CSC3150 Assignment3 Report

Ziqi Gao 高梓骐

118010077

1. Introduction

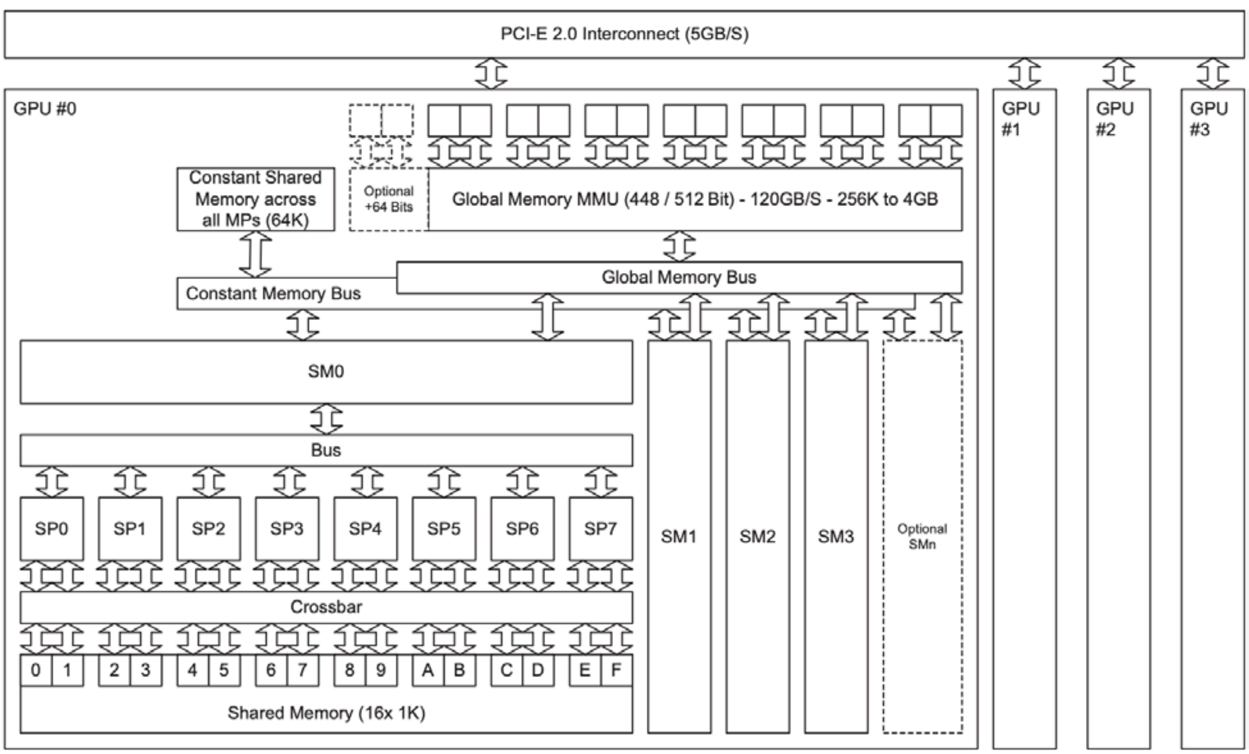
Versions of OS: Windows 10 Pro - 1909

Compilation Toolchains:

C++ Compiler: MSVC-14.27.29110 x64 architecture.

CUDA Compiler: NVCC 11

In this task, I will simulate a single process virtual memory system with CUDA. We will use one GPU core to execute the simulation. The global memory of the core is larger and slower， so it can be considered as the secondary memory in the virtual memory system. Moreover, the shared memory that is smaller and faster will be considered as our main memory.



In this program, we will simulate the process of writing data to the memory and the process of reading data from memory. We will make a page table to map the logical address and the physical address. Whenever we try to get access to a location not in our main memory, we will raise the page fault and count the number of the page fault number happens during the process. The result of the reading process will be written into a file called *snapshot.bin.*

2. Basic Design

This program consists of two parts. The first part is executed in our CPU, RAM, and disk. Another CUDA part is executed in the GPU. Those functions and variables that will be executed in GPU will have name identifiers like __global__, or __device__.

In the first part, we will only declare some variables and one main function that will pass those arguments to the GPU. After the GPU finishes the simulation, CPU reads the data from GPU's buffer and writes those data to the *snapshot.bin.* Those functions and variables are defined in *main.cu.*

Also, there are some functions and variables declared to be used in the GPU in *main.cu.* mykernel function will start the simulation by calling the use_program function in *user_program.cu*.

