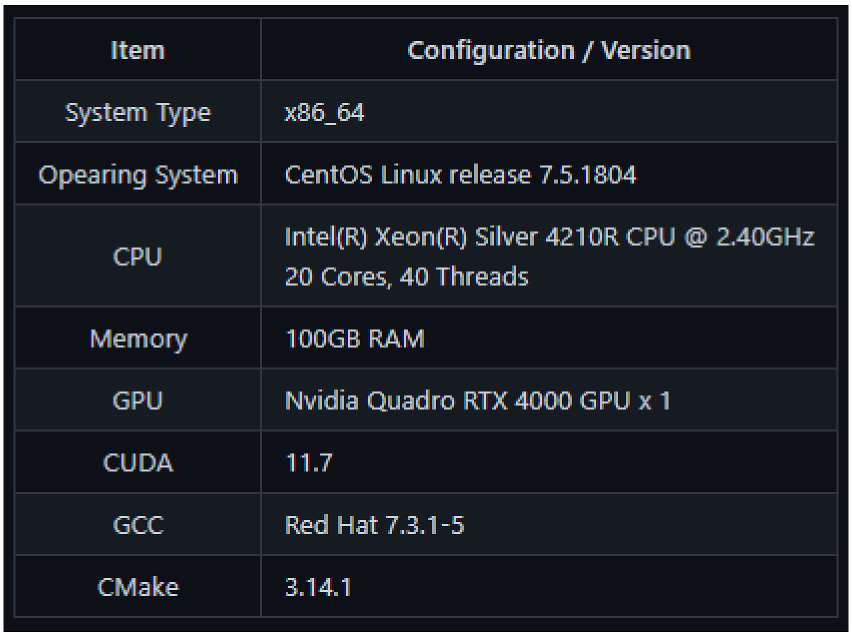
CSC3150_Assignment3_Report 120090263

1.Environment of running my program

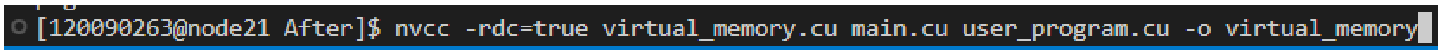
```
OS: CentOS Linux release 7.5.1804

VSC version: 1.73.0

CUDA version: 11.7

GPU information: Nvidia Quadro RTX 4000 GPU
```



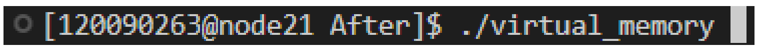
Item	Configuration / Version
System Type	x86_64
Opearing System	CentOS Linux release 7.5.1804
CPU	Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz 20 Cores, 40 Threads
Memory	100GB RAM
GPU	Nvidia Quadro RTX 4000 GPU x 1
CUDA	11.7
GCC	Red Hat 7.3.1-5
CMake	3.14.1

2.Execution step (use VSCode)

```
1. enter "nvcc -rdc=true virtual_memory.cu main.cu user_program.cu -o virtual_memory"
2. enter "./virtual_memory"
```



```
[120090263@node21 After]$ nvcc -rdc=true virtual_memory.cu main.cu user_program.cu -o virtual_memory
```



