CSC3150 Assignment 2 Report

119010177 Muhan Lin

I. Introduction

In this project, a mechanism of virtual memory is implemented via GPU's memory with single one thread or four concurrent threads. Two level memory, 32K shared memory and 128K global memory, have been simulated. An 16K inverted page table has been implemented to transfer virtual address to physical address. The writing policy is writing back and writing allocated. When swapping memory, LRU paging algorithm is adopted. The input of the program is a data file, whose data is written to input buffer by CPU. User should also offer a `user_program.cu` file to specify the read and writing operations. The output is snapshot.bin, to which CPU load data from GPU. The program also prints the input size and page fault number on the screen.

II. Design and Solution to Problems

1. Memory Map

In this project, the virtual memory size is the same as the disk storage size. Therefore, the virtual address maps directly to the disk address. That is to say, physical address is only applied to physical memory (shared memory).
The page table is only responsible for virtual and physical address transformation of pages in physical memory. As the page size is 32 bytes and page table entry is 32-bit wide, only 1K entries, the maximum number of physical pages, of page table is used, although it is initiated with 4K entries.

2. Inverted Page Table

Page table is responsible for recording information of all pages in physical memory and transforming virtual and physical address. Inverted page table records the virtual base addresses of pages, **vpn**, in its entries, with the corresponding index, **ppn**, serving as the corresponding physical base addresses. The transformation from virtual to physical address thus should be:

$physicalAddr = ppn * PAGESIZE + (virtualAddr \% PAGESIZE)$

Each page table entry is divided into the following parts:
(Count bits will be mentioned in latter parts.)

index	content
31	valid (1 for invalid)
30	dirty
29~28	tid
27~12	count
11~0	vpn

Valid bit is 1 if the corresponding page is not in physical memory. Dirty bit is set when the data in physical memory is inconsistent with that in disk. `tid` records the id of the thread owning this page.
Because the page size is 2^5 bytes, the last 5 bits of virtual address will not be stored into page table. In addition, the virtual memory size is 2^{17} bytes, so the number of effective bits in an address is only 17. As a result, the 16th to 5th bits of an address are **vpn**, the bits of virtual address which are needed to be stored into page table. Each **vpn** stands for an unique page.
Several functions have been implemented with bitwise operations to read each part of page table entry or generate a page table entry with given information:

```
__device__ u32 validBit(u32 ptItem);
__device__ u32 dirtyBit(u32 ptItem);
__device__ u32 count(u32 ptItem);
__device__ u32 getPtItem(u32 v, u32 dirty, u32 tid, u32 count, u32 vpn);
__device__ u32 getTid(u32 ptItem);
```