This user_program function will load data from the data buffer to the simulated main memory. Then it will look through the entire pages in the memory. Finally, it will call vm_snapshot function to write all the data got from the input buffer in the main memory and the secondary memory to the data buffer.

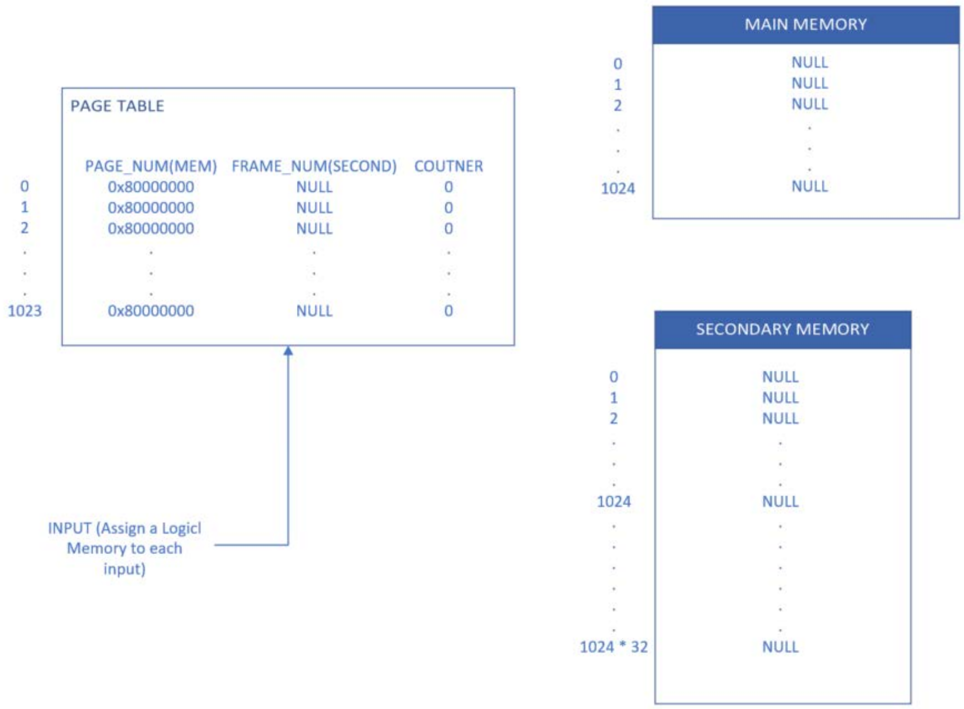
As to the virtual memory system, here are some figures showing how it works: The simulation process can be divided into the following parts:

a. Insert the first 32KB data:

At the beginning, the page table's first column (index: 0-1023) is set to 0x80000000 indicating that 0-1023's memory space is empty. When data comes in, it will raise a page fault since there is no corresponding physical memory for the data's logical memory (the logical memory address is equal to the order of the input). In practical, we will keep a free-frame list, but in this simulation, I do not work in this way. The page table works like a hash table, whenever the logical address comes in, it will first check the location calculated by the following hash function:

$$logical\ address\ mod\ 1024\ (Legnth\ of\ the\ Table)$$

If the memory's number is 0x80000000, it raises the page fault, and the operating system will conduct the LRU (least recently used) algorithm to allocate a new physical memory number to it. In this stage, all the allocated new physical memory number should be equal to the remainder, i.e., the result of the hash function. Also, the frame number in the secondary memory is also allocated according to the page number of the logical memory.



