CSC3150_A3_report

119010108 HuangPengxiang

11/9/2021

Contents

Overview

This is the third Assignment for Operating System. In this assignment, The programming goal is to implement the the simulation of the paging methodology and LRU algorithm in memory management **using both single thread or multiple thread**. This simulation is implemented in CUDA. The global memory is treated as disk storage and shared memory is treated as the traditional CPU physical memory. This project focused on designing a paging system with swapping, which can calculate the final page fault number and print it out.

Some Declarations

- There might be some **Indentation** unaccustomedness in your computer. I used Mac VSC to develop the program and test it on CenterOS, and I found the Indentation is not comfortable to view when I open it on test computer. I am sorry for that and I hope it won't affect my grade.
- I design this program for every .bin test case **except for the test size is smaller than 32KB**. because when the size is smaller than 32KB, the invalid bit will still exist and will affect my program to swap when you try to read the memory from bottom to top.
- I finish the bonus part. I generate four thread using CUDA, and each thread will **have a pid number in the invert page table**. Each thread will execute the write and read concurrently, and the the priority of each thread will be Thread 0 > Thread 1 > Thread 2 > Thread 3, which is designed to avoid the race condition.

Environment

The running environment is showing below:

- The nvcc version:

```
(base) [cuhksz@TC-301-04 source]$ nvcc -V
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2019 NVIDIA Corporation
Built on Fri_Feb__8_19:08:17_PST_2019
Cuda compilation tools, release 10.1, V10.1.105
```

- The test environment:

```
(base) [cuhksz@TC-301-04 source]$ uname -r
3.10.0-957.27.2.el7.x86_64
```

Running Guidance

The tree of my program

```
.
├── bonus
│   ├── data.bin
│   ├── main.cu
│   ├── makefile
│   ├── snapshot.bin
│   ├── user_program.cu
│   ├── virtual_memory.cu
│   └── virtual_memory.h
├── report.pdf
└── source
    ├── data.bin
    ├── main.cu
    ├── makefile
    ├── snapshot.bin
    ├── user_program.cu
    ├── virtual_memory.cu
    └── virtual_memory.h

2 directories, 15 files
```

Detailed step to run my program

I design a makefile for you to run my program. Basically, you should follow the steps below to run it.

```
$ cd /* where the source located */
$ make        // there will be some warning about "char *" in main.o, just choose to ignore it
$ make test   // use this command to run
$ make clean  // use this command to clear
```

Demo output

The demo output(input size = 128KB) for source part:

Notice: I use command cmp to compare the data.bin and snapshot.bin

```
(base) [cuhksz@TC-301-04 source]$ make test
./Project
input size: 131072
write done
read done
pagefault number is 8193
(base) [cuhksz@TC-301-04 source]$ cmp data.bin snapshot.bin
(base) [cuhksz@TC-301-04 source]$ 
```

The demo output(input size = 128KB) for bonus part:

```
(base) [cuhksz@TC-301-04 bonus]$ make test
./Project
input size: 131072
write done
read done
write done
read done
write done
read done
write done
read done
pagefault number is 32772
(base) [cuhksz@TC-301-04 bonus]$ cmp data.bin  snapshot.bin
(base) [cuhksz@TC-301-04 bonus]$ 
```

