OS Assignment3 Report

Chen Yian 118010030

October 2020

1 Introduction

In this assignment, we are required to implement a simple memory allocation system with page table using cuda. Being the first homework involving cuda programming, this one does not require a lot of cuda attributes. The challenging part in this homework is to fully understand page table and relevant operations.

First, before the author started demonstration, I shall clarify some notations involving bits.

1.1 Some notions involving bits

32-bit binary number is widely used in this homework. Thus, it is necessary for me to make some clarifications of wordings. Here, we count bit **from left to right**. For example, b_0 refers to the rightmost bit of the 32-bit number. Generally, we use b_n to represent the $(n+1)^{th}$ number from right to left.

1.2 Simple bit operations and macros

In this program, to simplify some common bit operations, the author defined several macros.

virtual_memory.h

```
#define set_bit(x,y)  (x|=(1<<y))
#define clr_bit(x,y)  (x&=~(1<<y))
#define get_bits(x, end, begin) ((x << (31 - end))>>(31 - (end - begin)))
#define get_bit(x,y)  get_bits(x, y, y)
```

Among them `get_bits` macro might be worth mentioning. This macro returns the end^{th} bit to $begin^{th}$ bit, where $end > begin$. This macros appear really useful when we want to slice the page or offset of an address. Also, in the remaining pages, readers will notice that all bits represent extra information lies on the left most bits. Thus, this macro will also help us to clear those bits when slicing pages. However, we need to be aware that this macro may as well bring us some bit overflow issue, thus we only use them to get page number.

2 Program Design

As the template already presents us with sufficient program structure. We may only be noticing the methods being used. We need to implement `vm_read`, `vm_write`, `vm_snapshot` by ourselves. With further considering, we shall find out that all three functions rely on one functions: Transforming virtual address to physical address, which we will denote as `vm_virtual_to_phys`.

virtual_memory.cu

```
__device__ uchar vm_read(VirtualMemory *vm, u32 addr) {
        int physical_addr = vm_virtual_to_phys(vm, addr);
        return vm->buffer[physical_addr];
}

__device__ void vm_write(VirtualMemory *vm, u32 addr, uchar value) {
        int physical_addr = vm_virtual_to_phys(vm, addr);
        if (physical_addr == -1) printf("Invalid address access!\n");
        vm->buffer[physical_addr] = value;
}

__device__ void vm_snapshot(VirtualMemory *vm, uchar *results, int offset,
                            int input_size) {
        for (int i = offset; i < input_size; i++) {
                uchar value = vm_read(vm, i);
                results[i] = value;
        }
}
```

After setting up the basic program structure, our most important task now, is certainly implementing the `vm_virtual_to_phys`. As following demonstrations, we may divide the implementation into two parts:

1. We will first use the virtual page from the virtual address to search the whole page table and see whether there is one corresponding physical page. If yes, then return the physical page plus the offset. If no, then turn to step 2.

2. We will check the corresponding second storage, and find one available page. The available page is either one empty page, or one page gained from

LRU algorithm. Then we will do swapping. After swapping, update the page table, and return the physical page plus the offset.

virtual_memory.cu

```
__device__ u32 vm_virtual_to_phys(VirtualMemory *vm, u32 addr) {
    /* Invalid memory access */
        if ((addr >> 5) > vm->PHYSICAL_MEM_SIZE) return -1;
        /* Search through the page table */
        u32 phys_page = vm_search_page_table(vm, addr);
        if (phys_page != -1) { // Can be found in the page table
                u32 phys_addr = ((phys_page << 5) | (get_bits(addr, 4, 0)));
                return phys_addr;
        }
        else { // Cannot be found in the page table, page fault occurs
                /* Page Fault! */
                (*vm->pagefault_num_ptr)++;

                phys_page = vm_page_fault(vm, addr);
                u32 phys_addr = ((phys_page << 5) | (get_bits(addr, 4, 0)));
                vm->charCounter = 0;
                return phys_addr;
        }
}
```

So far, we have clarified most of the implementations. Some details require further explanation though, including the LRU algorithm we implemented here.

2.1 LRU

LRU(least recently used) algorithm enables us to pick up the least recently used page. Many methods can be used to implement LRU method, including doubly linked list or stack. However, here we shall use a clock method to implement LRU.

Basically, we require a certain bit (in this program, the author specifically picked b_{30}) to be the LRU bit. Every time a page is called (created, read, written), the LRU bit will be set. And when LRU algorithm is invoked, one pointer will loop over the array, if pointer points to a variable with LRU bit set, the pointer will clear the LRU bit and loops to the next variable in array. The pointer will continuously looping until meeting one variable without a LRU bit.

virtual_memory.cu

```
__device__ u32 LRUsearch(VirtualMemory *vm) {
        int begin = vm->LRUptr;
        clr_bit(vm->invert_page_table[vm->LRUptr], 30);
        vm->LRUptr++;
        while (vm->LRUptr != begin) {
                vm->LRUptr %= 1024;
                if (((vm->invert_page_table[vm->LRUptr] & 0xf0000000) >> 30) =
                        return vm->LRUptr;
                } // Find b_{30}
                clr_bit(vm->invert_page_table[vm->LRUptr], 30); // clear b_{3
                vm->LRUptr++;
        }
        return begin;
}
```

2.2 Page fault number in total

2.2.1 Write

First 1024 pages will cause 1024 page fault. As initialized page tables are all empty and we need to swap the page into the empty page. The rest 3072 page will also cause 3072 page fault, as the page table is always full.

Write in total: 4096 faults.

Write result: Page table starts from page 3072 to page 4095

2.2.2 Read

Read takes an inverted order, thus read over 32768 address, i.e., from `input_size - 1` to `input_size - 32769`. By calculating, we loop over page 4095 to page 3071's last bit. Thus, one swap occurs.

Read in total: 1 fault

2.2.3 Snapshot

All the 4096 pages when invoking are not in the page table. Thus 4096 page faults occur

Snapshot in total: 4096 faults

In total: 8193 faults.

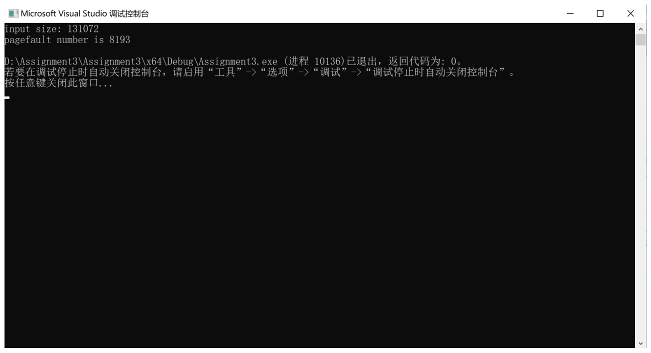
3 Environment

Windows 10 + Visual Studio 2017 + cuda 10.0

4 Execution Step

Open `Assignment3sln` with Visual Studio. Then make and run the project.

5 Result Display



```
input size: 131072
pagefault number is 8193

D:\Assignment3\Assignment3\x64\Debug\Assignment3.exe (进程 10136)已退出，返回代码为: 0。
若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...
```

program1: make