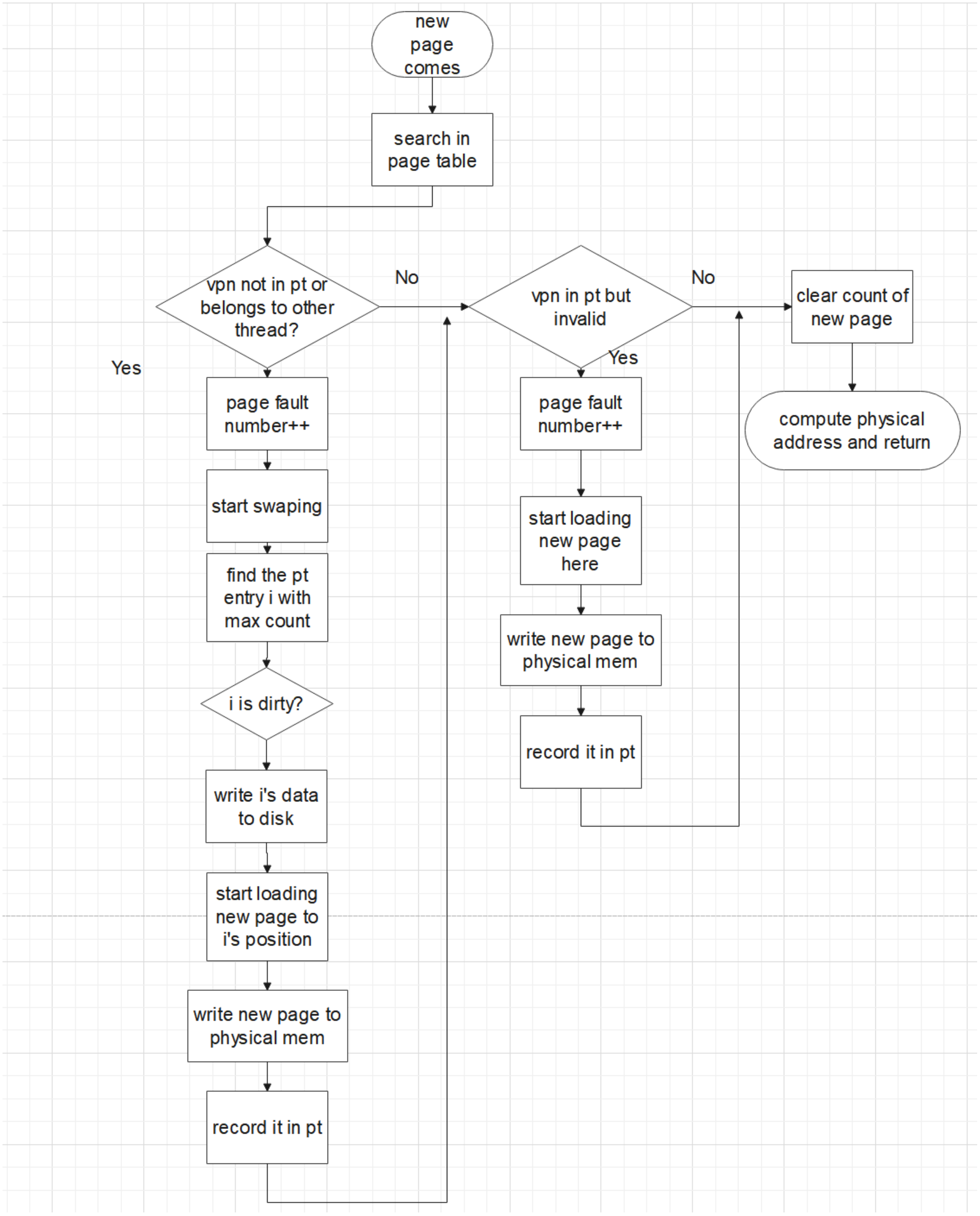
3. Paging Algorithm

Because the speed of physical memory is much faster than that of disk, processor prefer to access memory in physical memory. Each time a page is accessed, processor looks for it in page table first. If it cannot find the correct valid page, page fault is triggered. In other words, this page is not in physical memory.
In this case, if the vpn is actually in page table but with invalid bit, load the page directly into its original position and revise valid bit to 0 with `load` function. If no page table entry contains this vpn or the entry containing this vpn is of other threads, processor will find a victim page in page table, write victim's data back to disk if it is dirty, swap out this victim, load the correct page from disk into physical memory and record the new page in the page table with `swap` function.
LRU strategy has been adopted when choosing victim page. When swapping, the least recently used page in page table will be swapped out from page table and physical memory. Its position will be taken up by the new page. To implement LRU, 16 bits of `count` in page table entry record the time length for which this page has been in page table. At the moment when a page is loaded into page table and physical memory, its count bits are 0. Count bits increases by one when other page in page table is accessed, and are cleared to 0 when this page is read or written. When swapping, the page with largest `count` will be swapped out.
What is worth notice is that the swap operation is correct whether the page table is full or not. Invalid page entries only exist at the beginning of memory access. Because processor cannot remove pages from page table and then invalidate this entry when page table is not full, the number of pages in page table is nondecreasing. After page table becomes full, once after a page is swapped out, a new page comes into page table and clears valid bit to 0, so no invalid page table entry appear once after the page table becomes full. Because all entries are initiated to be invalid, invalid entries are always the least recently used entry in page table. If page table is not full when `swap` is called, the page still must be swapped into an entry with invalid page, ie. empty entry. If page table is full, the new page will replace the LRU page correctly. Thus the swap operation is correct in either situation.
Only after the page is in page table, processor can read or write. When writing, data is only written to physical memory, and dirty bit of this page is set to 1.

