

Operating System:
Virtual Memory Management

CONTENTS

1. Development environment	3
2. Execution Steps	4
3. Program Design	5
3.1 Overview	
3.2 Implementation	
4. Page fault & Interpretation	11
4.1 User Program 1 (8193 Page faults)	
4.2 User Program 2 (9215 Page faults)	
5. Program Output	12
6. Encountered Problems & Solution	13
7. Learning outcome	15

1. Development environment

For this assignment, I used cluster **Slurm** provided by CSC4005 to write and test my program. Specifically, there is an overview given by the CSC4005 TAs:

Slurm is an open source, fault-tolerant, and highly scalable system that we choose for cluster management and job scheduling. As a cluster workload manager, Slurm has three key functions.

- 1. It allocates exclusive and/or non-exclusive access to resources (compute nodes) to users for some duration of time so they can perform work.*
- 2. It provides a framework for starting, executing, and monitoring work (normally a parallel job) on the set of allocated nodes.*
- 3. It arbitrates contention for resources by managing a queue of pending work.*

The **working system configurations** are described below:

Item	Configuration / Version
System Type	x86_64
Operating 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

Figure 1: System Config

The host machine (the physical machine I used to connect to the slurm) configurations are described below:

- System Type: x64-based PC
- Operating System: Windows 11 10.0.22621
- CPU: Intel(R) Core (TM) i7-10750H CPU @ 2.60GHz
- Memory: 16GB
- GPU: NVIDIA GeForce RTX 2060
- GCC: 12.2.0
- CMake: 3.23.0

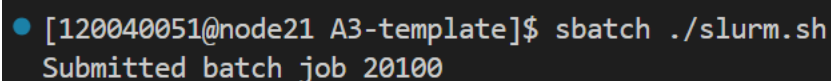
In order to establish a connection between my PC and the cluster, I used **Romote SSH** extension in the VS Code.

2. Execution Steps

In this assignment, the program is submitted to the cluster for execution using **sbatch** with the provided batch script. Specifically:

sbatch is used to submit a job script for later execution. The script will typically contain one or more srun commands to launch parallel tasks. If you use sbatch for job submission, you will get all the output recorded in the file that you assign in the batch script.

The corresponding command line is: `sbatch ./slurm.sh`



```
• [120040051@node21 A3-template]$ sbatch ./slurm.sh
Submitted batch job 20100
```

Figure 2: use sbatch to submit

In addition, the program can also run in an interactive way using self-defined **salloc** and **srun** commands. Specifically:

salloc is used to allocate resources for a job in real time. Typically, this is used to allocate resources and spawn a shell. The shell is then used to execute **srun** commands to launch parallel tasks.

srun is used to submit a job for execution in real time.

srun hostname prints the hostname of the nodes allocated.

For simplicity, during the writing and testing, I only adopt the method using **sbatch** with the given batch script.

3. Program Design

3.1 Overview

In this project, the target is to implement simple virtual memory in a kernel function of GPU that have a single thread, limit shared memory and global memory.

The basic program structure is illustrated below:

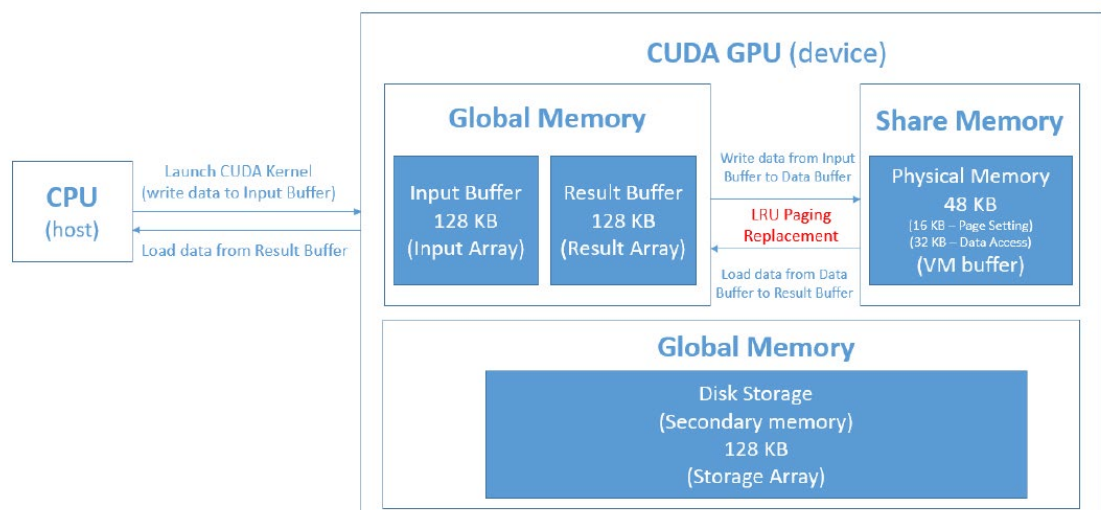


Figure 3: Program Structure

Configurations:

- Secondary memory (global memory): 128KB (131072 bytes)

-
- Physical memory (share memory): 48KB (49152 bytes)
32KB for data access; 16KB for page table setting
 - Page size: 32 bytes
 - Page table entries: 1024 (32KB / 32 bytes)

From a more detailed perspective, when the program tries to **read or write** a value from the given virtual address (VA), it first extracts the virtual page number (VPN) and offset from the VA and looks up the VPN in the inverted page table (IPT):

- If the corresponding VPN exists in the IPT, it is a **page hit**. The program will get the corresponding frame number from the IPT and goes to the specific address (i.e., frame number + offset) to read or write the target value;
- If not, then it is a **page fault**. The program will search the VPN in the swap table:
 - If the VPN exists in the swap table, load the corresponding frame from the disk and store it to the Least Recent Used (LRU) page in the memory, and store the original LRU page in the memory to the disk, then update the LRU information and the swap table;
 - If the VPN does not exist in the swap table, just return an error.

The following figure (Figure 4) gives a brief illustration of the logic mentioned above.

Regarding to the logic mentioned above and the provided assignment template, there are mainly three structures that needs to be designed and implemented by ourselves, namely
1. inverted page table 2. LRU algorithm 3. swap table.

In the next section 3.2, I will give a detailed report for the above three structures implemented in the program and how they interact with each other to function as a whole.