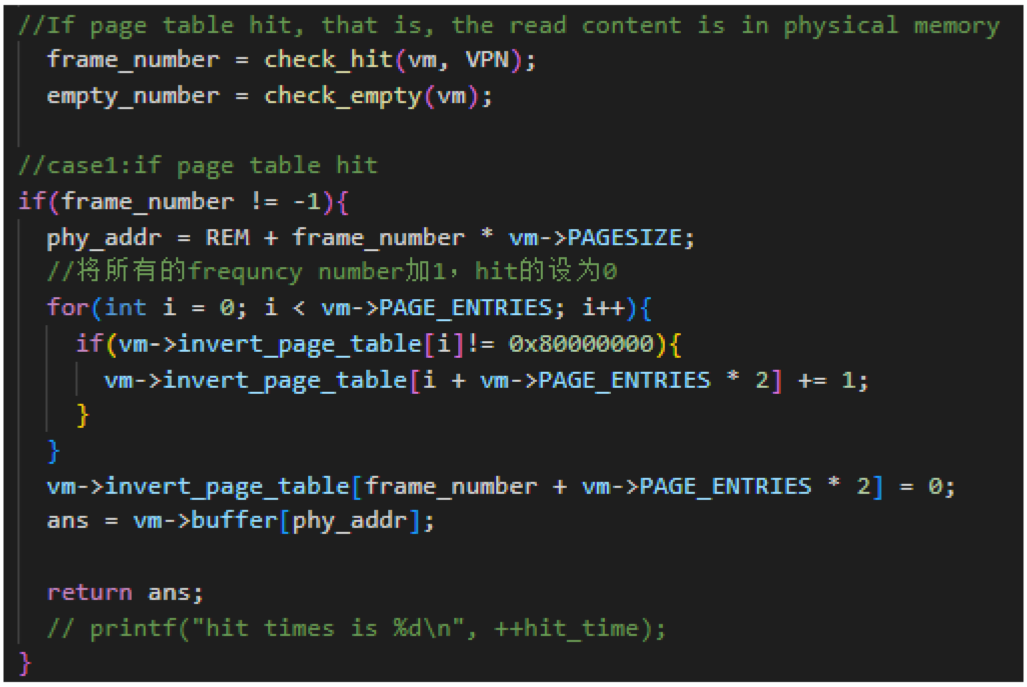
```
[120090263@node21 After]$ ./virtual_memory
```

3.Design

In my program, I generally seperate the situation into two cases.

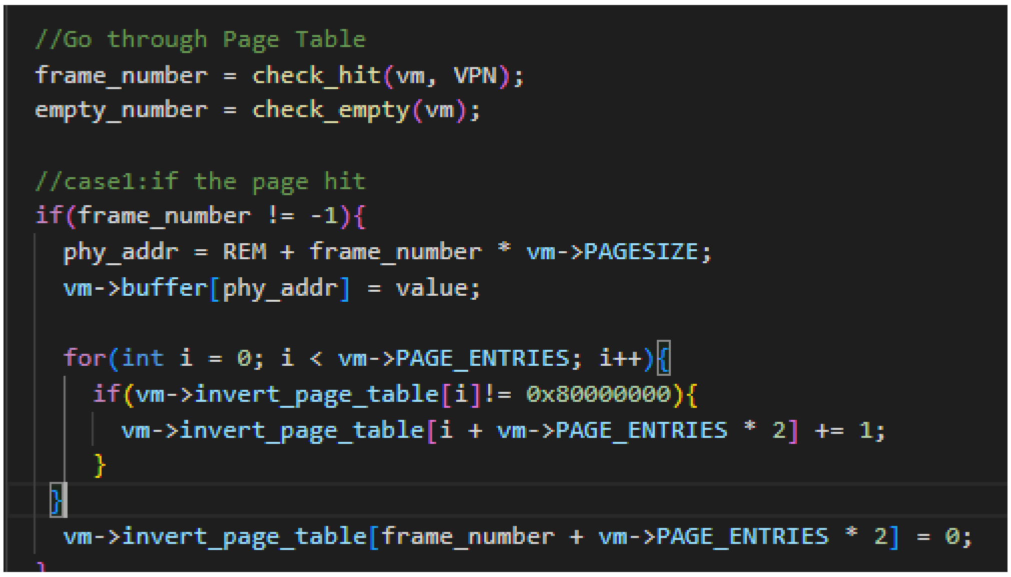
The first case is page table is hit in read/write process. I write check_hit() funtion to go through paga table, if the valid bit is valid and the virtual page number is matched, then the page is hit and return the frame number. In this case, if table is hit, I get the frame number and calculate *physical address*. Increase all sequency number by one and set the hit page to be 0. Then directly read/write the value.