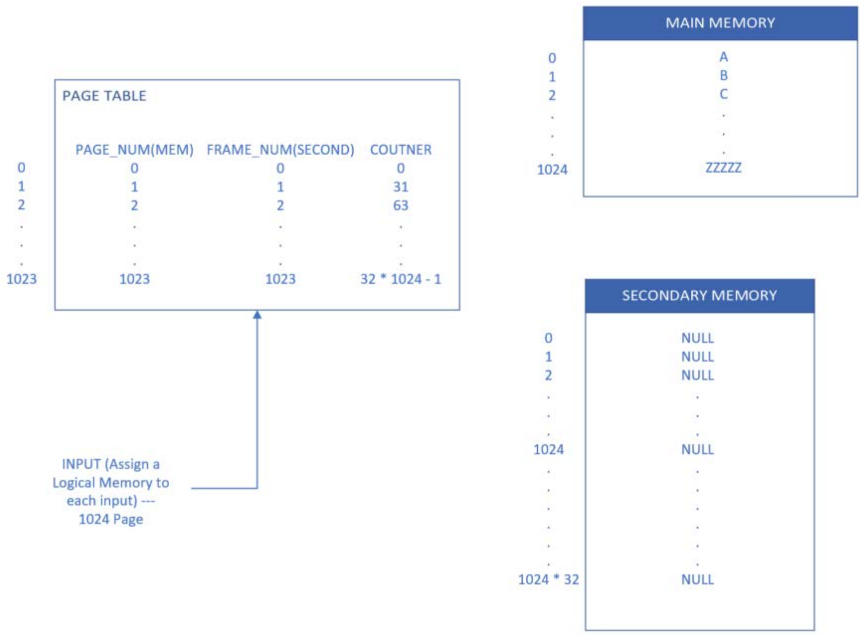
b. Insert the 32KB – 128KB Data:

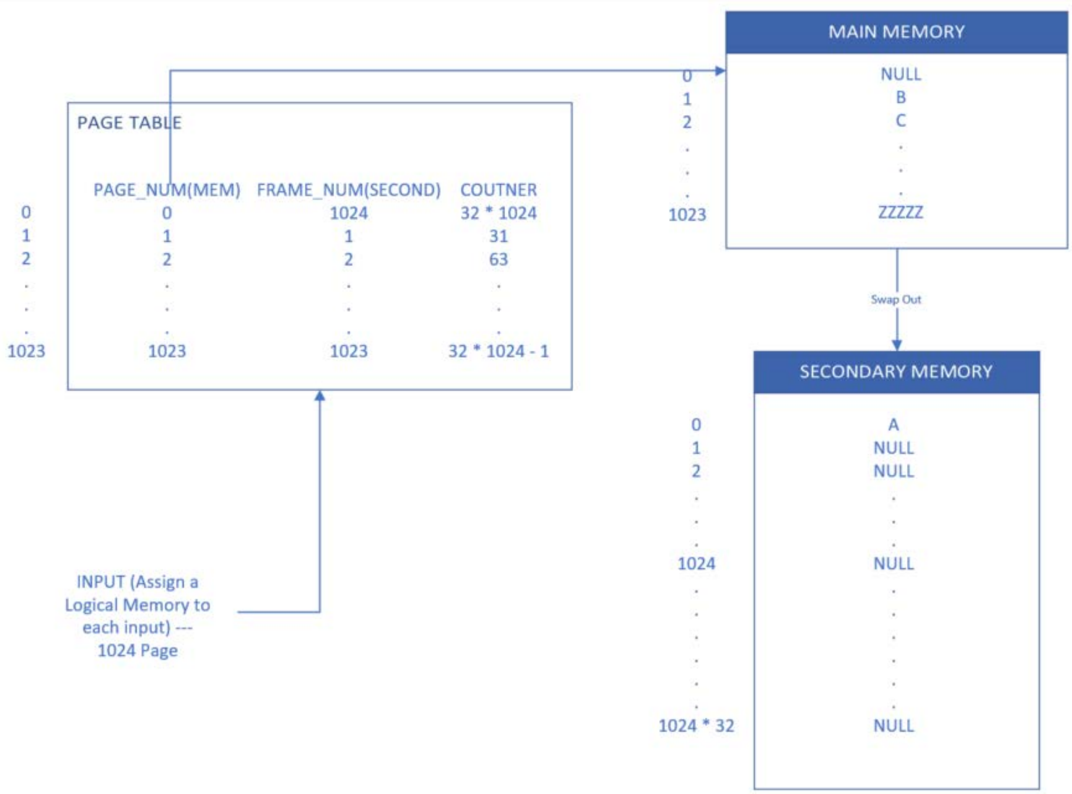
When we are trying to insert more data into the already full memory, we will need to use another strategy. For example, when we are inserting the first byte of the 1024 page. First of all, the hash table will map the logical address to the 0^{th} row of the page table. And it will find that the FRAME_NUM is not equal to the page number of that input value. Then we will conduct the LRU algorithm and find that the 0^{th} row has the smallest counter value. Then we will do the following steps:

i. Raise the page fault.

