

## Assignment 3

CSC3150

Student Number: 117010032

Student Name: 陈雅茜



香港中文大學(深圳)

The Chinese University of Hong Kong, Shenzhen

## Introduction: Environment and Steps to Compile

This program is written to simulate the virtual memory management mechanism based on paging and inverted page table. The program is implemented on CUDA and tested on the windows OS with CUDA version 9.2.148, VS version 2017, GPU for NVIDIA GeForce GTX 1060 6GB. The user should input the user program and a binary file, the program will automatically put all the data into the snapshot.bin file.

Noticed: the program will roughly take 13 minutes to finish the execution, please wait for it patiently.

The input user program should be like

```
1 #include "virtual_memory.h"
2 #include <cuda.h>
3 #include <cuda_runtime.h>
4
5 __device__ void user_program(VirtualMemory *vm, uchar *input, uchar *results,
6                             int input_size) {
7     for (int i = 0; i < input_size; i++)
8         vm_write(vm, i, input[i]);
9
10    for (int i = input_size - 1; i >= input_size - 32769; i--)
11        int value = vm_read(vm, i);
12
13    vm_snapshot(vm, results, 0, input_size);
14 }
```

The input data file should be like

```
37 0C 60 51 38 5A 25 0B 13 2A 2A 01 0C 07 05 2F
62 0B 3A 2B 3E 35 2B 3C 0E 27 60 3B 04 53 2A 3A
5E 5A 5B 31 1F 1B 0B 01 14 04 01 20 0B 05 1E 08
44 57 02 1D 5C 61 28 05 23 58 3F 5A 46 39 63 3F
62 59 3F 1C 44 49 1C 57 1C 51 46 5A 55 63 32 34
56 33 21 1D 2F 18 21 21 0B 30 16 50 38 15 2A 35
3D 38 20 1C 50 3B 43 08 5B 24 31 4C 23 62 4F 48
01 3F 64 63 57 54 20 31 1F 05 1D 56 4D 16 5A 26
1E 15 11 09 20 53 44 16 13 11 31 05 0E 1C 4C 0E
2A 1B 41 50 0A 60 1D 5D 34 39 1E 51 1E 14 46 3B
5C 56 44 4B 45 57 61 27 03 61 2B 45 4C 46 22 12
60 62 61 39 2D 19 31 61 21 4F 4D 3F 32 62 49 29
53 5C 10 03 4F 40 5D 51 3C 57 31 24 38 23 05 03
20 01 0C 4D 4E 3C 19 0A 5A 35 18 5B 32 31 20 54
28 63 57 12 0E 1F 33 49 12 33 3C 49 55 40 1C 45
11 27 2D 2E 32 15 37 28 49 1F 1E 4A 4F 0D 39 46
3F 5F 58 4C 1A 5A 01 2B 28 3C 43 4D 4C 5E 2D 2C
20 29 59 22 3D 5F 19 55 4D 36 3A 37 13 42 19 51
3D 40 09 26 35 09 50 2C 14 2E 14 5F 5C 40 5A 4B
38 4E 08 10 19 20 34 01 26 09 38 08 1B 20 28 57
2F 30 4C 33 38 37 5E 4C 34 42 16 5F 51 0C 46 25
29 1D 34 41 3D 04 42 32 40 49 39 2A 38 60 50 02
2C 37 34 33 3D 61 1A 0D 3E 64 3B 2B 0B 1C 4F 03
39 52 44 45 25 21 46 35 39 4E 5E 0C 49 4A 0D 44
1C 10 47 59 0C 30 35 1A 2F 0B 14 09 5B 32 0C 63
1F 1F 13 14 3F 58 48 47 41 11 52 59 2A 5E 09 16
3D 4F 3E 48 1A 0E 31 19 4C 44 21 12 45 60 10 34
1A 22 47 28 49 5E 0A 25 3E 5B 1A 04 24 22 19 60
40 56 14 59 33 44 0D 1A 58 62 60 38 5D 0B 07 47
61 1D 0A 15 16 14 3A 54 3E 23 27 62 14 3F 5D 53
30 0C 17 62 20 58 4B 13 25 16 1A 1D 55 55 33 51
```