read case1



```
//If page table hit, that is, the read content is in physical memory
  frame_number = check_hit(vm, VPN);
  empty_number = check_empty(vm);

//case1:if page table hit
if(frame_number != -1){
  phy_addr = REM + frame_number * vm->PAGESIZE;
  //将所有的frequncy number加1，hit的设为0
  for(int i = 0; i < vm->PAGE_ENTRIES; i++){
    if(vm->invert_page_table[i]!= 0x80000000){
      vm->invert_page_table[i + vm->PAGE_ENTRIES * 2] += 1;
    }
  }
  vm->invert_page_table[frame_number + vm->PAGE_ENTRIES * 2] = 0;
  ans = vm->buffer[phy_addr];

  return ans;
  // printf("hit times is %d\n", ++hit_time);
}
```

write case1



```
  //Go through Page Table
  frame_number = check_hit(vm, VPN);
  empty_number = check_empty(vm);

  //case1:if the page hit
  if(frame_number != -1){
    phy_addr = REM + frame_number * vm->PAGESIZE;
    vm->buffer[phy_addr] = value;

    for(int i = 0; i < vm->PAGE_ENTRIES; i++){
      if(vm->invert_page_table[i]!= 0x80000000){
        vm->invert_page_table[i + vm->PAGE_ENTRIES * 2] += 1;
      }
    }
    vm->invert_page_table[frame_number + vm->PAGE_ENTRIES * 2] = 0;
```

The second caseis that page table is not hit. Further I seperate it into two inner cases.

The first inner case is that the page table is not full (Although during read process this may not happen). In write function, simply update the page table, change frequency number and write the value into physical memory (virtual memeory buffer).

The second inner case is that the page table is full. I use LRU to decide the entry to be swapped out of page table. That is, traverse the frequency number to get the largest one and its corresponding frame number. First swap this page into disk and upadte the swap table. Then swap the requiring data from storage by locating the data thorugh previously recorded swap table information.

3.1Details

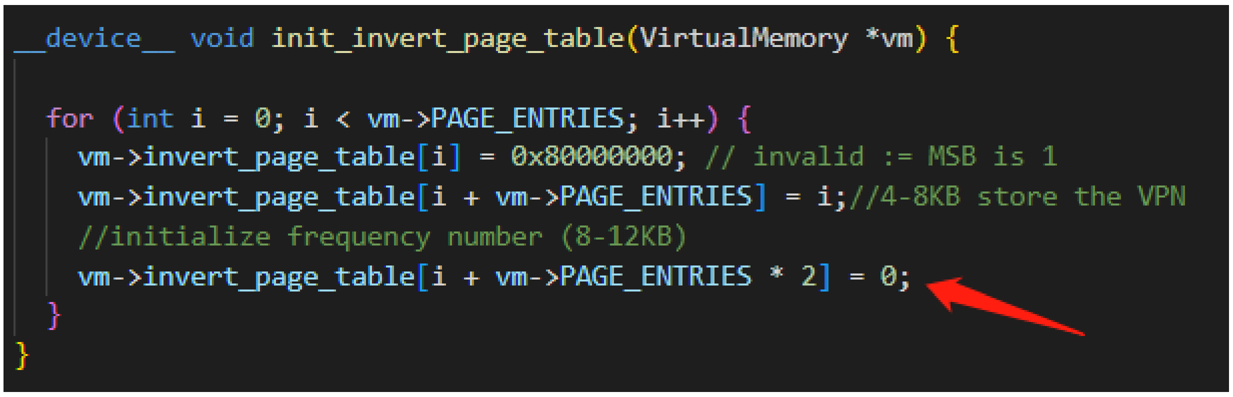
3.1.1. Physical address calculation

To map virtual address to physical address, first obtain the frame number. I seperate the given virtual address by dividing page size to get virtual page number and remaining byte. Then use the inverted page table to located the VPN (If hit) and get the frame number. Then, frame number times page size and plus remaining byte to get the physical address.

