issue
Ubuntu 16.04.5 LTS \n \l
```

  - version of kernel:

```
qpr@ubuntu:~/Desktop/Assignment_1_119010249$ uname -r
4.10.14
```

  - CUDA Version: 11.5
  - GPU information:

```
NVDIA Geforce RTX 3050 Laptop GPU
```

Part 2: the steps of running my program

- First compile the whole program: There is a shell script in the directory:

```
$ sh run.sh
```

- Then run the program by:

```
$ ./main.out
```

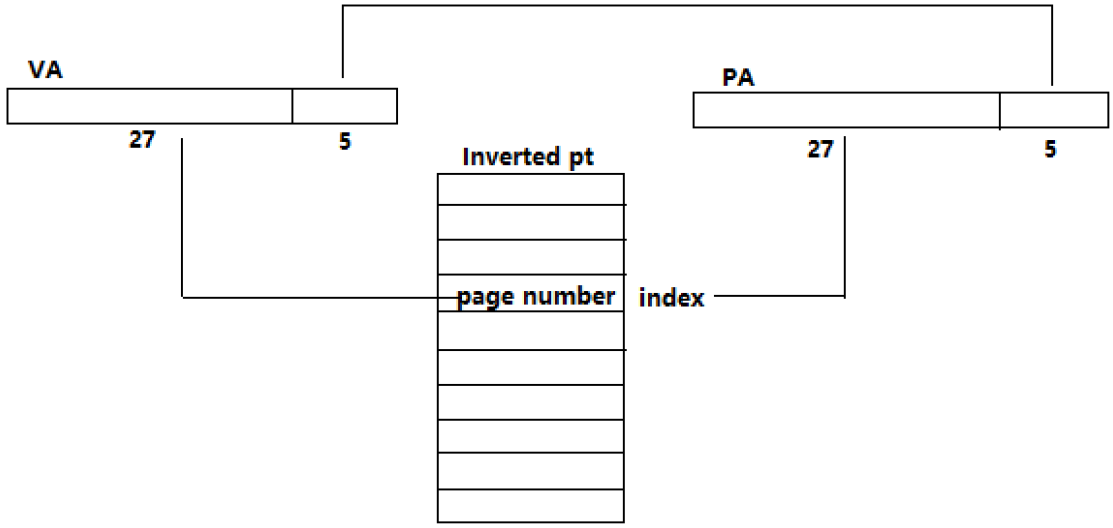
Part 3: Design Appoach

- The overall idea is to simulate the virtual memory using GPU and CPU. And there are mainly four parts of memory in this project:
  - Input buffer
  - Result buffer
  - Physical memory
  - Disk storage

  In the operations of `read` and `write`, different memory plays different roles. I will explain the connection between each memory part and how they cooperate with each other in detail.

- Logical address and physical address

- The logical address is converyed to physical address by page table: the principle is shown below:



Remark:

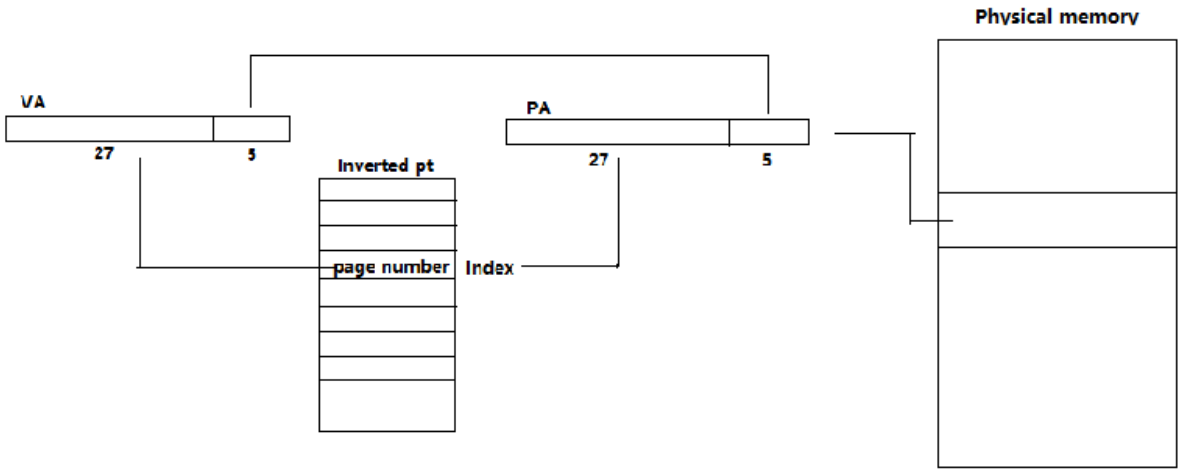
- The length of offset is 5 because the page size is 32 byte. Therefore the remainning bits can be used for the page number(base).
- As for the inverted page table. It has to iterate the whole page table and find the corresponding page number. Then, the index of that page number is the base address of the physical memory.

With the physical memory address, the `vm_read` and `vm_write` operation can performed on the physical memory space.

- vm_read( )

- There are three situations for this function: 1. the page is already loaded in the frame of physical memory. 2. there is an empty frame in the physical memory 3. page fault

1. When the page is already loaded in the physical memory, means the page number can be founded in the inverted page table. So the step of `vm_read` is to convert the virtual address into the physical address. Then using the physical address to directly read the data from the physical memory.

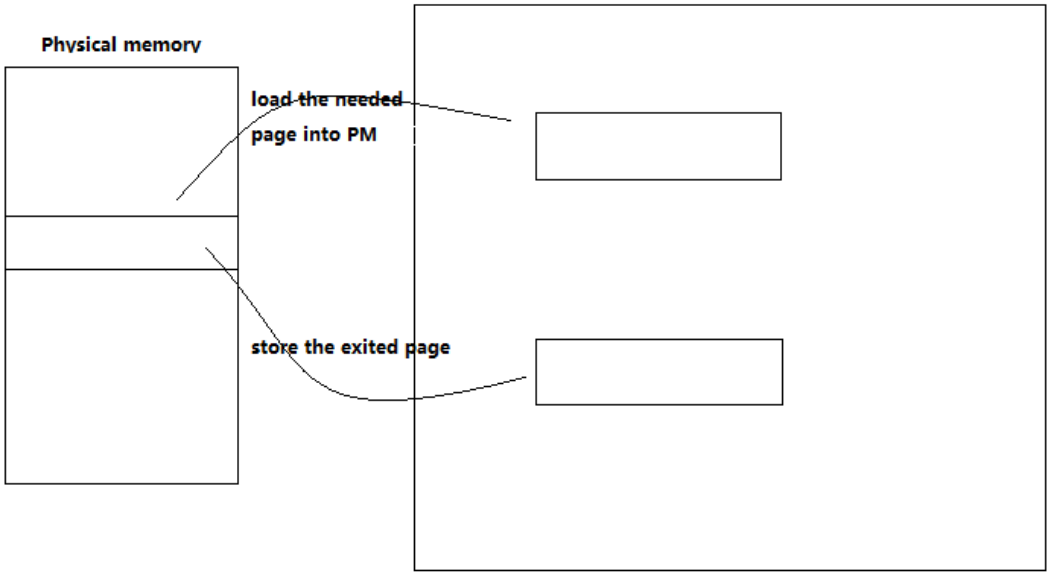


2. When there is page number cannot be found in the page table, my project will first allocate the page in the empty frame in the physical memory. The available bit indicates whether the

frame in physical memory is empty or not. After it allocate the page to the empty frame, it will update the valid bit and page number in the invert page table. And load the page into this frame.

3. When page fault. It means that the frame number cannot be found in the inverted page table (the corresponding page hasn't been loaded to the physical memory). To deal with the page fault. It has to do the swapping of page.