The output file should be like

| data.bin | main.cu                 | snapshot.bin            | virtual_memory.cu   |
|----------|-------------------------|-------------------------|---------------------|
| 00000000 | 37 0C 60 51 38 5A 25 0B | 13 2A 2A 01 0C 07 05 2F | 7. `Q8Z%...**.... / |
| 00000010 | 62 0B 3A 2B 3E 35 2B 3C | 0E 27 60 3B 04 53 2A 3A | b. :>5+<.'` : S*:   |
| 00000020 | 5E 5A 5B 31 1F 1B 0B 01 | 14 04 01 20 0B 05 1E 08 | `Z[1..... ..        |
| 00000030 | 44 57 02 1D 5C 61 28 05 | 23 58 3F 5A 46 39 63 3F | DW.. \a(.#X?ZF9c?   |
| 00000040 | 62 59 3F 1C 44 49 1C 57 | 1C 51 46 5A 55 63 32 34 | bY?.DI.W.QFZUc24    |
| 00000050 | 56 33 21 1D 2F 18 21 21 | 0B 30 16 50 38 15 2A 35 | V3!./.! !.0.P8.*5   |
| 00000060 | 3D 38 20 1C 50 3B 43 08 | 5B 24 31 4C 23 62 4F 48 | =8 .P:C. [\$1L#bOH  |
| 00000070 | 01 3F 64 63 57 54 20 31 | 1F 05 1D 56 4D 16 5A 26 | .?dcWT 1...VM.Z&    |
| 00000080 | 1E 15 11 09 20 53 44 16 | 13 11 31 05 0E 1C 4C 0E | .... SD...1...L.    |
| 00000090 | 2A 1B 41 50 0A 60 1D 5D | 34 39 1E 51 1E 14 46 3B | *.AP.`.]49.Q..F:    |
| 000000a0 | 5C 56 44 4B 45 57 61 27 | 03 61 2B 45 4C 46 22 12 | \VDKEWa'.a+ELF".    |
| 000000b0 | 60 62 61 39 2D 19 31 61 | 21 4F 4D 3F 32 62 49 29 | `ba9-.1a!OM?2bI)    |
| 000000c0 | 53 5C 10 03 4F 40 5D 51 | 3C 57 31 24 38 23 05 03 | S\..0@]Q<W1\$8#..   |
| 000000d0 | 20 01 0C 4D 4E 3C 19 0A | 5A 35 18 5B 32 31 20 54 | ..MN<..Z5.[21 T     |
| 000000e0 | 28 63 57 12 0E 1F 33 49 | 12 33 3C 49 55 40 1C 45 | (cW...3I.3<IU@.E    |
| 000000f0 | 11 27 2D 2E 32 15 37 28 | 49 1F 1E 4A 4F 0D 39 46 | .'-.2.7(I..JO.9F    |
| 00000100 | 3F 5F 58 4C 1A 5A 01 2B | 28 3C 43 4D 4C 5E 2D 2C | ? XL.Z.+(<CML^-,    |
| 00000110 | 20 29 59 22 3D 5F 19 55 | 4D 36 3A 37 13 42 19 51 | )Y"=..UM6:7.B.Q     |
| 00000120 | 3D 40 09 26 35 09 50 2C | 14 2E 14 5F 5C 40 5A 4B | =@.&5.P...._\@ZK    |
| 00000130 | 38 4E 08 10 19 20 34 01 | 26 09 38 08 1B 20 28 57 | 8N...4.&.8..(W      |
| 00000140 | 2F 30 4C 33 38 37 5E 4C | 34 42 16 5F 51 0C 46 25 | /0L387^L4B..Q.F%    |
| 00000150 | 29 1D 34 41 3D 04 42 32 | 40 49 39 2A 38 60 50 02 | ).4A=.B2@I9*8`P.    |
| 00000160 | 2C 37 34 33 3D 61 1A 0D | 3E 64 3B 2B 0B 1C 4F 03 | ,743=a..>d;+..0.    |
| 00000170 | 39 52 44 45 25 21 46 35 | 39 4E 5E 0C 49 4A 0D 44 | 9RDE%!F59N`.IJ.D    |
| 00000180 | 1C 10 47 59 0C 30 35 1A | 2F 0B 14 09 5B 32 0C 63 | ..GY.05./...[2.c    |
| 00000190 | 1F 1F 13 14 3F 58 48 47 | 41 11 52 59 2A 5E 09 16 | ....?XHGA.RY*..     |