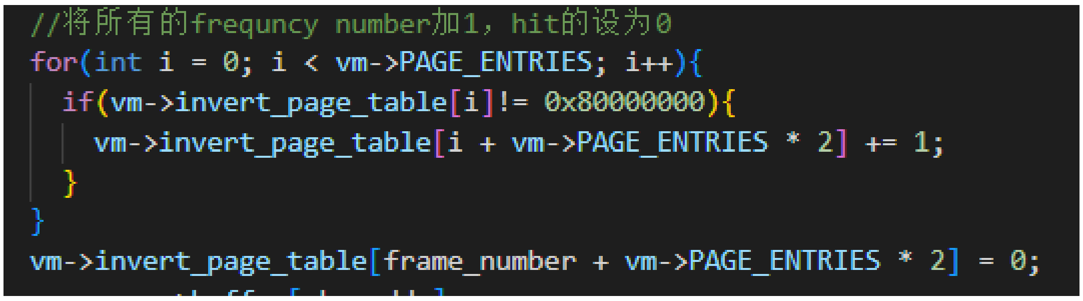
```
u32 VPN = addr / vm->PAGESIZE;
u32 REM = addr % vm->PAGESIZE; //the remaining data in last page
```

3.1.2. LRU implementation

In this program, I implement the LRU using sequency number, that is, every entry in page table has an sequency number, as the process goes on, if the page is hit, the number will be set to 0, otherwise it increases. During the swap process, the one that has the largest frequency number should be swapped out. The frequency number is stored in page setting area.

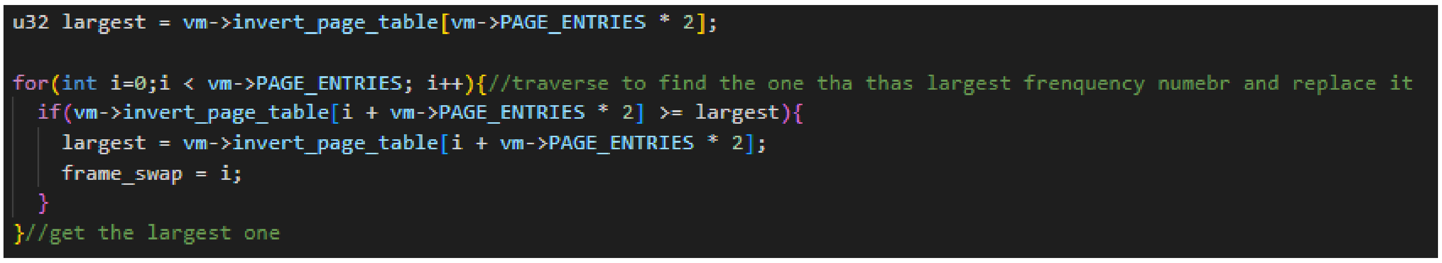


```
__device__ void init_invert_page_table(VirtualMemory *vm) {

  for (int i = 0; i < vm->PAGE_ENTRIES; i++) {
    vm->invert_page_table[i] = 0x80000000; // invalid := MSB is 1
    vm->invert_page_table[i + vm->PAGE_ENTRIES] = i;//4-8KB store the VPN
    //initialize frequency number (8-12KB)
    vm->invert_page_table[i + vm->PAGE_ENTRIES * 2] = 0;
  }
}
```



```
 //将所有的frequncy number加1, hit的设为0
 for(int i = 0; i < vm->PAGE_ENTRIES; i++){
   if(vm->invert_page_table[i]!= 0x80000000){
     vm->invert_page_table[i + vm->PAGE_ENTRIES * 2] += 1;
   }
 }
 vm->invert_page_table[frame_number + vm->PAGE_ENTRIES * 2] = 0;
```