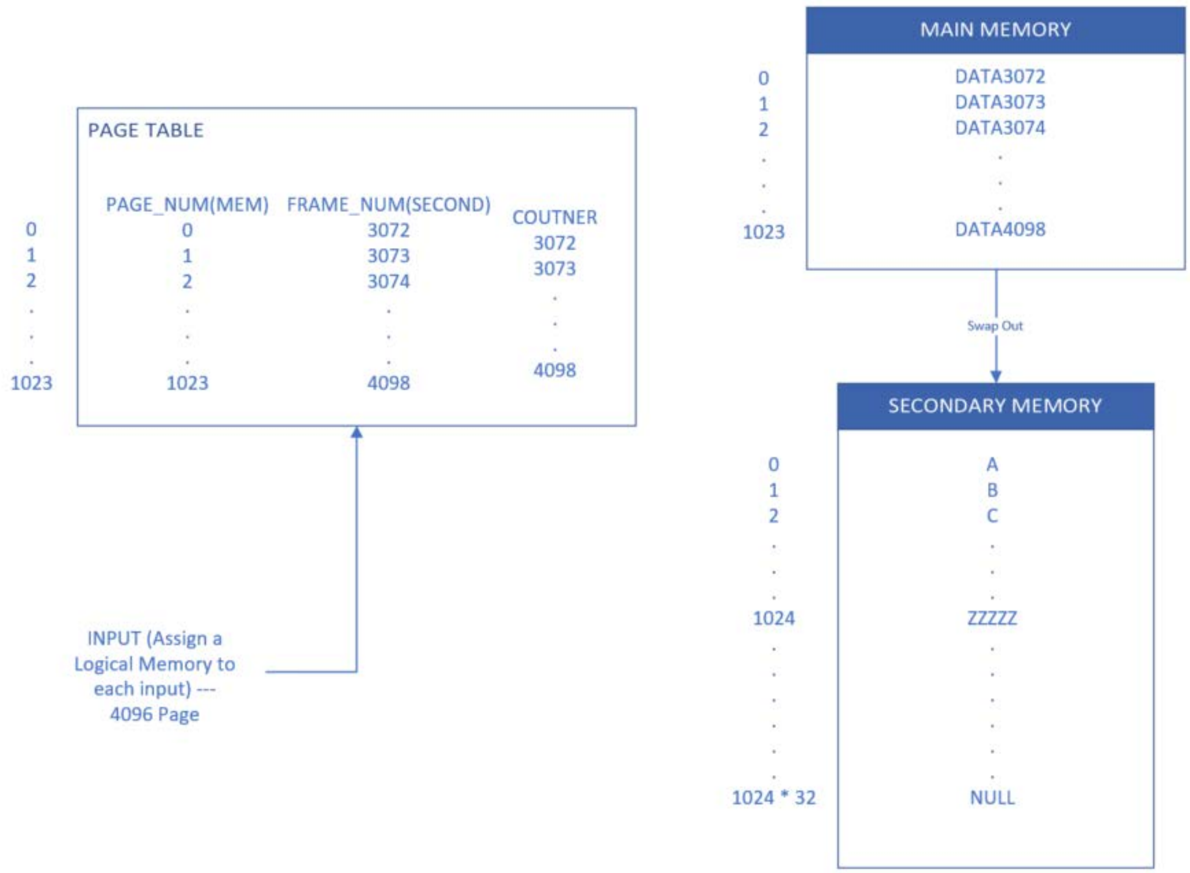
ii. Swap out the data in the main memory and save it to the 0^{th} row of the secondary memory according to the FRAME_NUM in the page table.

iii. Update the page table with the three values to be 0, 1024, 1024 * 32.


PAGE TABLE
PAGE_NUM(MEM) FRAME_NUM(SECOND) COUTNER
0 0 0 0
1 1 1 31
2 2 2 63
1023 1023 1023 32 * 1024 - 1
INPUT (Assign a Logical Memory to each input) --- 1024 Page
MAIN MEMORY
0 A
1 B
2 C
1024 ZZZZZ
SECONDARY MEMORY
0 NULL
1 NULL
2 NULL
1024 NULL
1024 * 32 NULL