| 000001a0 | 3D 4F 3E 48 1A 0E 31 19 | 4C 44 21 12 45 60 10 34 | =O>H..1.LD!.E`.4    |
| 000001b0 | 1A 22 47 28 49 5E 0A 25 | 3E 5B 1A 04 24 22 19 60 | ..`G(I`.%)>[.\$.`.  |
| 000001c0 | 40 56 14 59 33 44 0D 1A | 58 62 60 38 5D 0B 07 47 | @V.Y3D..Xb`8]..G    |
| 000001d0 | 61 1D 0A 15 16 14 3A 54 | 3E 23 27 62 14 3F 5D 53 | a.....>T#`b.?)S     |
| 000001e0 | 30 0C 17 62 20 58 4B 13 | 25 16 1A 1D 55 55 33 51 | 0..b XK.%...UU3Q    |
| 000001f0 | 0D 3D 01 57 20 0A 16 5D | 60 3C 5A 0F 4A 23 31 15 | ..=..W..]>Z.J#1.    |
| 00000200 | 62 18 46 1D 0B 60 63 63 | 12 19 1B 36 3D 4E 22 49 | b.F...`cc...6=N`I   |
| 00000210 | 5A 56 0B 15 30 20 41 2B | 2B 07 0A 10 29 3A 58 26 | ZV..0 A++...):X&    |
| 00000220 | 21 09 13 2B 05 11 5D 4A | 5D 14 1B 35 31 0C 4E 5A | !...+..]J]..51.NZ   |
| 00000230 | 31 58 0A 30 48 1A 5B 0E | 20 34 52 48 3D 45 3E 5E | 1X.OH.[.4RH=E>      |
| 00000240 | 4E 20 58 22 64 51 3B 5D | 34 25 61 64 64 1A 29 30 | N X`dq;]4%add.)0    |
| 00000250 | 42 02 60 59 1B 26 36 3B | 59 23 52 31 38 5F 5E 21 | B.`Y.&6;Y#R18`!     |
| 00000260 | 1A 52 12 1A 3E 1C 46 0D | 40 12 40 3F 2C 04 3E 3D | .R...>.F.@?..>=     |
| 00000270 | 39 09 31 53 2E 02 29 56 | 25 4B 57 5C 45 50 4C 5F | 9.1S..)V%KW\EPL_    |
| 00000280 | 3D 2D 14 4A 48 29 56 23 | 3A 01 31 01 38 0A 3D 0C | =-.JH)V#:.1.8.=.    |
| 00000290 | 13 3D 5F 10 3F 57 02 33 | 3D 28 5E 52 47 45 4C 20 | ..=?W.3=(`RGEL      |
| 000002a0 | 41 2F 39 24 57 2B 16 2C | 5F 16 61 33 1F 09 3E 01 | A/9\$W+...a3...>.   |
| 000002b0 | 46 38 11 54 5F 46 22 37 | 3D 1B 24 1F 2F 3F 0E 0B | F8.T_F`7=.\$./?..   |
| 000002c0 | 09 47 62 2F 0D 47 5B 07 | 5C 27 09 4A 2F 47 1B 10 | .Gb/.G[.`\`J/G..    |
| 000002d0 | 1A 2B 63 48 40 54 1B 4C | 3E 0E 06 08 1D 14 46 59 | ..+cH@T.L>.....FY   |
| 000002e0 | 2A 43 24 06 25 4E 0C 50 | 10 15 06 3E 2B 20 1E 14 | *C\$.%N.P...>+ ..   |
| 000002f0 | 4A 50 5C 59 40 46 10 19 | 53 15 55 3F 5C 36 34 21 | JP\Y@F...S.U?\64!   |
| 00000300 | 15 57 26 09 40 02 29 1F | 4A 2E 5C 10 4D 15 57 02 | .W&.@.).J.\.M.W.    |
| 00000310 | 01 4E 2A 10 2F 39 28 52 | 1D 4C 2C 15 1E 5F 05 02 | .N*./9(R.L....      |
| 00000320 | 21 5F 0A 30 30 02 4E 49 | 2F 16 28 4B 5E 1A 4C 5E | !_.00.NI/.(K`.L`    |
| 00000330 | 04 11 3D 32 19 35 53 36 | 1C 1B 1A 39 49 52 3A 3A | ..=2.5S6...9IR:1    |
| 00000340 | 4C 14 05 4B 15 23 2F 14 | 08 56 5E 01 0C 16 5F 43 | L..K.#/..V`...C     |
| 00000350 | 5A 07 44 0F 3B 33 14 57 | 1D 2D 5F 01 4E 05 0A 36 | Z.D.:3.W.-..N.._6   |

