

Report for CSC3150 assignment 3

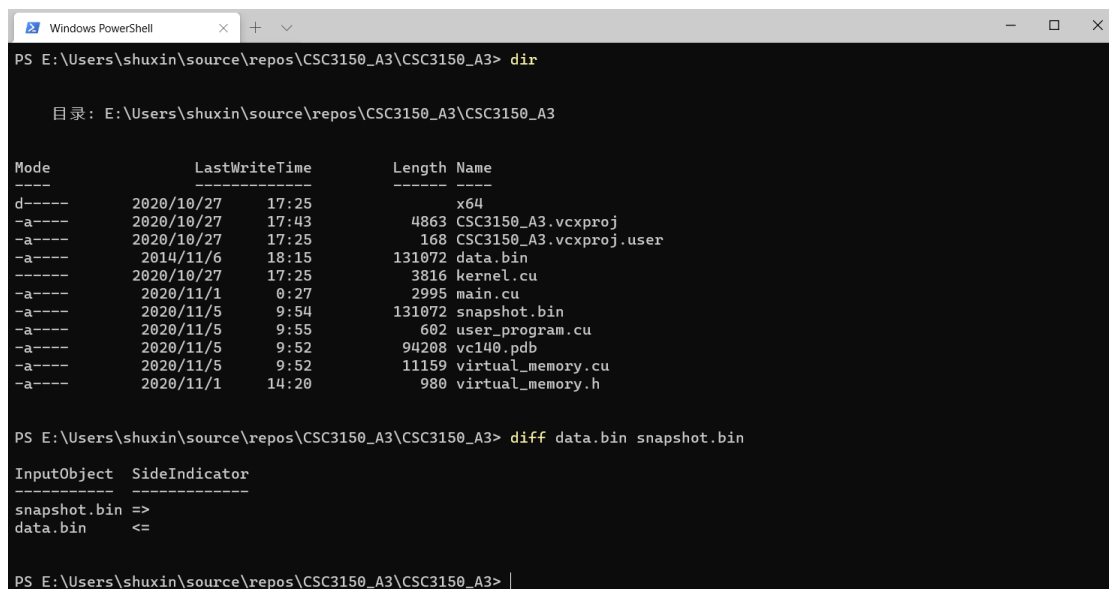
118020362 舒欣

Environment of running my program:

Windows 10, VS2019, CUDA 10.2, GPU is GTX 1650

Execution steps of running my program:

You load my CSC3150_A3 folder into VS2019 (open CSC3150_A3.sln), then press ctrl + F5, it will start to compile and execute, then it will output the page number and you can see snapshot.bin generated under the CSC3150_A3 folder. To see it, you could add snapshot.bin to CSC3150_A3 project and open it to have a look. (just choose add when you right click CSC3150_A3) To compare snapshot.bin with data.bin, you can go to windows terminal and type diff data.bin snapshot.bin under CSC3150_A3 folder. It will show like this:



```
Windows PowerShell
PS E:\Users\shuxin\source\repos\CSC3150_A3\CSC3150_A3> dir

目录: E:\Users\shuxin\source\repos\CSC3150_A3\CSC3150_A3

Mode                LastWriteTime         Length Name
----                -
d-----          2020/10/27          17:25         x64
-a-----          2020/10/27          17:43        4863 CSC3150_A3.vcxproj
-a-----          2020/10/27          17:25        168 CSC3150_A3.vcxproj.user
-a-----          2014/11/6            18:15       131072 data.bin
-a-----          2020/10/27          17:25        3816 kernel.cu
-a-----          2020/11/1             0:27        2995 main.cu
-a-----          2020/11/5             9:54       131072 snapshot.bin
-a-----          2020/11/5             9:55        602 user_program.cu
-a-----          2020/11/5             9:52       94208 vc140.pdb
-a-----          2020/11/5             9:52      11159 virtual_memory.cu
-a-----          2020/11/1            14:20        980 virtual_memory.h

PS E:\Users\shuxin\source\repos\CSC3150_A3\CSC3150_A3> diff data.bin snapshot.bin

InputObject      SideIndicator
-----
snapshot.bin =>
data.bin      <=

PS E:\Users\shuxin\source\repos\CSC3150_A3\CSC3150_A3> |
```

So snapshot.bin is identical to data.bin.

How did I design my program?

Since the number of virtual page (128KB / 32 bytes per page = 2^{12} pages) is

larger than the number of virtual page entries ($32\text{KB} / 32 \text{ bytes per page} = 2^{10}$ pages), I use invert page table to map physical page number to virtual page number.

To perform **LRU**, I create an `__device__ int` variable called **count**. It works like this, when a byte in a page is being read or write, count will increment by 1 and be assigned to this page, so that this page will have the most priority. Therefore, whenever a byte is being read or write, the page it is in will have the most priority comparing to other pages. By the way, I use the first 1024 entries of invert page table to represent valid / invalid bit, and use the second 1024 entries of invert page table to represent the mapped virtual page number, and use the third 1024 entries to represent the **count** of each physical page.

some explanations of some functions:

*init invert page table(Virtual Memory * vm):* initialize valid bit of each physical page to invalid, and mapped page number being -1 (no mapping virtual page), and set **count** to 0.

