CSC3150 Assignment3

119010054 Dai Yulong

1. Environments and Run Methods

1.1 Environments

My project adopts the recommended environment on TC301 computer:

- OS version

```
[cuhksz@TC-301-18 ~]$ cat /proc/version
Linux version 3.10.0-1160.42.2.el7.x86_64 (mockbuild@kbuilder.bsys.centos.org)
(gcc version 4.8.5 20150623 (Red Hat 4.8.5-44) (GCC) ) #1 SMP Tue Sep 7 14:49:57
UTC 2021
```

- CUDA version

```
[cuhksz@TC-301-18 ~]$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2019 NVIDIA Corporation
Built on Fri_Feb__8_19:08:17_PST_2019
Cuda compilation tools, release 10.1, V10.1.105
```

- GPU information

```
[cuhksz@TC-301-18 ~]$ lshw -C display
WARNING: you should run this program as super-user.
  *-display
       description: VGA compatible controller
       product: GP106 [GeForce GTX 1060 6GB]
       vendor: NVIDIA Corporation
       physical id: 0
       bus info: pci@0000:01:00.0
       version: a1
       width: 64 bits
       clock: 33MHz
```

```
       capabilities: vga_controller bus_master cap_list rom
       configuration: driver=nvidia latency=0
       resources: irq:143 memory:f0000000-f0ffffff memory:c0000000-cfffffff
memory:d0000000-d1ffffff ioport:3000(size=128) memory:f1080000-f10fffff
  *-display
       description: VGA compatible controller
       product: UHD Graphics 630 (Desktop)
       vendor: Intel Corporation
       physical id: 2
       bus info: pci@0000:00:02.0
       version: 00
       width: 64 bits
       clock: 33MHz
       capabilities: vga_controller bus_master cap_list rom
       configuration: driver=i915 latency=0
       resources: iomemory:400-3ff irq:140 memory:4000000000-4000fffffff
memory:e0000000-efffffff ioport:4000(size=64)
WARNING: output may be incomplete or inaccurate, you should run this program as
super-user.
```

1.2 Run Methods

For both two programs (`source` and `bonus`), I write makefile to compile it. So only two steps are needed to run the program.

1. Using `cd` to enter the `source` or `bonus` directory, then type `make` command and enter.
2. Enter `./main.out` to execute the program:

```
[cuhksz@TC-301-18 source]$ make
nvcc --relocatable-device-code=true main.cu virtual_memory.cu user_program.cu -o
main.out
```

```
[cuhksz@TC-301-18 bonus]$ make
nvcc --relocatable-device-code=true main.cu virtual_memory.cu user_program.cu -o
main.out
```

3. In the corresponding directory, type `make clean` command to clear the `main.out`.

2. Design Ideas

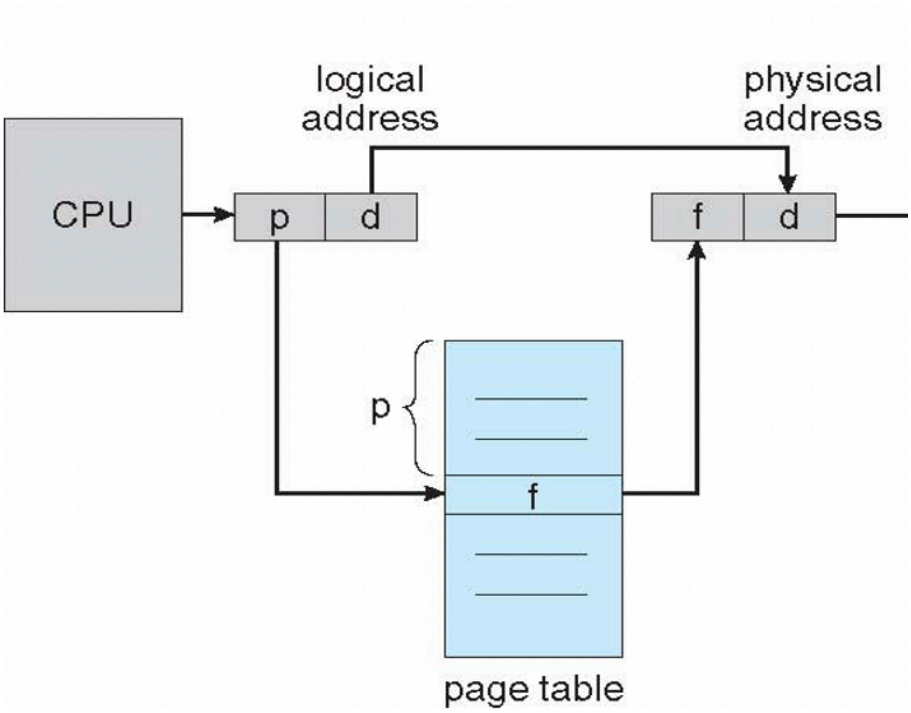
The assignment 3 mainly focus on simulating the mechanism of virtual memory via GPU's memory. And there are mainly three parts of memory in this project (listed terminologies in one line are equivalent):