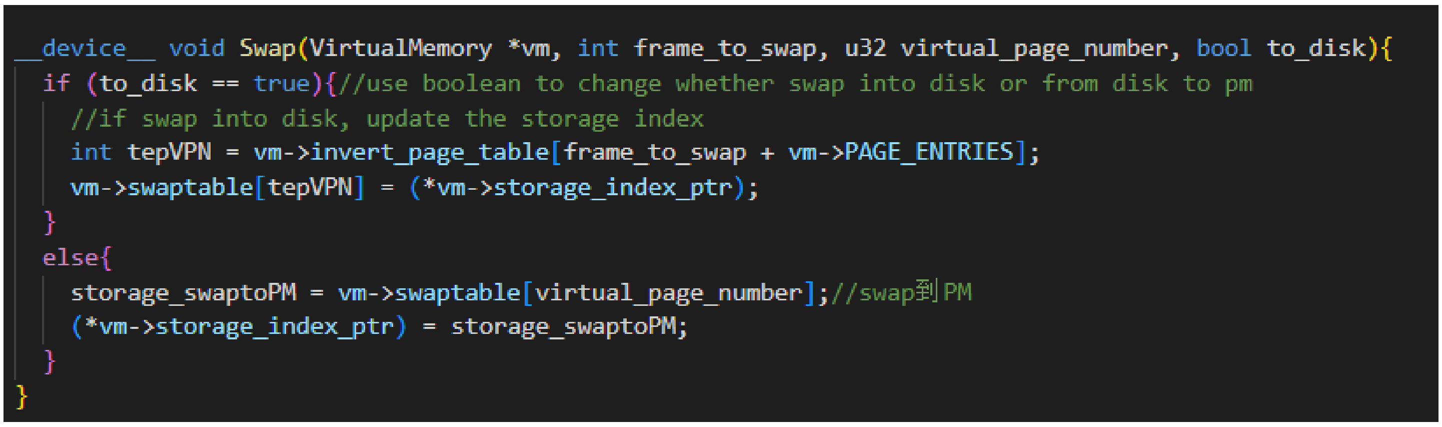
find the largest frequency number



```
u32 largest = vm->invert_page_table[vm->PAGE_ENTRIES * 2];

for(int i=0;i < vm->PAGE_ENTRIES; i++){//traverse to find the one tha thas largest frenquency numebr and replace it
  if(vm->invert_page_table[i + vm->PAGE_ENTRIES * 2] >= largest){
    largest = vm->invert_page_table[i + vm->PAGE_ENTRIES * 2];
    frame_swap = i;
  }
}//get the largest one
```