You should use the “Ctrl” + “F7” to compile each CUDA file and use the “Ctrl” + “F5” to execute the program.

## Flow: How Did I Design My Program

The total process will consist of three parts:

1. Load the data from the binary data file to the input buffer.

```
__host__ void write_binaryFile(char *fileName, void *buffer, int bufferSize) {
    FILE *fp;
    fp = fopen(fileName, "wb");
    fwrite(buffer, 1, bufferSize, fp);
    fclose(fp);
}

__host__ int load_binaryFile(char *fileName, void *buffer, int bufferSize) {
    FILE *fp;

    fp = fopen(fileName, "rb");
    if (!fp) {
        printf("****Unable to open file %s****\n", fileName);
        exit(1);
    }

    // Get file length
    fseek(fp, 0, SEEK_END);
    int fileLen = ftell(fp);
    fseek(fp, 0, SEEK_SET);

    if (fileLen > bufferSize) {
        printf("****invalid testcase!****\n");
        printf("****software warning: the file: %s size****\n", fileName);
        printf("****is greater than buffer size****\n");
        exit(1);
    }

    // Read file contents into buffer
    fread(buffer, fileLen, 1, fp);
    fclose(fp);

    return fileLen;
}
```

2. Use the write function to write the data from the input buffer to the physical memory. When the physical memory is full, use the LRU mechanism to decide which page should be removed from the physical memory to the disk storage and put the current page into that frame.

```
__device__ void vm_write(VirtualMemory *vm, u32 addr, uchar value) {
    /* Complete vm_write function to write value into data buffer */
    u32 offset = addr % 32;
    u32 page_number = addr / 32;
    u32 frame_number;
    if (!check_page_fault(vm, page_number)) {
        frame_number = vm->invert_page_table[vm->PAGE_ENTRIES];
        check_frame_full(vm, page_number, frame_number);
        vm->invert_page_table[frame_number] = page_number;
    }
    else {
        frame_number = find_frame_number(vm, page_number);
    }
    vm->buffer[frame_number * 32 + offset] = value;
    change_frame_table_valid_to_invalid(vm, frame_number);
}
```

3. Use the read function to find the required pages first in the physical memory. If the target page is not in the physical memory, we should again use the LRU mechanism to swap the page in the storage with the page in the physical memory.

```
__device__ uchar vm_read(VirtualMemory *vm, u32 addr) {
    /* Complate vm_read function to read single element from data buffer */
    u32 offset = addr % 32;
    u32 page_number = addr / 32;
    u32 frame_number;
    if (!check_page_fault(vm, page_number)) {
        frame_number = vm->invert_page_table[vm->PAGE_ENTRIES];
        move_to_memory(vm, frame_number, page_number);
    }
    else {
        frame_number = find_frame_number(vm, page_number);
    }
    change_frame_table_valid_to_invalid(vm, frame_number);
}
```