- Input buffer (Data stored in `data.bin`): also known as virtual memory.
- Shared memory: It includes invert page table (16KB) and buffer (physical memory 32KB).
- Global memory: It plays the role of disk storage (also known as secondary storage).

These different level of memories cooperate with each other to simulate the machanism memory system. The following content will introduce VM and PM match machanism, design of invert page table, the program flow of `vm_read()`, `vm_write`, `vm_snapshot` and bonus.

2.1 VM and PM match machanism

The VM and PM match machanism can be briefly represented by this diagram:



In this question, the size of virtual memory (the size of data.bin) is 128KB and the size of physical memory is 32KB (defined as shared memory in cuda programming). And the page size is defined as 32 bytes, which means total number of pages in virtual memory is $\frac{128KB}{32bytes} = 4096$. And the number of frames entries is $\frac{32KB}{32bytes} = 1024$. Therefore, the p in above diagram is 27 and d is 5. The page number p is used to search the corresponding frame number f in the page table (in this question, the invert page table is used). The offset d remains the same.

2.2 The design of invert page table

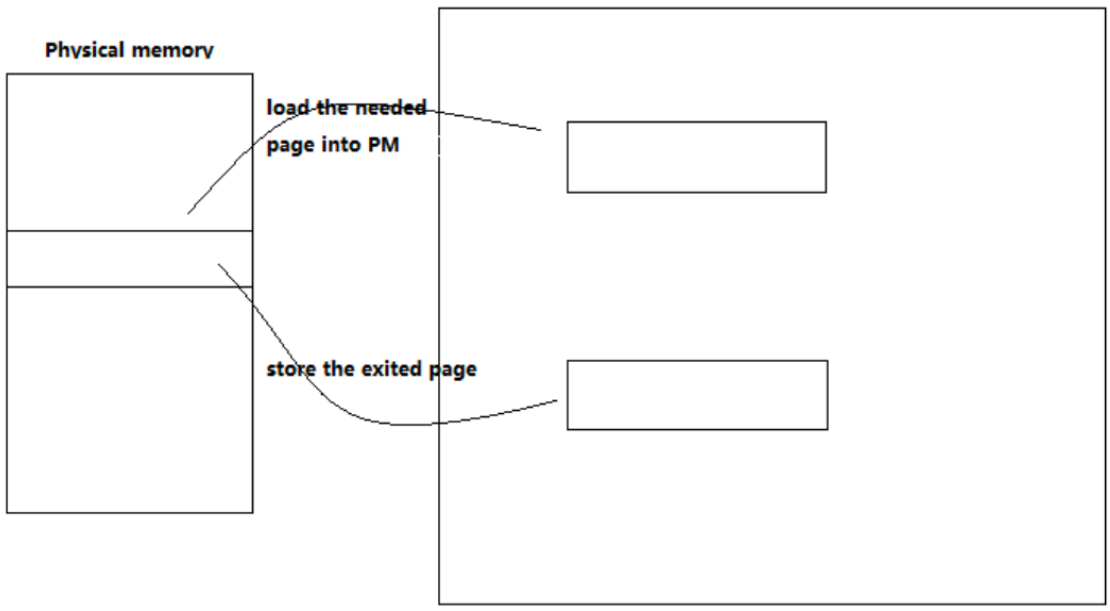
The size of invert page table is 16KB and it's defined as shared memory in cuda programming. To find the frame number, we have to iterate the whole invert page table and find the corresponding page number. Then, the index of that page number is the base address of the physical memory. To use all 16KB, the invert page table is divided into four parts (Each part is 4KB, and the four parts add up to just 16KB):