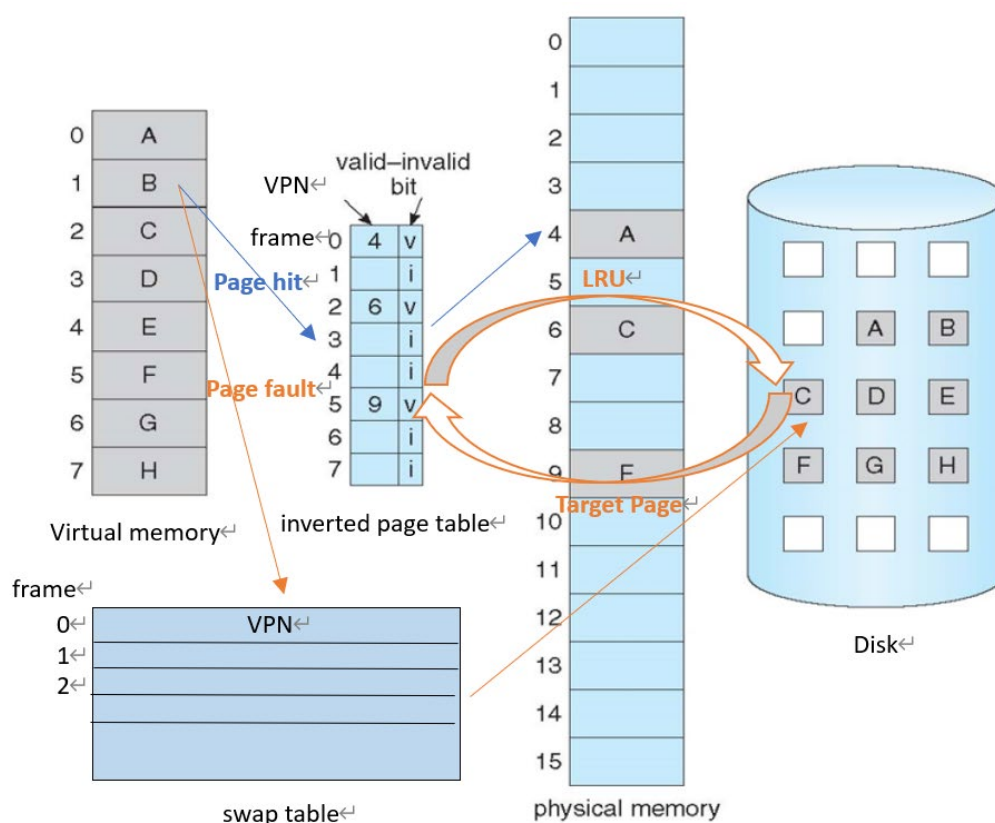


Figure 4: Memory Read & Write Logic

3.2 Implementation

Before implementing any data structures mentioned above, one key task is to extract the VPN and the offset from the VA. In this program, this is handled by simple bits operation:

```
/* write a single element to data buffer */
__device__ void vm_write(VirtualMemory *vm, u32 addr, uchar value) {

    unsigned offset = addr & 0x1f;           // extract offset from addr
    u32 page_number = (addr >> 5) & ((1 << 13) - 1); // extract the page number from addr
}
```

Figure 5: extract address and offset

As mentioned in Section 3.1, the page size is 32B, and 1 value (unsigned char) is 1B, so one page consists of $32 = 2^5$ entries. Therefore, **the offset is the last 5 bits** in the VA;

On the other hand, the virtual memory in this program is 160KB (32KB memory + 128 disk), and $2^{17} \text{ B} < 160\text{KB} < 2^{18} \text{ B}$, so in order to

support all the 160K (163840) entry addresses, **the page number should be the upper 13 (18-5) bits before the offset.**

For simplicity and with limited time, the program designed and implemented the above three structures using **static arrays**.

- **Inverted page table (IPT)**

```
// page table does not include the 5-bit offset
// the lower
for (int i = 0; i < vm->PAGE_ENTRIES; i++) {
    vm->invert_page_table[i] = EMPTY_ENTRY; // invalid := MSB is 1

    vm->invert_page_table[i + vm->PAGE_ENTRIES] = i; // corresponding frame index

    vm->invert_page_table[i + vm->PAGE_ENTRIES*2] = EMPTY_ENTRY; // times after the last reference
}
```

Figure 6: IPT

In total, the inverted page table should have $1024 \times 3 = 3072$ entries. With 4 Bytes for each entry (uint32_t), **the total size of the IPT is designed to be 12 KB**, which satisfies the requirement of maximum 16KB of page table setting.

EMPTY_ENTRY is defined as $1 \ll 31$ here. It just represents an invalid entry, no other operations involved.

- **LRU algorithm (LRU)**

In this program, the LRU algorithm is maintained in the third 1024 entries in the page table, which records the total reference times after the last reference, i.e., **how many references the program has made to the other pages since the last reference to this page**. *These entries are referred to as LRU bits in the following.*

As a result, every time the program refers to a page in the inverted page table, no matter read or write, page hit or just have loaded from the disk, the LRU bit for this entry is clear to 0, and the LRU bit for the rest entries +1. (Figure 7)

In this way, when the program searches for the LRU page number

in the IPT, it can just loop over all the LRU bit entries and find the largest one to be the LRU page. (Figure 8)

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

Figure 7: Set LRU bit after page hit

```
for (int i = 0; i < vm->PAGE_ENTRIES; i++) {
    if (vm->invert_page_table[i + vm->PAGE_ENTRIES * 2] > LRU_fqcy) {
        LRU_VPN = vm->invert_page_table[i];
        LRU_index = vm->invert_page_table[i + vm->PAGE_ENTRIES];
        LRU_fqcy = vm->invert_page_table[i + vm->PAGE_ENTRIES * 2];
    }
}
```

Figure 8: Find the LRU

- **Swap Table (ST)**

Similar to the IPT, the **swap table is also inverted** in order to be more time and spatial efficient (total disk frame numbers < total VPNs). It stores the information about the pages that have been swapped out of the memory to store in the disk.