4. Then dump the contents to “snapshot.bin”.

```
__device__ void vm_snapshot(VirtualMemory *vm, uchar *results, int offset,
    int input_size) {

    for (int i = 0; i < input_size; i++) {
        u32 page_number = i / 32;
        u32 frame_offset = i % 32;
        u32 frame_number;
        if (!check_page_fault(vm, page_number)) {
            frame_number = vm->invert_page_table[vm->PAGE_ENTRIES];
            move_to_memory(vm, frame_number, page_number);
        }
        else {
            frame_number = find_frame_number(vm, page_number);
        }
        memory_move_to_result(vm, results, page_number);
        change_frame_table_valid_to_invalid(vm, frame_number);
    }
}
```

## **Functions: How did I Design My Program**

1. `__device__ void init_invert_page_table(VirtualMemory *vm)` function is used to initialize the inverted page table. The first `PAGE_ENTRIES` entries

will be filled with 0x80000000 which indicates that there is currently no frame number in this entry.

The later PAGE\_ENTRIES entries is used as the frame list. The least recent used frame number will be stored at the top of the list and the most recently used one will be stored at the bottom of the list. It will be initialized with number i for the ith entry.

```
__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;  
    }  
}
```

2. \_\_device\_\_ void change\_frame\_table\_valid\_to\_invalid(VirtualMemory \*vm)  
function is used to update the frame list after each page is loaded in the physical memory. It will put the top frame number at the bottom of the list since this frame is just be used and put other frame numbers forward by one block.

```
__device__ void change_frame_table_valid_to_invalid(VirtualMemory *vm, u32 frame_number) {  
    int tempt = vm->invert_page_table[vm->PAGE_ENTRIES + find_frame_number_in_frame_table(vm, frame_number)];  
    for (int i = find_frame_number_in_frame_table(vm, frame_number); i < vm->PAGE_ENTRIES - 1; i++) {  
        vm->invert_page_table[i + vm->PAGE_ENTRIES] = vm->invert_page_table[i + vm->PAGE_ENTRIES + 1];  
    }  
    vm->invert_page_table[2 * vm->PAGE_ENTRIES - 1] = tempt;  
}
```

3. \_\_device\_\_ int find\_frame\_number\_in\_frame\_table(VirtualMemory \*vm, u32 frame\_number) this function is used to find the corresponding block in the frame list according to the given frame number. The frame list is the later part in the invert page table. It is initialized in the init\_invert\_page\_table function with the number i in ith entry. The top entry will store the frame number of the least recently used while the bottom entry will store the frame number of the most recently used one.

```
__device__ int find_frame_number_in_frame_table(VirtualMemory *vm, u32 frame_number) {  
    for (int i = 0; i < vm->PAGE_ENTRIES; i++) {  
        if (vm->invert_page_table[i + vm->PAGE_ENTRIES] == frame_number) return i;  
    }  
    printf("out of index\n");  
    return -1;  
}
```

4. `__device__ int find_frame_number(VirtualMemory *vm, u32 page_number)` function is used to find the corresponding frame number according to the given page number. It is implemented by searching through the page table and check if the page number storage in each entry of the page table is equal to the given page number. If found, return the frame number, if not return -1 and print the out of index error.

```
__device__ int find_frame_number(VirtualMemory *vm, u32 page_number) {
    for (int i = 0; i < vm->PAGE_ENTRIES; i++) {
        if (vm->invert_page_table[i] == page_number) return i;
    }
    printf("out of index\n");
    return -1;
}
```

5. `__device__ void move_to_storage(VirtualMemory *vm, u32 frame_number)` this function is used to move the page from physical memory to the storage according to the given frame number.

```
__device__ void move_to_storage(VirtualMemory *vm, u32 frame_number) {
    u32 page_number = vm->invert_page_table[frame_number];
    for (int i = 0; i < 32; i++) {
        vm->storage[page_number * 32 + i] = vm->buffer[frame_number * 32 + i];
    }
}
```

6. `__device__ void memory_move_to_result(VirtualMemory *vm, uchar* result, u32 page_number)` this function is used to move the data from physical memory to the result buffer when we dump the contents to the snapshot binary file by the given page number.

```
__device__ void memory_move_to_result(VirtualMemory *vm, uchar* result, u32 page_number) {
    u32 frame_number = find_frame_number(vm, page_number);
    for (int i = 0; i < 32; i++) {
        result[page_number * 32 + i] = vm->buffer[frame_number * 32 + i];
    }
}
```