- Swapping of page:
    - First, it will using the **LRU** strategy. There is an array, which counts the frequence of use of each pages in physical memory. Every time when there is an `read` or `write` operation, the count of the corresponding page will be set to 0. Other pages will be decrease 1. Therefore, the one will least count will be the one which is the least frequent used. Noted that when doing the swapping, the page whose valid bit (means this memory location is empty) is 1 will be used first. After all the empty pages in memory locations being used, the least frequently used one would be replaced.
    - Then, after deciding which page to be swapped. It will be first load back to the disk storage. Then clear the corresponding loaction in physical memory. And finally load the needed page into the physical memory.
    - What's more, the page number in the inverted page table should also be changed to the new loaded page's number.
    - The following is the illustration of swapping:



- Then aftering swapping the page, restart the operation of vm_read( ), and this time it can directly read the data from the frame in physical memory since the page fault has been resolved.

- vm_write( )

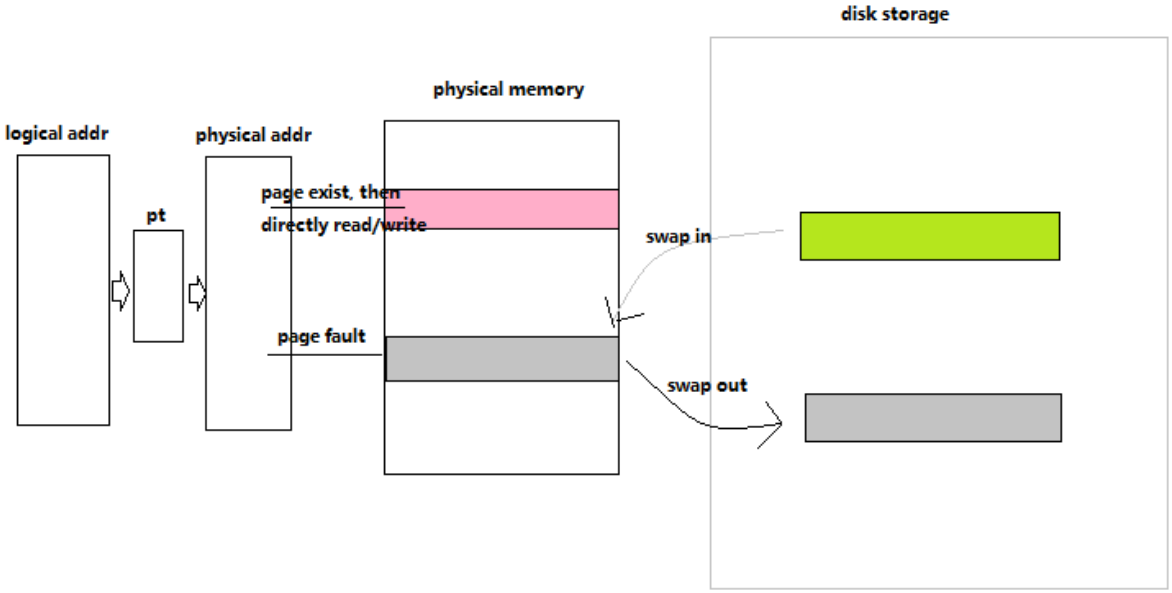
- As for the vm_wrtie, if page fault doesn't happen, it would directly write the data into the availble frame in physical memory. However, if there is a page fault, it will first allocate the empty frame in physical memory to the page, otherwise it will do the swapping of pages or as the one in `vm_read( )` does. Also the write policy I adopted here is: vm_write will only write the physical memory but not write to the disk storage. Only when swapping, the newest data would be written back to the backstore (this policy is mainly for reducing the times of accessing disk storage).

- After swapping, it would write the data into the new-loaded page in physical memory.

- vm_snapshot( )

- The function of vm_snapshot( ) is to load the data in disk storage to the result buffer. However, it cannot directly load the data from disk storage, instead, it should load data from the physical memory. Therefore, it use the logical memory and via `vm_read( )` to load the data.

Here is a **overall design** of memory management:



where the input buffer (which provides logical address and value) and the result buffer (which provides destinate logical address) are not included in this picture.

- Multi-threads

- In this part, 4 threads were launched to concurrently do the vm_read and vm_write of the problem. The code of launching 4 threads in one block is:

```
mykernel<<<1, Thread_Num, INVERT_PAGE_TABLE_SIZE>>>(input_size);
```

- And here, we assume that all these four pages use the same page table and physcal memory. Therefore, to distinguish the pages each threads are using, the I introduce `pid` to the inverted page table, which represents who is this page belong to.
- Therefore, the paging mechanism has to be modified: to judge whether there is a page fault, not only it should consider whether the page number is in the invert page table, but also it should consider whether the pid of this page is the corresponding executed thread. Although the page number is the same, it would occur page fault when the page is belong to another thread.
- Then, because of the race condion, we set the priority of each threads, means the thread 0 will first do its operations. After that it's thread 1 and so on. And when one thread is performing, other threads should have to wait. This idea is implemented by `__syncthreads( )`, which guarantee that when other threads would wait until one thread finish its work.

Part 4: The page fault number of my output

- The page fault number is 8193 in this project.
- The reason is that:
    - First the user program use vm_write to write the data into the buffer. This data total are included in 4096 pages. And because the iteration is from 1 to 131072, means only when one page is full, it will reference to next page. Therefore, there will be totally 4096 pages fault, because all 4096 pages should be loaded from disk to physical memory empty frame and after it's full, the least frequently used one will be written back to backstore.
    - Second, the there are iteration of vm_read( ), which will read last 32769 bytes. Since it is right after the vm_write( ), the pages of last 32768 bytes are already in the physical memory, which will cause no page fault. However, there is 1 byte that is not in the physical memory, so 1 page fault occurs here.
    - Finally, user program uses `vm_snapshot( )` to load all the data in disk storage into the result buffer. Since it read data from 1 to 131072. The situation is same as the first part (iteration of vm_write( )). Therefore, there will be 4096 page faults here.
    - Totally, there will be 8193 page faults in my project.

- For Bonus:

- In bonus part, the page fault number would be 32772. That's because all the threads are repeatly performing the same user_program. Meanwhile, although the previous thread has loaded some pages into the physical memory and set the valid bit to 0, because of the `pid`, page fault will still happen at the begining.
- In general, the page fault number of the bonus part would be 4 times of the sequential one. And the final number is 32772.

Part 5: the problems I met and my solution

- The first problem I met is how to convert the logical address to the physical address, because, I first don't know how to use the inverted page table and how to map the page base address.
    - The solution is that I first recognize that the page size is 32 byte, which means I should use 5 bits of address to be the offset of page. Then the remaining 27 bits could represents the page number. Then I recognize that there are at most 4096 pages in this project because the maximum size of disk storage is 128KB, which means that 17 bits in the logical address can be used to indicate the page numbers. Then I review the principle of inverted page table, and find that its index corresponds to the base address of pages in physical memory. So the convertion is using the 17 bits of logical address(page number) to find the related index in the inverted page table. And combine with the offset, it can get the physical address.
- The second problem I met is how to deal with the page fault. I wondered if the page is not in the physical memory, how can I find it in the disk storage.
    - The solution is that I can also assign numbers to the pages in disk storage, so that I can find them via page number (so that there are totally 4096 page number). When the page fault happens, I can retrieve one and swap it out to the physical memory corresponding frame in physical memory, and update its page number in the inverted page table.

- The third problem I met is the algorithm of LRU strategy. Finally, I adopt the following strategy:
    - During each time of iteration (vm_read or vm_write), when the page is referenced, I will reset its reference bit to 0. While for others weren't referenced, I will decrease its reference bit by 1. Therefore, the least recently used page will have a smallest reference bit.

Part 6: Screenshot of program output

- One Thread:

```
C:\Users\86183\Desktop\csc3150_project3_newest_success>main.out
input size: 131072
pagefault number is 8193
```

And using `fc` operation to compare the data.bin and results.bin, they are the same:

```
C:\Users\86183\Desktop\csc3150_project3_newest_success>fc snapshot.bin data.bin
正在比较文件 snapshot.bin 和 DATA.BIN
FC: 找不到差异
```

-Four threads:

```
C:\Users\86183\Desktop\csc3150_project3_newest_success>main.out
input size: 131072
pagefault number is 32772
```

Part 7: Things I learned from project

- In general, I learned the whole hierarchy of memory management.
- I know how the virtual memory work and how the user can use virtual memory address to reference to the logical memory. The convertion is performed by the page table.
- I learned how to swap the pages between physical memory and disk storage, and under what situation I should do the swapping (page fault). During the swapping, I should first choose one page to exit, then stores it back to the diskstore. For the security, before loading the new page to physical memory, the exited frame in physical memory should be clear first. After that, the needed page would be loaded to the physical memory and the page table would be updated.
- Meanwhile, I learned several programming skills of CUDA. For example some key words `__shared__`, `__device__`, `__host__` and `__managed__`, they can help to distinguish the executed program and memory access of host(CPU) and device(GPU). What's more, I also learned how to launch blocks of threads and manage them to perform a more efficient execution.