- **First 4KB**: It's used to store the valid bit, which indicate whether the corresponding physical memory frame is empty. It is initialized as 'invalid'. If the corresponding frame has data (write in), it will change to 'valid'.
- **Second 4KB**: This 4KB stores the page number. Here the page number and frame number (index) match one-to-one with each other. Seach the frame number here!
- **Third 4KB**: This 4KB stores the frequency clock, which count the time of last used. It's initialized as 0. Everytime a read or write operation occurs, all 1024 entries of frequency clock will increase 1. And the accessed entry of frequency clock will be cleared to 0. Thus, highest number means least recently used frame (swap out in LRU process).
- **Fourth 4KB**: It's used to store the pid in bonus part. In bonus part, in every read or write operation, the program first checks whether the pid is right or not.

2.3 `vm_read` implementation

The `vm_read` operation handles two situations:

- If the page has been already loaded in the physical memory, the frame number in physical memory can be easily found. So `vm_read` only need to convert the virtual address into the physical address, then use the physical address to directly read data from the physical memory.
- And if the page hasn't been loaded in the physical memory, a page fault occurs. To deal with the page fault, the `vm_read` use LRU strategy to swap the pages.
    - Firstly, the program uses frequency clock to get the least used block to swap the pages. Frequency clock is an array that implemented in the third 4KB of invert page table. Highest number in this array means least recently used frame. So the first step is to find corresponding least recently used frame.
    - Secondly, after deciding which frame to be swapped, first load the current value in that frame back to the corresponding position of disk storage. Then clear that value and load the needed frame from disk storage into the physical memory. At the same time, refresh the invert page table. Notice that I use direct mapping between virtual memory address and disk storage, cause both of them are 128KB.
    - The following diagram illustrates the swap process:



2.4 `vm_write` implementation

The `vm_write` operation handles three situations:

- If the corresponding frame exists, the program directly write the value into corresponding address (base + offset).
- If the correspongding frame doesn't exist and there is an empty frame, the program will find the least index empty frame to put the block into it. At the same time, refresh the invert page table and page fault number increases.
- And if the corresponding frame doesn't exist and there is no empty frame, an page fault occurs. The program swaps pages based on LRU strategy similar to vm_write process.

For the simplicity and readability of my code, I combine the paging process in `vm_read` and `vm_write` into a new function `paging`. It returns the final address in physical momory for reading or writing.

2.5 `snapshot` implementation

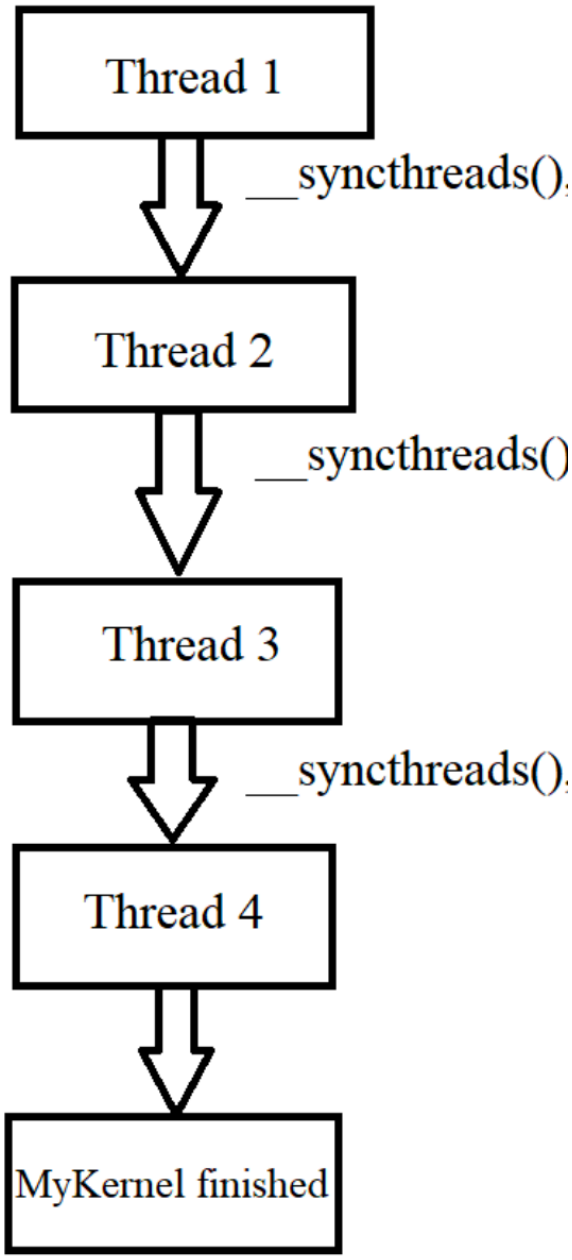
This function is to load the data in disk storage to the result buffer. However, it's impossible to directly load the data from disk storage. So, I call `vm_read` to load all data.