7. `__device__ void move_to_memory(VirtualMemory *vm, u32 frame_number, u32 page_number)` this function is used to move the data from the storage to the physical memory according to the given page and frame numbers.



```

__device__ void move_to_memory(VirtualMemory *vm, u32 frame_number, u32 page_number) {
    u32 original_page_number = vm->invert_page_table[frame_number];
    for (int i = 0; i < 32; i++) {
        vm->storage[original_page_number * 32 + i] = vm->buffer[frame_number * 32 + i];
        vm->buffer[frame_number * 32 + i] = vm->storage[page_number * 32 + i];
    }
    vm->invert_page_table[frame_number] = page_number;
}

```

8. `__device__ bool check_page_fault(VirtualMemory *vm, u32 page_number)`  
 check pagefault function is used to determine if we can find the given page in the frame or not by searching through the invert page table. If the page fault occurred, the function will return false and add the page fault number by 1. If not, the function will return true.

```

__device__ bool check_page_fault(VirtualMemory *vm, u32 page_number) {
    for (int i = 0; i < vm->PAGE_ENTRIES; i++) {
        if (vm->invert_page_table[i] == page_number) {
            return true;
        }
    }
    *vm->pagefault_num_ptr = *vm->pagefault_num_ptr + 1;
    return false;
}

```

9. `__device__ void check_frame_full(VirtualMemory *vm, u32 page_number, u32 frame_number)` the function is used to check if the given frame has already been occupied or not by checking whether the number inside the corresponding entry in the page table is 0x80000000 or not. If the number is 0x80000000, this means that the frame has not been taken, if not it indicates that the frames has already been taken.

```

__device__ void check_frame_full(VirtualMemory *vm, u32 page_number, u32 frame_number) {
    if (vm->invert_page_table[frame_number] != 0x80000000) {
        move_to_storage(vm, frame_number);
    }
}

```

10. `__device__ uchar vm_read(VirtualMemory *vm, u32 addr)` read function is used to read the target pages from the buffer. We should firstly use the check page fault function to check if the target pages are in the physical memory or not. If the target pages are not in the physical memory we should use the move to memory function to move the pages from storage to physical memory. For each byte read, we should change the frame list since the recently used frame may be changed.



```

__device__ uchar vm_read(VirtualMemory *vm, u32 addr) {
    /* Complete vm_read function to read single element from data buffer */
    u32 offset = addr % 32;
    u32 page_number = addr / 32;
    u32 frame_number;
    if (!check_page_fault(vm, page_number)) {
        frame_number = vm->invert_page_table[vm->PAGE_ENTRIES];
        move_to_memory(vm, frame_number, page_number);
    }
    else {
        frame_number = find_frame_number(vm, page_number);
    }
    change_frame_table_valid_to_invalid(vm, frame_number);
}

```

11. `__device__ void vm_write(VirtualMemory *vm, u32 addr, uchar value)` write function is used to write the data from input buffer to the physical memory and storage. Each time when we are loading the page into the memory we should check if the frame is full or not. If the frame is already occupied by some page, we should move that page to the storage and then put the new page into the frame.

```

__device__ void vm_write(VirtualMemory *vm, u32 addr, uchar value) {
    /* Complete vm_write function to write value into data buffer */
    u32 offset = addr % 32;
    u32 page_number = addr / 32;
    u32 frame_number;
    if (!check_page_fault(vm, page_number)) {
        frame_number = vm->invert_page_table[vm->PAGE_ENTRIES];
        check_frame_full(vm, page_number, frame_number);
        vm->invert_page_table[frame_number] = page_number;
    }
    else {
        frame_number = find_frame_number(vm, page_number);
    }
    vm->buffer[frame_number * 32 + offset] = value;
    change_frame_table_valid_to_invalid(vm, frame_number);
}

```

12. `device__ void vm_snapshot(VirtualMemory *vm, uchar *results, int offset, int input_size)` the snapshot function is used to dump the contents into result buffer. It will go through each page in the input buffer and find