The swap table has **total 4096 entries** since the disk has 128KB for storing array and the page size is 32B ($128\text{KB}/32\text{B} = 4\text{K} = 4096$). So the index of each index represents to frame number in the disk, and each entry is the VPN.

```
// initialize swap table entry's MSB to 1
for (int i = 0; i < ENTRY_NUMER; i++) {
    swap_table[i] = EMPTY_ENTRY;
}
```

Figure 9: Swap table initialization

```
// implement swap table in the global memory (disk)
// store virtual page number and corresponding disk number
__device__ __managed__ u32 swap_table[ENTRY_NUMER * 2];
```

Figure 10: Swap table declaration

Note that the swap table is declared to have 4096 (ENTRY_NUMER) x 2 entries is aimed to avoid some potential stack overflow problems. In fact, as mentioned, only the first 4096 entries are used.

At the time when the page fault occurs, these three structures need to work together to do the swap operation.

```
// [MEM TO DISK]: LRU page to empty disk frame
// [DISK TO MEM]: matched disk frame to LRU memory page
for (int i=0; i < ENTRY_PER_PAGE; i++) {
    tmp_buffer = vm->buffer[mem + i];
    vm->buffer[mem + i] = vm->storage[disk + i];
    vm->storage[disk_empty + i] = tmp_buffer;
}
```

Figure 11: Swap Operation upon read page fault

After data swapping, the program updates the IPT and ST.

```
// update page table, swap table
vm->invert_page_table[LRU_index] = page_number;
vm->invert_page_table[LRU_index + vm->PAGE_ENTRIES * 2] = 0;
swap_table[empty_frame] = LRU_VPN;
```

Figure 12: Update tables

4. Page fault & Interpretation

4.1 User Program 1

The first (default) user program should generate **8193** page faults. The input size is 128KB, with each value taking up 1B.

```
__device__ void user_program(VirtualMemory *vm, uchar *input, uchar *results,
                             int input_size) {
    for (int i = 0; i < input_size; i++)
        vm_write(vm, i, input[i]);

    for (int i = input_size - 1; i >= input_size - 32769; i--)
        int value = vm_read(vm, i);

    vm_snapshot(vm, results, 0, input_size);
}
// expect page fault num: 8193
```

Figure 13: user program 1

To derive 8193 page faults, **the first for loop will generate 4096 page faults** (128KB / 32B = 4K) with one page fault for each page; **the second for loop will generate 1 page fault** at $i = \text{input_size} - 32769$, which belongs to the 1025th page counting from the last, which just should not be in the inverted page table, which stored the last 1024 pages; **the last expression will also generate 4096 page faults**, since it just read sequentially starting from VA = 0.

As a result, in total we have **4096+1+4096 = 8193** page faults.

4.2 User Program 2

Another user program is given by the TA for testing.

```
__device__ void user_program(VirtualMemory *vm, uchar *input, uchar *results,
                             int input_size) {
    // write the data.bin to the VM starting from address 32*1024
    for (int i = 0; i < input_size; i++)
        vm_write(vm, 32*1024+i, input[i]);
    // write (32KB-32B) data to the VM starting from 0
    for (int i = 0; i < 32*1023; i++)
        vm_write(vm, i, input[i+32*1024]);
    // readout VM[32K, 160K] and output to snapshot.bin, which should be the same with data.bin
    vm_snapshot(vm, results, 32*1024, input_size);
}
// expected page fault num: 9215
```

Figure 14: user program 2

This user program should generate **9125** page faults.

To derive it, **the first for loop will generate 4096 page faults**, as it write sequentially from VM = 32K to 160K (128KB / 32B = 4K); the **second for loop will generate another 1023 page fault**, as it write sequentially from VM = 0 to 32K – 32 – 1 . At this time, the inverted page table should store 1023 VPNs from VM = 0 to 32K – 32 – 1 and 1 VPN for the VA = 160K – 32 – 1 to 160K (last page); finally, **the third expression with snapshot will generate another 4096 page faults**, since it just reads sequentially from the VA = 32K – 1 to VA = 160K – 1, and there will be no page hit along the reading. And the snapshot file is expected to be consistent with the input file.

As a result, in total we have **4096+1023+4096 = 9215** page faults.

5. Program Output

The estimated execution time for user program 1 & 2 is **around 45 seconds**.

