



THE CHINESE UNIVERSITY OF HONG KONG, SHENZHEN

CSC 3150

OPERATING SYSTEMS

---

# Assignment 3

---

*Author:*  
Shi Wenlan

*Student Number:*  
119010265

November 9, 2021

## Contents

|          |   |          |
|----------|---|----------|
| <b>1</b> | <b>Task 1</b>                                   | <b>2</b> |
| 1.1      | The environment of running my program . . . . . | 2        |
| 1.2      | The steps to execute my program . . . . .       | 2        |
| 1.3      | How did I design the program . . . . .          | 2        |
| 1.4      | Page fault number: . . . . .                    | 3        |
| 1.5      | Problem I met: . . . . .                        | 3        |
| 1.6      | Screenshots of my program output . . . . .      | 4        |
| 1.7      | What did I learn from the task . . . . .        | 4        |
| <b>2</b> | <b>Bonus</b>                                    | <b>4</b> |
| 2.1      | The environment of running my program . . . . . | 4        |
| 2.2      | The steps to execute my program . . . . .       | 4        |
| 2.3      | How did I design the program . . . . .          | 4        |
| 2.4      | Page fault number: . . . . .                    | 5        |
| 2.5      | Problem I met: . . . . .                        | 5        |
| 2.6      | Screenshots of my program output . . . . .      | 5        |
| 2.7      | What did I learn from the task . . . . .        | 5        |

## 1 Task 1

### 1.1 The environment of running my program

**Windows environment (develop environment):** OS: Windows 10, VS version: 2019, CUDA version: 11.1, GPU: NVIDIA GeForce GTX 1650

**Linux environment (TC301):** nvcc: release 10.1, V10.1.105, `uname -a`: Linux 3.10.0-957.21.3.el7.x86\_64

### 1.2 The steps to execute my program

**Windows (using Visual Studio):** Add existing items (the source codes) into your cuda project. Click to compile each .cu file. Then run the program. (I have attached the screen recording that I ran it on my computer. )

**Linux (using .sh script):** Get into the source code folder in the terminal. Then enter `/bin/shscript.sh` to run the script.

### 1.3 How did I design the program

**How to construct page table:** I used the first 1024 cells of the *pt* to store valid bits, 1 means invalid, and 0 means valid. The next 1024 cells are used to store the first 27 bits of the virtual address (the remaining 5 bits are offset).

**How to implement LRU:** LRU requires finding the least recently used page table entry and swapping its corresponding physical page to disk. The data structure I used is the min heap, so that the least recently used entry can be found in constant time, and the min heap can be maintained in  $O(\lg n)$  time only. My method of implementing min heap is to build a structure called *node* and store its pointers in two arrays, *nodes* and *ALU*. I have a one-to-one correspondence between each *node* and each physical address, and store its corresponding physical address, the timestamp of the last access, and its location in the min heap in the *node*. The pointers of *node* are sorted in the order of physical addresses in the array *nodes* (used to locate the node when updating the timestamp of a node without using LRU), while in the array *LRU* they form the min heap according to timestamp (used to find the least recently used entry). In order to facilitate the maintenance of the heap, I wrote two functions *perlocateUp()* and *perlocateDown()*.

**How to implement *vm\_write*:** Whenever the program need to write data, it first need to check the page table to confirm whether the virtual address need to be written exists in the page table. If it exists, overwrite the content of the corresponding physical memory, update the timestamp of the corresponding node, done. If it does not exist, then check whether there is any empty entry in the page table. If there is an empty entry, set its valid bit to valid, store the virtual address, write the content in the corresponding physical memory page, update the timestamp of the corresponding node, done. If the page table is full, call LRU, store the physical memory page corresponding to the least recently used entry to disk, transfer the page that needs to be written from the disk and store it in the vacated physical memory page, change the virtual address attribute of the enrty, and update the timestamp of the corresponding node, done.

**How to implement *vm\_read*:** Whenever the program need to read data, it first need to check the page table to confirm whether the virtual address need to be read exists in the page table. If it exists, update the timestamp of the corresponding node, and return its corresponding physical memory content, done. If it does not exist (the page table should be full at this time), call LRU, store the physical memory page corresponding to the least recently used entry to disk, transfer the page that needs to be read from the disk and store it in the vacated physical memory page, change the virtual address attribute of the enrty, update the corresponding The timestamp of the node, and return its corresponding physical memory content, done.

## 1.4 Page fault number:

Page fault number of my output is 8193, it comes out from  $4096 + 1 + 4096$ . The total amount of data is 4096 pages. There were 4096 page faults during sequential writing in. When reading the last 32769 pieces of content, 1025 pages were included, and another page fault was experienced. Finally, 4096 pages are read sequentially, and there are 4096 page faults again.