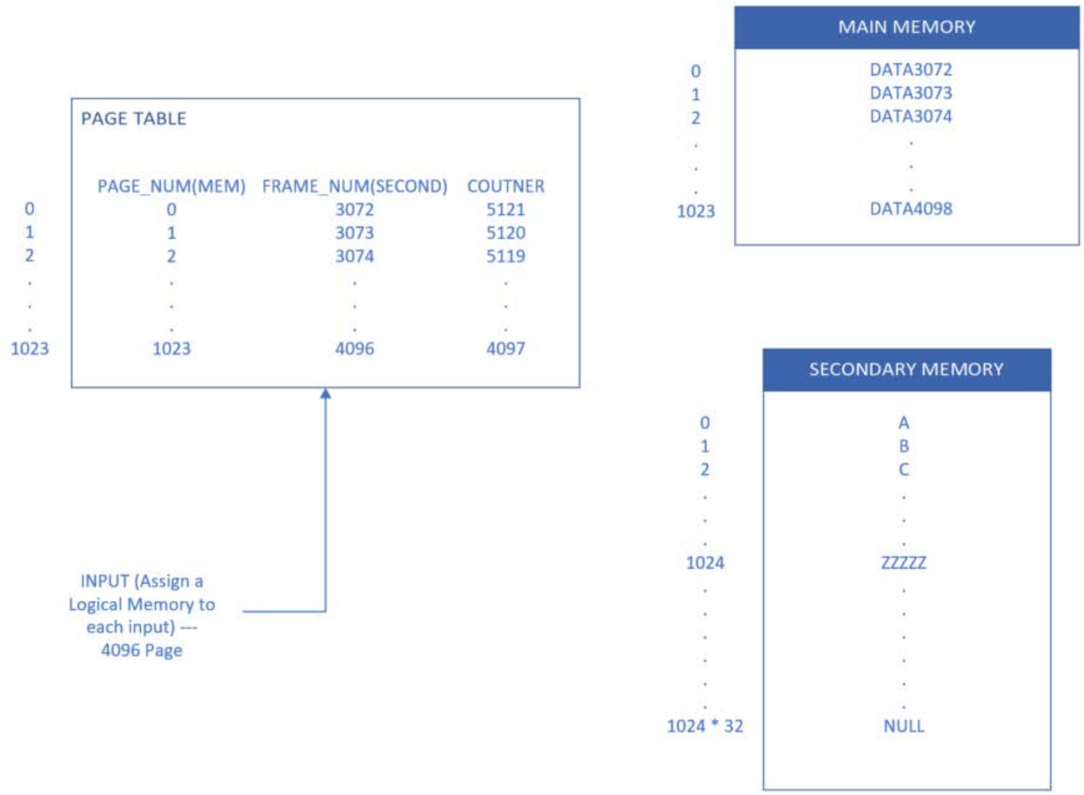

PAGE TABLE
PAGE_NUM(MEM) FRAME_NUM(SECOND) COUTNER
0 0 1024 32 * 1024
1 1 1 31
2 2 2 63
1023 1023 1023 32 * 1024 - 1
INPUT (Assign a Logical Memory to each input) --- 1024 Page
MAIN MEMORY
0 NULL
1 B
2 C
1023 ZZZZZ
Swap Out
SECONDARY MEMORY
0 A
1 NULL
2 NULL
1024 NULL
1024 * 32 NULL

After all the input are finished, those three tables look like this:

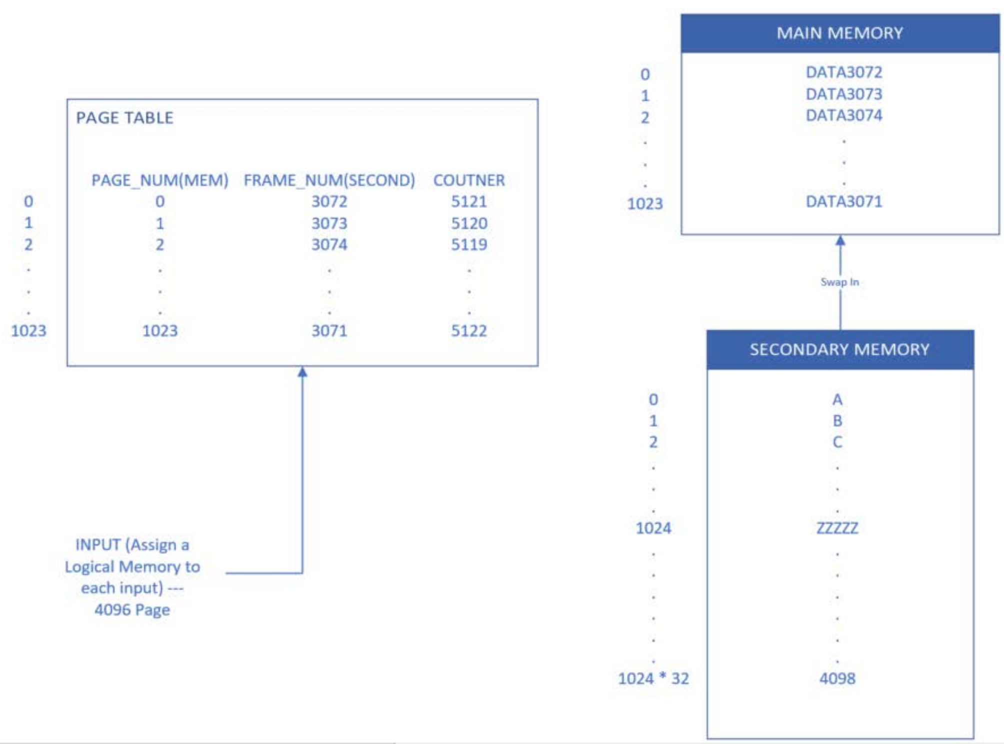


c. Read 32769 Times in the Reverse Order:

In this step, we will read the data in reverse order. 32769 = 1024 Page + 1 Byte. As a result, this step will go through all of the data in the main memory and one extra bit from the secondary memory by swapping in. The memory after the first 32768 times read will become:



The 32769 times read will swap the data in row 1023 out (DATA4098) and swap in DATA 3071 to the memory.



d. Read from the 0 page (Snapshot):

This step will use vm_read function for many times to read data from the memory and the secondary memory. The process is almost the same as the c part.

3. LRU Algorithm and Data Structure:

The structure of my page table is a one-dimension array. However, it has 4 * 1024 = 4096 rows = 16KB. I consider the 0-1023 row as the PAGE_NUM, 1024-2047 row as the FRAME_NUM, 2048-3071 rows as the COUNTER, and 3072-4095 rows as the modify bit.

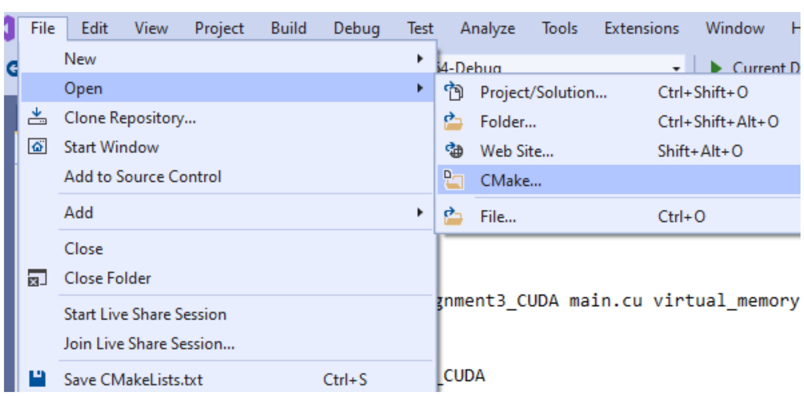
I conduct the LRU algorithm by adding a counter to the page table. Whenever I update one row in the page table, memory, or secondary memory, the corresponding row's counter will plus 1. The row with its least value of the counter will be replaced if we need to execute the swapping.

4. Execution

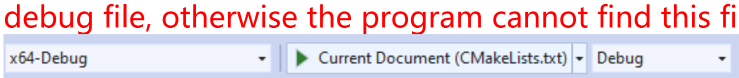
I did not develop my project in VS because I am not familiar with its complier configuration, and I met MSB3721 fault and I cannot figure it out. Therefore, I will also upload my CLion project to you, if my VS project cannot be built, please use it. Please use CLion or other IDE to open the cmake file to build my project, thanks!

For Visual Studio Users:

File >> Open >> CMake >> Open my CMakeLists.txt



After you import the cmake file, please wait for a moment and click the Current Document (CMakeLists.txt) to run my code. (Please copy the data.bin to the debug file, otherwise the program cannot find this file.)