2.6 Bonus implementation

In the bonus part, 4 threads are launched to concurrently run `user_problem`. The code of launching 4 threads in one block is:

```
dim3 block(4,1);
mykernel<<<1, block, INVERT_PAGE_TABLE_SIZE>>>(input_size);
```

Using the pid in the invert page table, I can ensure that each thread call the right corresponding page table. Then, because of the race condition, I set the priority of each threads, means that the thread 0 first run the `user_program`. 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 finishes its work. The bonus logic of four threads is shown below:



3. The explanation of page fault number

3.1 For basic version, the page fault number is 8193 = 4096 + 1 + 4096.

- Firstly, in the process of `vm_write`, the `user_program` write all data (128KB) into the physical memory/disk storage. **In this stage, 4096 page faults are created**. For first 32KB data, because all frames are empty, it creates 1024 page faults. Then for the rest 96KB, each page needs a swap operation to swap out the least recently used page. So there will be other 3036 page faults. When the `vm_write` operations are over, the PM stores last 32KB data in the `data.bin`
- Secondly, in the process of `vm_read`, the `user_program` read 32KB + 1byte memory from the end of `data.bin`. **In this stage, there is 1 page fault**. Because at the end of first stage, the physical memory has already stored last 32KB data. So only the last byte need a LRU swap process to swap in, which increase the page fault number by 1.
- Thirdly, in the process of `vm_snapshot`, the `user_program` reads all data in the disk by calling `vm_read`. **In this stage, because it reads the total `data.bin` file, it creates 4096 page faults**. Every page in this stage needs a LRU swap process.

3.2 For bonus version, the page fault number is 32772 = 8193 * 4.

In the bonus program, since four threads run the `user_program` one by one and each thread has its independent page table, the number of page fault number is **8193 * 4 = 32772**.

4. Screenshot of sample output

4.1 The screenshot of basic version

```
[cuhksz@TC-301-18 source]$ ./main.out
input size: 131072
pagefault number is 8193
```

4.2 The screenshot of bonus

```
[cuhksz@TC-301-18 bonus]$ ./main.out
input size: 131072
pagefault number is 32772
```

5. The problems I met and my solution

- The first problem I met is how to I met is how to convert the logical address to the physical address. And I'm also confused about how to use the inverted page table and how to map the page base address. I read the book and ppts, and also draw a picture to find out the behind machanism.
- The second problem I met is how and where to implement the LRU strategy. I need a 1024 length array to store all frequency clock. And by observation, I find the current invert page table implementation is only 8KB. And there's 8KB left! So, I create this array in the invert page table and also store the pid in the invert page table. 4KB + 4KB + 4KB + 4KB = 16KB! So I perfectly use the 16KB invert page table size.

6. Learning

After finishing the assignment3, I have an overall and deep understanding of memory management.

- I have developed high-level programming skills of CUDA. For example, I learn what is `__shared__, __device__, __host__, and __managed__`, which can distinguish the executed program and memory access of host(CPU) and device(GPU).
- I understand if we want to use a much larger virtual memory than the physical memory, we need use invert page table. Through the invert page table, we can convert the large page number to a smaller actual frame number. Since most memory is not used, we can map virtual memory to the physical memory efficiently.
- I learn that there're many swapping strategy. And each has its own strength and weakness. And in this assignment, we choose LRU strategy.