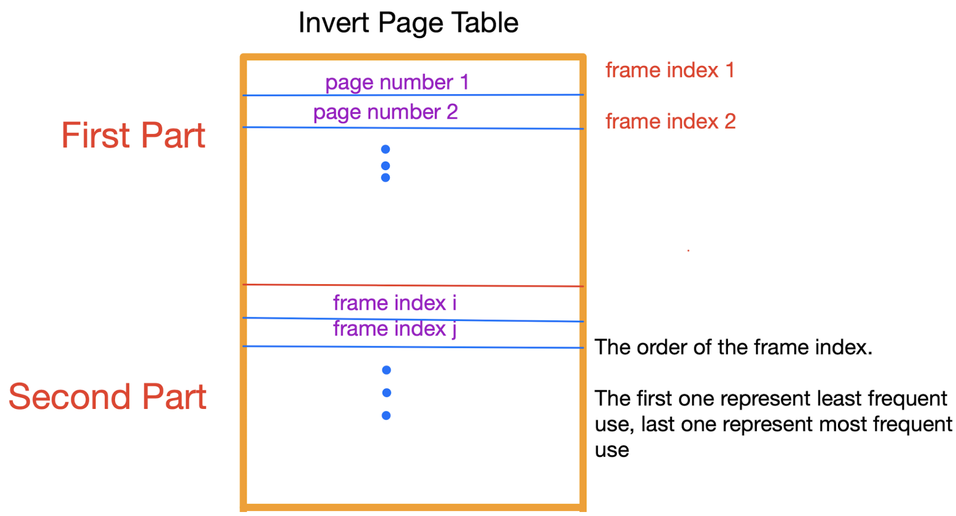
Program Design

Source design

The main work for this design is to construct the page table and LRU algorithm, and how to map the virtual memory to physical memory. I will show these part one by one.

How I design the inverted page table

The page table will be initialized as two parts. The first parts will be initialized as 0x80000000, which the MSB is 1 and represent it is invalid to use. The second part is initialized as index number, which I use it implement the LRU algorithm. The first index will represent the least frequent use and the last one will represent the most frequent use. The value in the index will be the frame number that in the first part. Every time write/read will update the invert page table, and update the order in the second part. If page fault occur, it will find the first index of frame number in second part which represent the least frequent use frame, then update it into a new page number. then read/write a value to swap into memory/disk, and I update the invert page table after I read/write.



How I design the LRU algorithm

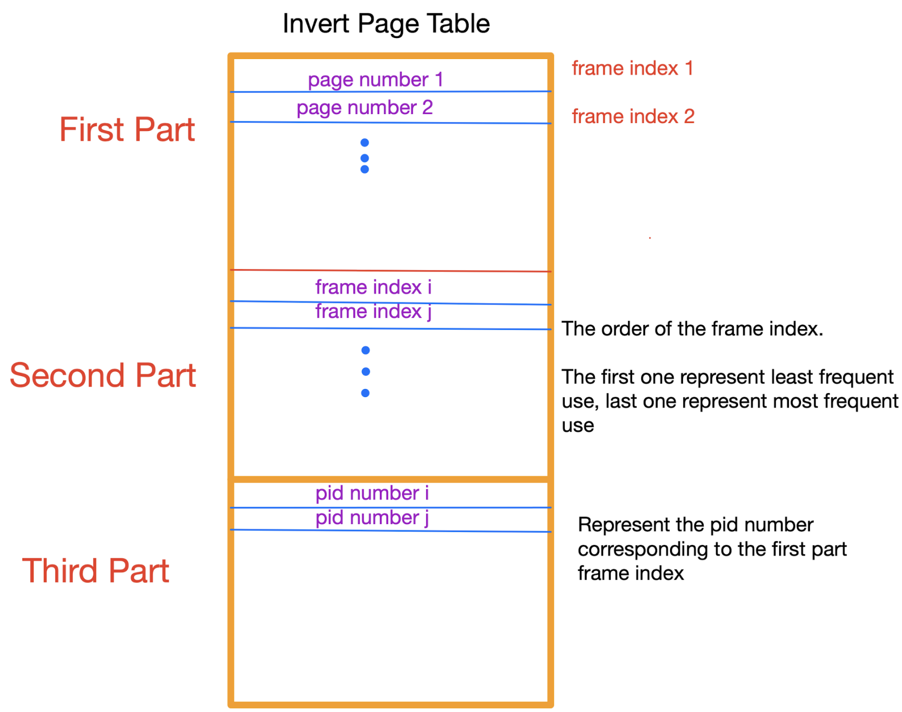
The LRU algorithm is basically always weed out the least frequent used page and replace its space. And the order in my second part of page table is represented as the least-most frequent use order. The first one is represented as least frequent use and the last one is represent the most frequent use. Every time I read/write a value into it, I will maintain this table, which is move the move the frame number after it one forward and move the current frame index to the last.