*vm_read(Virtual Memory * vm, u32 addr):* since *addr* is the virtual address I want to read from *vm* buffer, I can get its offset and virtual page number. Then, I use a bool variable **find** to indicate if a corresponding physical page is found. Next, I can go through the page table and find whether there is a physical page whose mapped virtual page number is the same as the virtual page number of the byte I want to read. once find, set **find** to true and stop immediately. If not find, we need to find the physical page that has the smallest **count**. (which means smallest priority) Afterwards, we should swap out the old page into storage and swap the page I want

to read into the same physical page. Finally, increment this physical page's **count** and read the result from vm buffer. Done!

vm_write(VirtualMemory * vm, u32 addr, uchar value): first, we can get the virtual page number and offset of the byte that we want to write into vm buffer. Then, I use a bool variable **find** to indicate if an empty (invalid) physical page is found. Note, that invalid here means the physical page is unused. The procedure is like this: we first search if the virtual page where the written byte reside is already in physical memory, if it is inside then we just write the byte in physical memory directly and update the **count**; if not find, we need to search is there is empty page that we can use, if there is, then we just update the mapped virtual page number of the empty physical page, then write the byte in and update the **count**; if still not find, we need to find the physical page with smallest **count**, then swap out the old page into storage, and update the mapped virtual page number of this physical page, finally write the byte in and update the **count** of this physical page. Done!

Note that when I swap out the old page into storage, the old page will be placed at the same address where it is in input buffer, since the size of input buffer and storage is the same.

vm_snapshot(VirtualMemory * vm, uchar * results, int offset, int input size): I understand this function as load the content of storage buffer into result buffer through vm buffer. Thus, I use vm_read to read from vm buffer one byte by one byte, and write the byte into result buffer. Done!

What is the page fault number of my program? Explain how it comes.

My page fault number is 8193. In `vm_write`, when write 128 KB data into `vm` buffer, since `vm` buffer only has 32 KB physical memory and 1024 page entries, it will go through $128 / 32 = 4$ times of 1024 page faults, which is $1024 * 4 = 4096$ times of page faults. In `vm_read`, when read from last byte to last 32KB +1 byte, the page fault will only occur once, since physical memory is 32KB. In `vm_snapshot()`, when read 128 KB from start to end, every time a page is read, a page fault will occur, so overall there are $128 \text{ KB} / 32 \text{ B} = 4 \text{ KB} = 4096$ times of page faults. In sum, `vm_write`, `vm_read` and `vm_snapshot` will have $4096 + 1 + 4096 = 8193$ times of page fault.

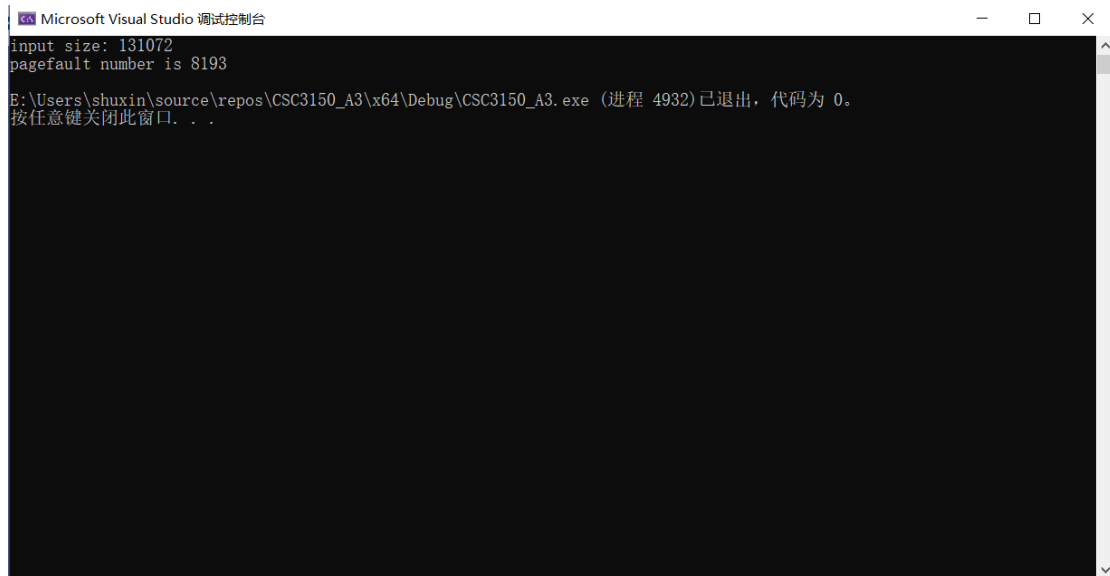
What problems I met in this assignment and what are my solutions?

I do not encounter many problems in the process of designing data structure and writing code, but I do encounter problem when debugging.

Originally, I use `& 31` (0000011111) to capture the last five digits, which is the offset of the virtual address, however, the program shows that offset can only be 0 and 32, so I finally change it to `% 32`, and it works! I still do not know why using bit operator is wrong here.

Another problem I encounter is a problem of the initialization of invert page table. The sample code you give us set the mapped virtual page number to the physical page number, which is absolutely wrong since considering that invert page table is always full after writing, I do not check whether a physical page is valid/invalid. If I do not set initialized virtual page number to -1, then at the first place every physical page is used. It really took me a while to think out that.

Screenshots of my program output:

A screenshot of the Microsoft Visual Studio Debug Console window. The window title is "Microsoft Visual Studio 调试控制台". The output text is as follows:

```
input size: 131072
pagefault number is 8193
E:\Users\shuxin\source\repos\CSC3150_A3\x64\Debug\CSC3150_A3.exe (进程 4932) 已退出, 代码为 0。
按任意键关闭此窗口. . .
```

Note: since I use invert page table, it needs to go through the whole page table when finding a virtual page, my program will take roughly 1 minute to output the page fault number.

What did I learn from this assignment?

I think that to reduce the hardness of debugging, I should be more careful when writing code!

I learned the basic process of memory management. Invert page table is very convenient to use both single thread and multithread, however, its speed is very slow.

I learned CUDA programming, how to use GPU to simulate physical memory and secondary memory.

I also learned LRU algorithm, clearly know the process of swapping page in and out and how can a page fault occurs.