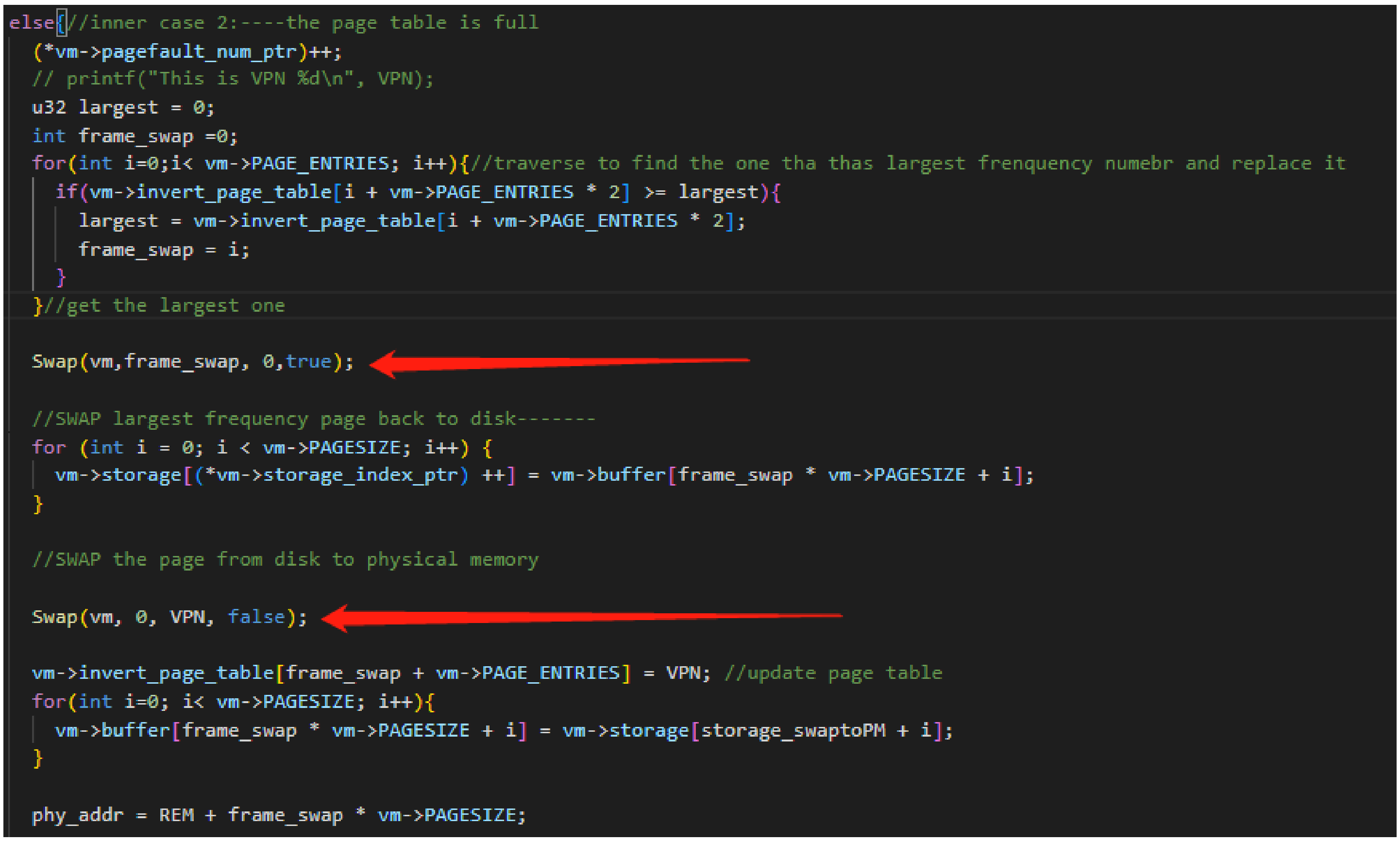
3.1.3. Swap Table implementation

Additionally, a swap table is designed. The virtual page number is used as index and the element is the address of swapped frame in disk. Everytime when one page is swapped out of disk, the empty pointer will point to the location. A boolean variable is used to deal with 2 situations that whether it is swap into disk or swap from disk to physical memeory.



```
__device__ void Swap(VirtualMemory *vm, int frame_to_swap, u32 virtual_page_number, bool to_disk){
  if (to_disk == true){//use boolean to change whether swap into disk or from disk to pm
    //if swap into disk, update the storage index
    int tepVPN = vm->invert_page_table[frame_to_swap + vm->PAGE_ENTRIES];
    vm->swaptable[tepVPN] = (*vm->storage_index_ptr);
  }
  else{
    storage_swaptoPM = vm->swaptable[virtual_page_number];//swap到PM
    (*vm->storage_index_ptr) = storage_swaptoPM;
  }
}
```