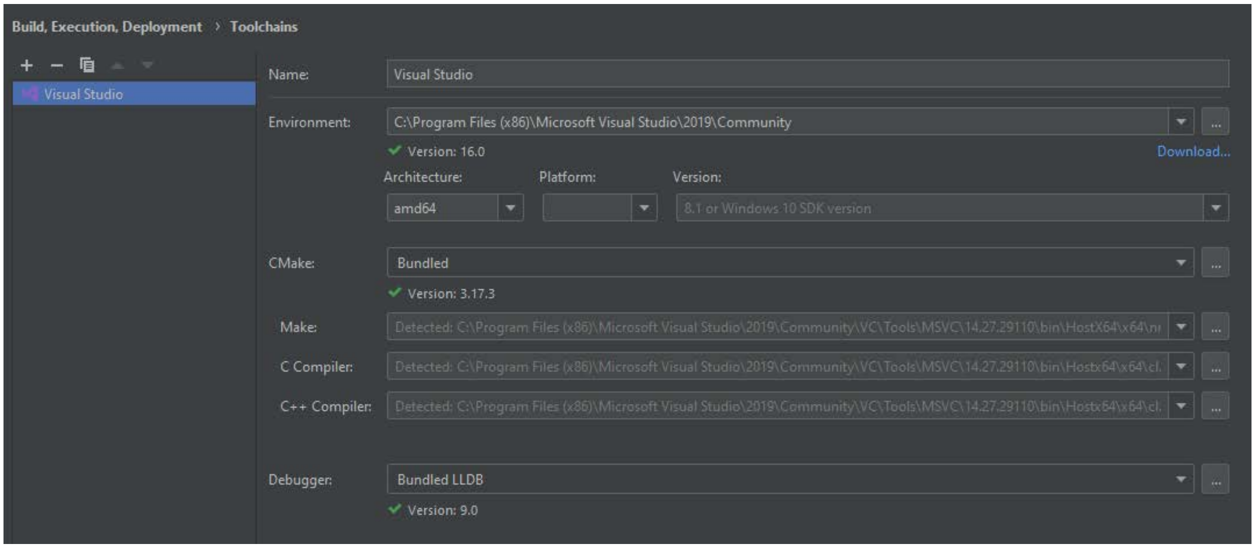


CLion Users:

Click open a new project and choose my CMakeLIsts.txt to load my project.

After you import the cmake file, please wait for a moment run my file.

Since I am using CLion as my IDE to code this program, I have a CMAKE_FILE, and the setting of the CLion Toolchains is as follows:

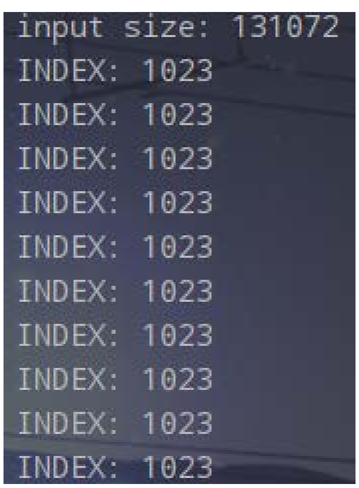


What I am doing during my development is to save the data.bin with my source code, and run them in IDE. Then I can check my snapshot.bin in the inner cmake-build-debug directory.

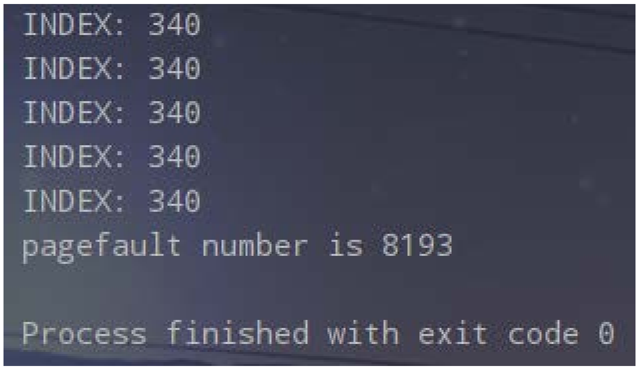
Also, I changed the virtual_memory.h to virtual_memory.cuh to link my code. I come across link error if I use virtual_memory.h.

The output of my program is as follows:

At the beginning, it will output the input size of the file. Whenever it read a byte, it will output the byte's index in the memory.



In the end, it prints out the total page fault number.