## 1.5 Problem I met:

After I exactly figured out the assignment requirements, there was no major problem. I only encountered one bug in programming, that is, my page fault count was not added. After checking the code, I found that the *pagefault\_num\_ptr* attribute I added is a pointer, the problem is solved after dereferencing it.

## 1.6 Screenshots of my program output

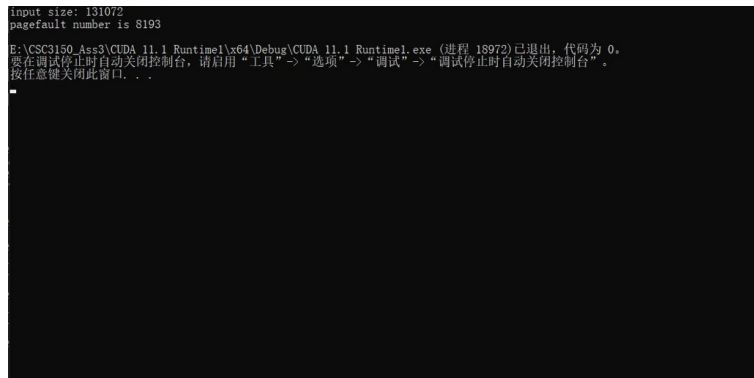


Figure 1: Screenshots when the program terminated

## 1.7 What did I learn from the task

I have an in-depth understanding of the concept and implementation of LRU, inverted page table, and content swap.

## 2 Bonus

### 2.1 The environment of running my program

The same as that in Task1 part.

### 2.2 The steps to execute my program

The same as that in Task1 part.

### 2.3 How did I design the program

My understanding of the bonus requirements is that each thread has independent memory spaces. Then each thread has to wait for others to complete otherwise the storage cannot afford to 4 thread's parallel writing operation.

**How to modify page table:** I need 1-bit valid bit and 27-bit vpn, a 32-bit entry is enough to store them. So I evenly distribute the 16KB pt to four threads, each with 4KB(1024 entries). For 32 bits, I set the highest bit as valid bit, and the lowest 27 bits as vpn. If the stored content is less than 0x80000000, the entry is valid, otherwise it is invalid.

**How to modify *mykernel*:** Use `__syncthreads()` to separate 4 stages, and each stage allows one thread to execute the user program. Use `threadIdx.x` to get the pid of the current thread, and determine whether the thread should execute the user program at this stage according to the pid.

**How to modify *vm\_write* and *vm\_read*:** When reading the page table, add an offset determined by the pid to the index to ensure that the corresponding page table is accessed.

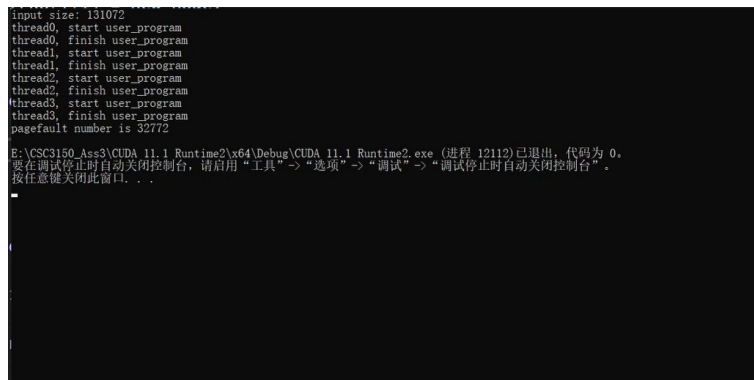
## 2.4 Page fault number:

Page fault number of my output is 32772, it comes out from  $8193 * 4$ . Since 4 threads execute the same program one by one sequentially.

## 2.5 Problem I met:

I found that I cannot access `threadIdx.x` in *vm\_write* and *vm\_read*. So I set a new public attribute *pid* to *VirtualMemory* structure to store pid information get from *mykernel* in *main.cu* to solve the problem.

## 2.6 Screenshots of my program output



```

input size: 131072
thread0, start user_program
thread0, finish user_program
thread1, start user_program
thread1, finish user_program
thread2, start user_program
thread2, finish user_program
thread3, start user_program
thread3, finish user_program
pagefault number is 32772
E:\CSC3150_Ass3\CUDA 11.1 Runtime2\x64\Debug\CUDA 11.1 Runtime2.exe (进程 12112) 已退出, 代码为 0。
要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”。
按任意键关闭此窗口。

```

Figure 2: Screenshots when the program terminated(bonus)

## 2.7 What did I learn from the task

I have improved my proficiency in the use of CUDA multithreading.