swap process example in program



```
else{//inner case 2:----the page table is full
  (*vm->pagefault_num_ptr)++;
  // printf("This is VPN %d\n", VPN);
  u32 largest = 0;
  int frame_swap =0;
  for(int i=0;i< vm->PAGE_ENTRIES; i++){//traverse to find the one tha thas largest frenquency numebr and replace it
    if(vm->invert_page_table[i + vm->PAGE_ENTRIES * 2] >= largest){
      largest = vm->invert_page_table[i + vm->PAGE_ENTRIES * 2];
      frame_swap = i;
    }
  }//get the largest one

  Swap(vm,frame_swap, 0,true);

  //SWAP largest frequency page back to disk-------
  for (int i = 0; i < vm->PAGESIZE; i++) {
    vm->storage[(*vm->storage_index_ptr) ++] = vm->buffer[frame_swap * vm->PAGESIZE + i];
  }

  //SWAP the page from disk to physical memory

  Swap(vm, 0, VPN, false);

  vm->invert_page_table[frame_swap + vm->PAGE_ENTRIES] = VPN; //update page table
  for(int i=0; i< vm->PAGESIZE; i++){
    vm->buffer[frame_swap * vm->PAGESIZE + i] = vm->storage[storage_swaptoPM + i];
  }

  phy_addr = REM + frame_swap * vm->PAGESIZE;
```

4.Page fault number

For test case1, my program's page fault number is 8193. First, during write processs, since there is nothing in physical memory, the page fault number is 4096. Then during read process, since it reading from tail to head, the pages in physical memory will hit but as the data has one more byte there will be one page fault. Last, the snapshot will again has 4096 page fault. Totally there will be 8193 pages.

For test case2, my program's page fault number is 9215. The first and second write process will have 4096 and 1023 page faults due to their data size respectivesly. Then the snapshot will have 4096 page faults. Totally there are 9215 page faults.

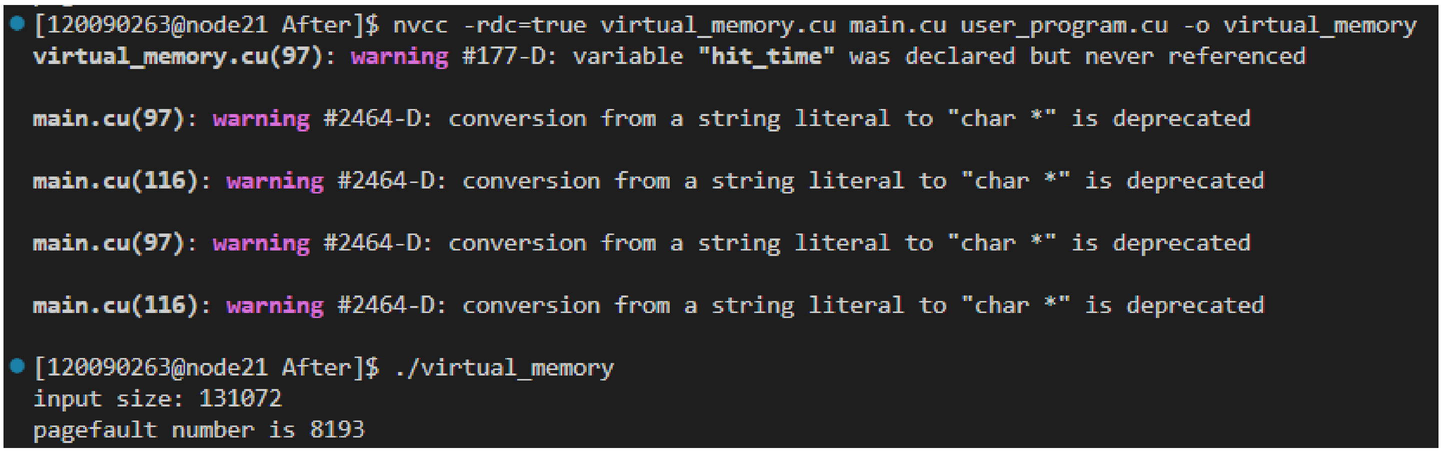
5. My problem

Problem: When designing the swap process, I encounter the problem that since I keep writing to disk regardless of those pages swapped back to physical memory, the disk will exceeds the given 128KB (131072 bytes).

Solution: Each time one page is swapped back to physical memory, I change the disk pointer to point at this location to indicate that this is empty. And next time when one page is swapped into disk, it can be stored in this location. Therefore, it wont exceed the given storage space.