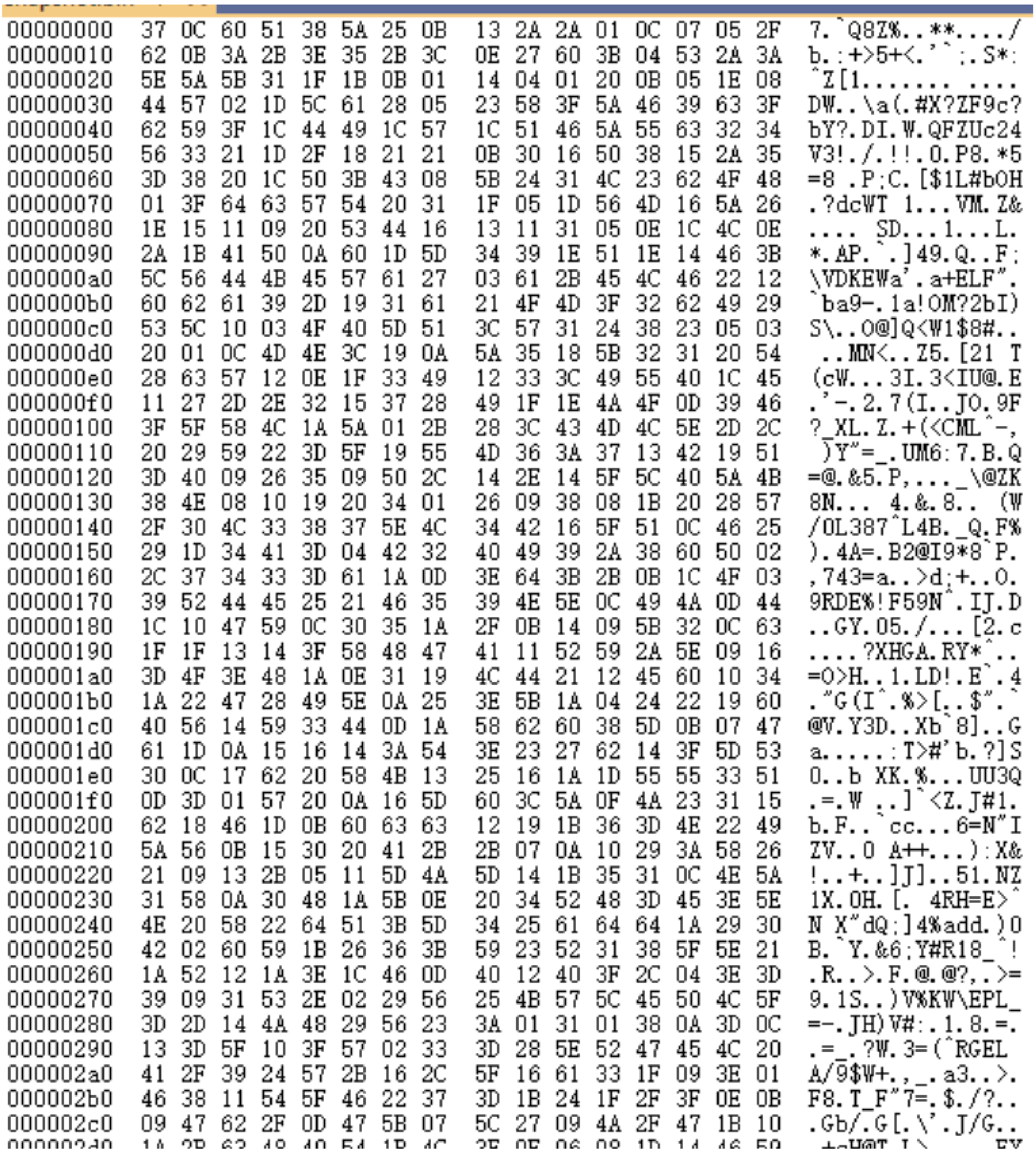
8193 = 4196 + 1 + 4196, the first 4196 comes from the write part. Whenever we write a page to the memory, it will raise a page fault.

The 1 comes from the 32769 times read. Since 32769 = 1024 pages + 1 byte. The last 1 byte's page number is not in the memory so it will raise the page fault.

The second 4196 comes from the snapshot part. Since we read it from the first byte to the last byte, we do not have any needed content in the beginning. As a result, every page read from the memory will raise a page fault.

My program will execute for about 20 seconds, which is a not bad speed.

And the snapshot.bin looks like that:



5. What I Learned

I learned lots of knowledge about the Virtual Memory Schema for one process memory management through this project. It is really hard to work out how this schema works because it is abstract and tedious. I need to draw lots of figures to help me understand the process. Also, the writing process is much different from the content introduced in the textbook that only covers the read part.

There is still something I need to get improved. For example, I did not use a common data structure like linked list or stack. I use a one-dimension array to represent my page table. I think the speed is fine for this data structure, but I think stack and linked list may be more space efficient because those two structures' own properties can show the order of the data without the counter.