How I map the virtual meomory to physical memory

For write, First when The page table still has some invalid bit, then it will keep load page into memory untill it is full. If the memory is full, It will find the least frequent use page and weed it out. and load the new page into the table. **I map the program page to disk memory in linear way**. which means The map between program page and disk is linear. When it try to read the value and page fault occur, it will find the value from disk based on linear way and load it into memory. During this process, I will update and maintain my page table.

Bonus Design

The main work for bonus is to construct muti-thread to execute the single program. Each thread will execute the program one by one and the priority is Thread 0 > Thread 1 > Thread 2 > Thread 3. And after the previous thread finish the snap part, the next thread will begin to execute the program. And Also I add a new content into the invert page table, which is **Pid Num**. When one thread write the value into memory, the page table will update the corresponding pid number into it. when other thread execute, it will check whether the pid is corresponding to itself, if not, then page fault will occur, and swap the pid number to its own. The first part of page table is the same as the single thread one. And The memory management and map is also follow the same logic as the single thread one.



Program Implementation

Source Implementation

For `vm_write`, I first check If there occur page fault. if occur, then check whether the frame is full, if the frame is full, I move the page that least frequent use into storage and load the new page into the original frame. if page fault not occur, then there will be some page valid and exist, I found that frame number and load the value into the memory. Finally I update the table to maintain the LRU table.

```
__device__ void vm_write(VirtualMemory *vm, u32 addr, uchar value) {

    u32 offset = addr % 32;
    u32 page_num = addr / 32;
    u32 frame_num;

    if (ISPageFault(vm, page_num)) {
        /* Page fault occur, set the first index to the least recent index in table */
        frame_num = vm->invert_page_table[vm->PAGE_ENTRIES];

        /* check whether the table is full, if full, swap the page to the disk */
        ISFrameFull(vm, page_num, frame_num);

        /* set the page num to this frame */
        vm->invert_page_table[frame_num] = page_num;
    }
    else {
        /* no page fault occcur, find the index frame num to access the data */
        frame_num = FindFrameNum(vm, page_num);
    }

    vm->buffer[frame_num * 32 + offset] = value;

    /* Update the invert_page_table to maintain its order */
    UpdateTable(vm, frame_num);
}
```

For `vm_read` and `vm_snap`, I first check if the page fault occur. if occur, then I swap the memory to the disk and load the needed page into the frame. if not occur, then I find the frame number that contain this page. and load the value into result buffer. Finally update the page table.

```
__device__ uchar vm_read(VirtualMemory *vm, u32 addr) {

    u32 offset = addr % 32;
    u32 page_num = addr / 32;
    u32 frame_num;

    /* Page fault occur, set the first index to the least recent use and replace it */
    if (ISPageFault(vm,page_num)) {
        frame_num = vm->invert_page_table[vm->PAGE_ENTRIES];

        /* load the needed value to memory */
        SwapToMemory(vm, frame_num, page_num);
    }
    else {
        frame_num = FindFrameNum(vm, page_num);
    }

    /* find the frame num and return its value in corresponding meory */
    u32 newframe_num = FindFrameNum(vm, page_num);
    uchar ReturnValue = vm->buffer[newframe_num * 32 + offset];

    /* Update the table */
    UpdateTable(vm, frame_num);

    return ReturnValue;
}
```

Bonus Implementation

I first initialize the four threads, and let each thread concurrently execute the program. and I add a `_syncthreads` function to let other thread wait the unique thread execute the program. And I also add content to update the page table by changing the pid number.

```
    // user program the access pattern for testing paging
    for (int current = 0; current < Thread_Num; current++){
      if (current == rank){
        user_program(&vm, input, results, input_size);
        current_thread = rank + 1;
      }
      __syncthreads();
    }
}
```

Page Fault Calculation

For Input size = 128KB in single thread

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.

The page faults in write is 4096 since every page loaded from the input buffer is a new page for the physical memory. There are totally 128K data with 32bytes in each page, we can have 4096 pages. Therefore, it will generate 4096 page faults.

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.

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.

For Input size = 128KB in four threads

Since there are four threads to execute, and each thread has unique pid, every time change a new thread pid will generate a page fault. then the total page fault in four threads should be

$$FourThread = 4 * SingelThread$$

, which is 32272.