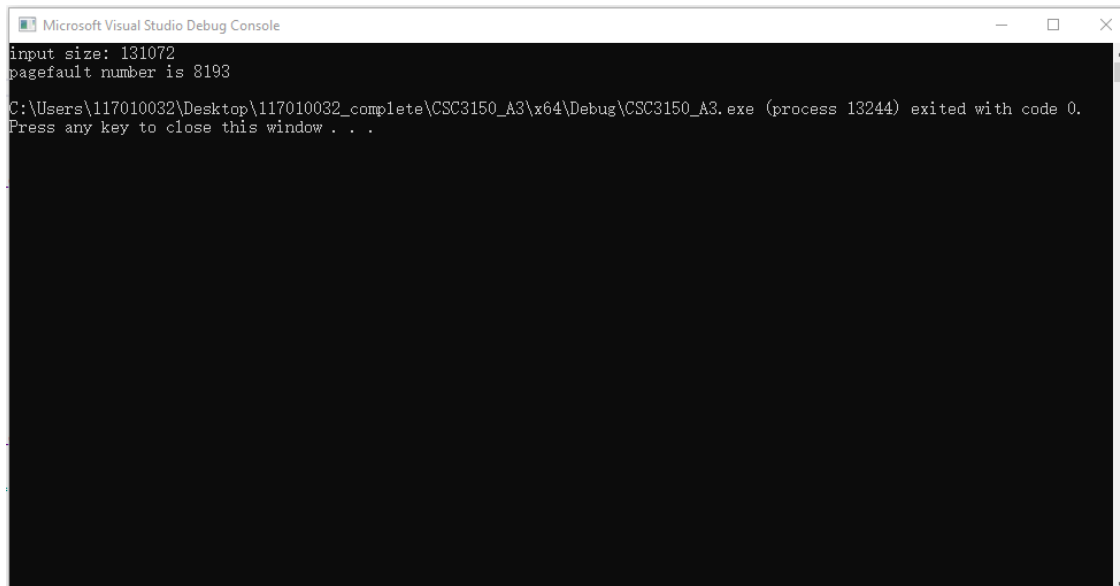
the page in the physical memory. If page is in the physical memory, we directly put the page from physical memory into the result buffer, otherwise we swap the page in storage with the page in the target frame, then move the page into the result buffer.

```
__device__ void vm_snapshot(VirtualMemory *vm, uchar *results, int offset,
    int input_size) {

    for (int i = 0; i < input_size; i++) {
        u32 page_number = i / 32;
        u32 frame_offset = i % 32;
        u32 frame_number;
        if (!check_page_fault(vm, page_number)) {
            frame_number = vm->invert_page_table[vm->PAGE_ENTRIES];
            move_to_memory(vm, frame_number, page_number);
        }
        else {
            frame_number = find_frame_number(vm, page_number);
        }
        memory_move_to_result(vm, results, page_number);
        change_frame_table_valid_to_invalid(vm, frame_number);
    }
}
```

## **Results: Screen Shot of My Output Result and What is My Page Fault and Reason**

The final console interface is shown as below, the page fault is 8193

A screenshot of the Microsoft Visual Studio Debug Console window. The window has a title bar that says "Microsoft Visual Studio Debug Console". The console output shows: "input size: 131072", "pagefault number is 8193", and "C:\Users\117010032\Desktop\117010032\_complete\CSC3150\_A3\x64\Debug\CSC3150\_A3.exe (process 13244) exited with code 0. Press any key to close this window . . .". The console window is black with white text. There are standard window controls (minimize, maximize, close) in the top right corner.

```
Microsoft Visual Studio Debug Console
input size: 131072
pagefault number is 8193
C:\Users\117010032\Desktop\117010032_complete\CSC3150_A3\x64\Debug\CSC3150_A3.exe (process 13244) exited with code 0.
Press any key to close this window . . .
```

The page fault number is 8193 which contains 4096 page-faults from the write section, 1 page-fault from the read section and 4096 page-faults from the snapshot section.