User Program 1

Page fault 8193

```
main.cu(92): warning #2464-D: conversion from a string literal to "char *" is deprecated
main.cu(112): warning #2464-D: conversion from a string literal to "char *" is deprecated
main.cu(92): warning #2464-D: conversion from a string literal to "char *" is deprecated
main.cu(112): warning #2464-D: conversion from a string literal to "char *" is deprecated
input size: 131072
pagefault number is 8193
```

Figure 15: user program 1 results

User Program 2:

Page fault: 9215

```
main.cu(92): warning #2464-D: conversion from a string literal to "char *" is deprecated
main.cu(112): warning #2464-D: conversion from a string literal to "char *" is deprecated
main.cu(92): warning #2464-D: conversion from a string literal to "char *" is deprecated
main.cu(112): warning #2464-D: conversion from a string literal to "char *" is deprecated

input size: 131072
pagefault number is 9215
```

Figure 16: user program 2 output

No difference between snapshot.bin and data.bin (diff command returns no results indicating two files are identical)

```
[120040051@node21 A3-template]$ diff snapshot.bin data.bin
[120040051@node21 A3-template]$
```

Figure 17: compare files

6. Encountered Problems & Solution

There are many small problems that I have encountered for this assignment, such as not familiar with interacting with the slurm, not familiar with the basic knowledges and program structures about CUDA, don't know what is a swap table and etc. Such problems regarding to a lack of knowledge can be easily surmounted by carefully go through tutorial materials and loop up in the Internet.

However, one major problem I encountered during the debugging is that **the problem cannot derive the correct page fault number for the user program 2 while it can dump the file correctly and can pass the user program 1.**

Since I prefer to debug by myself, it took me about 3 days to figure out the underlying reason that accounts for this problem. It was really a hard time, and I had gone through the whole program structure

carefully with many handwritten drafts ...

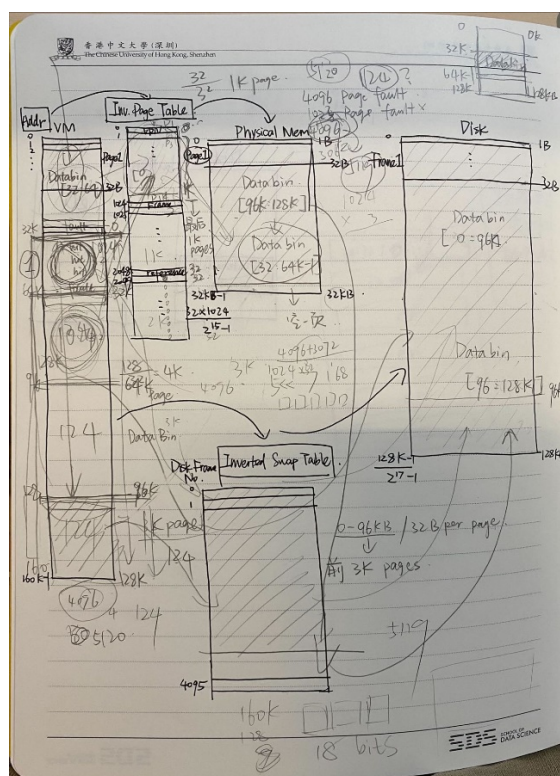


Figure 18: Some piece of debugging draft

The problem lies in the design logic of the LRU algorithm. Initially, I designed the LRU bits by maintaining the total reference times for each VPN. However, this design has pitfalls: it cannot tell the most or least used recent page in general cases.

A simple example is that if a page is referred just now, but only with 1 page hit, then by the previous logic, it would be the LRU page comparing with other pages with more than one page hits. But it is clearly not the case: the program just referred to it, and actually it is the most recent used page.

As a result, the solution is to store the total reference times the program has made since the last reference to this page. Specifically, if the program refers to this page in the page table, the value of the LRU bit will be set as 0 (the reference gap is 0), and the value of the LRU bit for other pages in the page table will +1. To search the LRU, the program just needs to loop through the LRU bits and find the largest one.

7. Learning Outcome

Through this assignment, I've learned:

1. How to interact with slurm cluster for program execution and debugging.
2. The fundamental concepts and some basic skills for CUDA programming.
3. The work flow for the virtual memory management program.
4. The concept of page fault, and how to calculate and interpret it.
5. The concept and the working mechanism of, together with different methods to implement: page table (inverted page table), LRU algorithm and swap table.