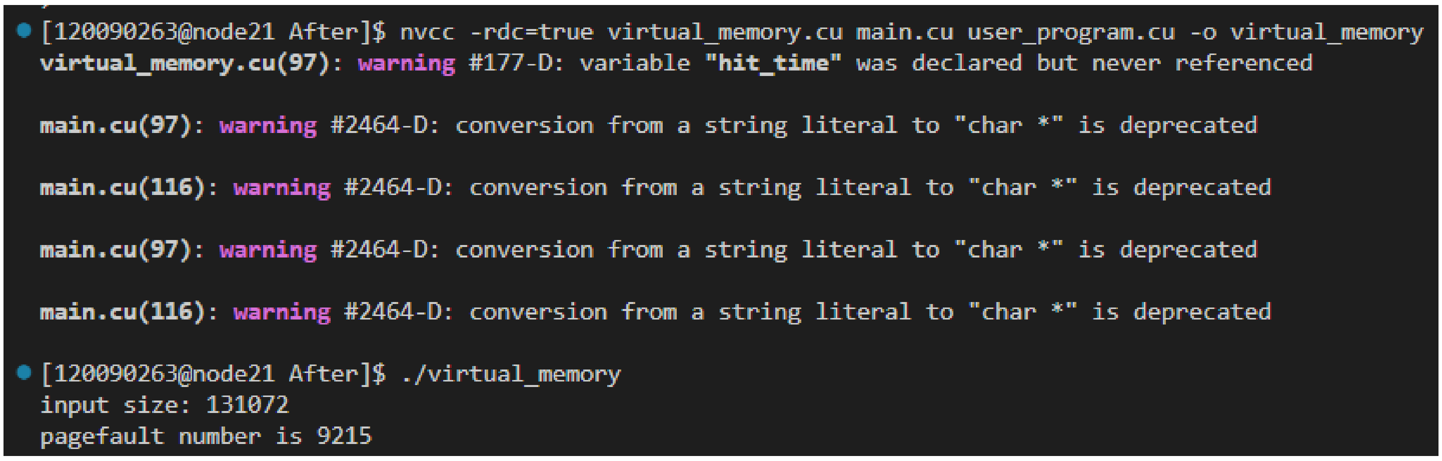
6. Screen shot

test1



```
[120090263@node21 After]$ nvcc -rdc=true virtual_memory.cu main.cu user_program.cu -o virtual_memory
virtual_memory.cu(97): warning #177-D: variable "hit_time" was declared but never referenced

main.cu(97): warning #2464-D: conversion from a string literal to "char *" is deprecated

main.cu(116): warning #2464-D: conversion from a string literal to "char *" is deprecated

main.cu(97): warning #2464-D: conversion from a string literal to "char *" is deprecated

main.cu(116): warning #2464-D: conversion from a string literal to "char *" is deprecated

[120090263@node21 After]$ ./virtual_memory
input size: 131072
pagefault number is 8193
```

test2



```
[120090263@node21 After]$ nvcc -rdc=true virtual_memory.cu main.cu user_program.cu -o virtual_memory
virtual_memory.cu(97): warning #177-D: variable "hit_time" was declared but never referenced

main.cu(97): warning #2464-D: conversion from a string literal to "char *" is deprecated

main.cu(116): warning #2464-D: conversion from a string literal to "char *" is deprecated

main.cu(97): warning #2464-D: conversion from a string literal to "char *" is deprecated

main.cu(116): warning #2464-D: conversion from a string literal to "char *" is deprecated

[120090263@node21 After]$ ./virtual_memory
input size: 131072
pagefault number is 9215
```

7. What I learned from this project

In this project, I implement the virtual memory by using CUDA system. I get a better understanding of the relationship between virtual memory, physical memory and disk. I also understand how page table is implemented. Besides, I get to know the use of CUDA, including its structure, grammar and relation with C language.

*ps: Bonus task is not accomplished in this program