The Problem I have met

- Significant amount of time is spent on page table design understanding and virtual memory principle interpreting. I was not fully understand about the page table design, and How to design LRU algorithm. In the ordinary page table, each process holds its own page table. I was also unclear about the definition of frames and pages. Therefore, in the beginning, I had a hard time understanding what are the data storing in the page table entries, what should be swapped out to the storage during page fault, how should the program know the location of pages in the storage, how to search the desired page in the page table, where does the page in the page table stores. Finally, I figured out all questions above and ended up with a much more thorough understanding on the paging methodology of operating system.
- The second Problem is How to deal with the page fault. I was stuck with how to deal with the page fault at the beginning and how to map the disk memory to the physical memory. Everytime I output the snapshot.bin file and The page fault is correct but the content in it is differ from data.bin. It stucks me a lot of time. And finally I found that there is a significant error during my read section. I swap the wrong memory and it also occur the page fault, but the content I swap is wrong

```
input size: 131072
write done
read done
55 46 22 55 89 51 57 28 88 12 44 16 49 41 36 96 69 90 45 28 79 29 12 51 48 63 45 97 61 23 68 93 94 46 22 55 89 51 57 28
88 12 44 16 49 41 36 96 69 90 45 28 79 29 12 51 48 63 45 97 61 23 68 93 98 46 22 55 89 51 57 28 88 12 44 16 49 41 36 96
69 90 45 28 79 29 12 51 48 63 45 97 61 23 68 93 61 46 22 55 89 51 57 28 88 12 44 16 49 41 36 96 69 90 45 28 79 29 12 51
48 63 45 97 61 23 68 93 30 46 22 55 89 51 57 28 88 12 44 16 49 41 36 96 69 90 45 28 79 29 12 51 48 63 45 97 61 23 68 93
92 46 22 55 89 51 57 28 88 12 44 16 49 41 36 96 69 90 45 28 79 29 12 51 48 63 45 97 61 23 68 93 83 46 22 55 89 51 57 28
88 12 44 16 49 41 36 96 69 90 45 28 79 29 12 51 48 63 45 97 61 23 68 93 40 46 22 55 89 51 57 28 88 12 44 16 49 41 36 96
69 90 45 28 79 29 12 51 48 63 45 97 61 23 68 93 63 46 22 55 89 51 57 28 88 12 44 16 49 41 36 96 69 90 45 28 79 29 12 51
```

- The third problem is the one I didn't sloved. When the input file is smaller than 32kb, when the program execute the read section, It will read the invalid value, and can not found the corresponding value in the disk. and the the disk value have no page number, then my program will terminated. **It even can not finish the read part**

```
(base) [cuhksz@TC-301-04 source]$ make test
./Project
input size: 2188
write done
pagefault number is 70
(base) [cuhksz@TC-301-04 source]$
```

What I Learned From This Project

- The paging mechanism is to divide the program or data into different pages and load pages into both shared memory and global memory. 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 When we try to read the data,
- 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.
- There can be multiple thread to execute the same program by using CUDA. all we do is need to synchronization, and let each thread concurrently execute it. we also need to design a new content for the page table to represent the pid.

Other Test For This Project

Input size is 38265 For single thread:

```
(base) [cuhksz@TC-301-04 source]$ make test
./Project
input size: 38265
write done
read done
pagefault number is 1540
(base) [cuhksz@TC-301-04 source]$ 
```

Input size is 38265 For Four threads:

```
(base) [cuhksz@TC-301-04 bonus]$ make test
./Project
input size: 38265
write done
read done
write done
read done
write done
read done
write done
read done
pagefault number is 6160
(base) [cuhksz@TC-301-04 bonus]$ 
```

Input size is 62750 for single thread:

```
(base) [cuhksz@TC-301-04 source]$ make test
./Project
input size: 62750
write done
read done
pagefault number is 3835
(base) [cuhksz@TC-301-04 source]$ cmp data.bin snapshot.bin
(base) [cuhksz@TC-301-04 source]$ 
```

Input size is 62750 for four threads:

```
(base) [cuhksz@TC-301-04 bonus]$ make test
./Project
input size: 62750
write done
read done
write done
read done
write done
read done
write done
read done
pagefault number is 15340
(base) [cuhksz@TC-301-04 bonus]$ 
```