1. The page faults in write section is 4096 since every page loaded from the input buffer is a new page for the physical memory. There are totally 128K ( $2^7 * 2^{10}$  bytes) data with 32bytes in each page, we can have 4096 pages. Therefore, it will generate 4096 page-faults.
2. The page faults in the read section is 1. There are totally 32769 bytes required to be read in the physical memory according the user program. The size for those data is equal to 1024 pages and 1 byte with each page contains 32 bytes. Since the user program will read the data from the bottom of the data binary file, and the data are written from the top to the bottom in the write section, the first 1024 pages will be found in the physical memory. Therefore, only the last byte will generate page fault.
3. The page fault in the snapshot section is 4096 since in this section the program is required to read the data again from the top to the bottom. Each page will generate the page fault since only the bottom pages are stored in the physical memory at the beginning.

Therefore, there are totally 8193 page-faults generated in the whole process.

### **Results: What Are the Problems I Met in this Assignment and My Solutions**

1. How to construct the frame list in order to store the sequence of frame numbers.  
My solution is to use an array to store the sequence. The top entry will store the frame number of the least recently used and the bottom one will store the frame number of the most recently used. Whenever a frame has been used, we will swap the frame to the bottom of the list and move every frame number stored below it upward by one entry.
2. How to get the page number for each data stored in the storage.

My solution is to store the data in the storage according to their page number. Then, when we are tracking the page from the storage, we can get its page number according to its position.

3. When to update the frame list.

My solution is to update the frame list after any data is put in the physical memory.

4. How to get the frame number for each data

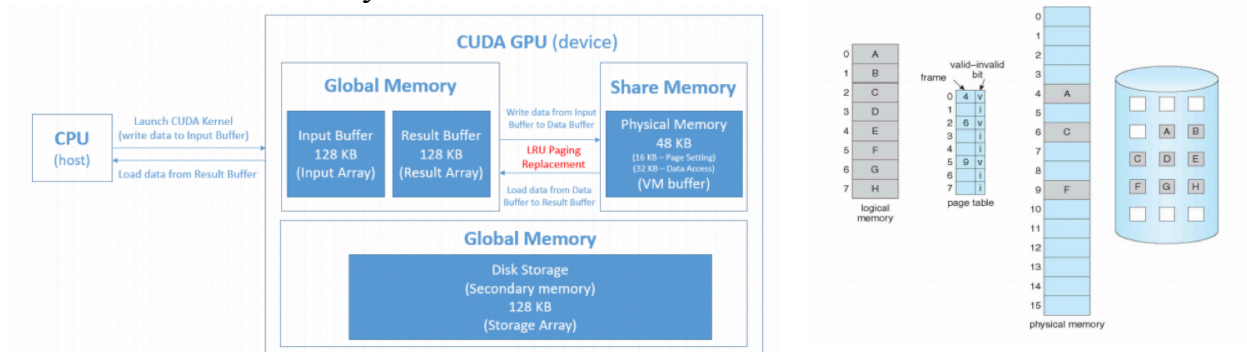
My solution is to assign the first frame in the frame list if the page fault occurred. Otherwise, we will go through the invert page table to find the corresponding frame for each data according to its page number.

## **Conclusion: What I have learnt from this program**

In this assignment I have learnt (1) the basic mechanism of memory management (2) how to use the invert page table to store the page-frame information (3) how to design the frame list (4) the LRU mechanism

(1) The paging mechanism is to divide the program or data into different pages and load pages into both shared memory and global memory. When we try to read the data, we will first search the page table to see if the corresponding page is in the physical memory or not. If the page is not in the physical memory, we will swap the page in the storage with the page in the memory.

The basic memory structure should be like below



(2) The invert page table is used to store the corresponding page number in the table with the index of the frame number. We can go through the whole table to see if the page is in the physical memory or not and also return the corresponding frame number.

- (3) There are multiple ways to design a frame list which is used to store the least recently used frame number. You can use the counter to design the list. It is an array which contains the number of rounds that one frame hasn't been used with its index indicates the frame number. For each time one frame is used, its corresponding number in the frame list will be changed to 0 while other counters will be added by 1. The index with the largest counter will be the frame number of the least recently used. We can also implement the frame list using stack.
- (4) The LRU mechanism is used to determine which frame should be used when page fault occurred. We will always swap the least recently used page into the storage in order to put the new page in.