This algorithm is implemented in function

```
__device__ u32 Virtual2Physical(u32 virtualAddr);
```

The logic flow chart of it is as follows.


new page comes
search in page table
vpn not in pt or belongs to other thread?
No
Yes
page fault number++
start swaping
find the pt entry i with max count
i is dirty?
write i's data to disk
start loading new page to i's position
write new page to physical mem
record it in pt
vpn in pt but invalid
No
Yes
page fault number++
start loading new page here
write new page to physical mem
record it in pt
clear count of new page
compute physical address and return

4. Read

First increase the count of all items by 1. Then get the corresponding physical address by calling `Virtual2Physical`. Fetch the data from vm buffer with the physical address.

5. Write

First increase the count of all items by 1. Then get the corresponding physical address by calling `Virtual2Physical`. Write the data into the physical address. Set the dirty bit of this page in page table to be 1.

6. Snapshot

With `vm_read`, load the `input_size` pieces of data on the addresses starting from offset into output buffer byte by byte.

7. Multi-thread

This part is realized in bonus part. As the priority order is thread 0 > thread 1 > thread 2 > thread 3, four threads execute the user program one by one. A thread cannot start user program until all threads with priority higher than its priority finish user program.
2 bits are used to record thread id in page table, as there are $2^2$ threads.
Paging operations are thus realized by each thread in order. The paging algorithm stated above has considered the multi-thread case, so it will not be stated again here.

a. Thread Creation

Assign 1 and 4 to the first and second attributes of `mykernel`. Then 4 threads will be created. Although the initiation of `vm` is inside `mykernel` and thus will be called by four threads, the data in `vm` are shared by four threads and no data inconsistency will occur, because the arguments passed to `vm_init` are all global.

b. Thread Scheduling

A for loop from 0 to 3 is added in `mykernel` to schedule threads after all initiation work is completed. In the ith loop, if the thread's id is i, this thread executes user program. Otherwise, the thread does nothing and wait until the correct thread finishes the user program. Then 4 threads enter the next loop simultaneously.
Attribute `threadIdx.x` is used to identify a thread, and function `__syncthreads()` can make threads wait until all 4 threads reach this point.

III. Environment

1. Linux Version

```
-bash-4.2$ cat /etc/redhat-release
CentOS Linux release 7.6.1810 (Core)
```

2. CUDA Version

```
-bash-4.2$ 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
```

3. GPU Version

```
-bash-4.2$ lspci | grep -i vga
00:02.0 VGA compatible controller: Intel Corporation UHD Graphics 630 (Desktop)
01:00.0 VGA compatible controller: NVIDIA Corporation GP106 [GeForce GTX 1060 6GB]
(rev a1)
```

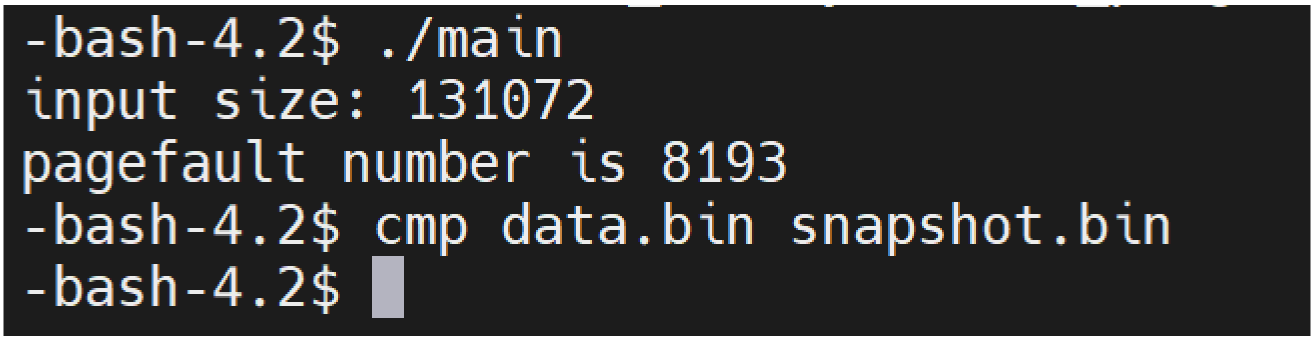
IV. Execution Steps

```
-bash-4.2$ cd Assignment_3_119010177/Source
-bash-4.2$ make
-bash-4.2$ ./main
```

Bonus:

```
-bash-4.2$ cd Assignment_3_119010177/Bonus
-bash-4.2$ make
-bash-4.2$ ./main
```

V. Outputs

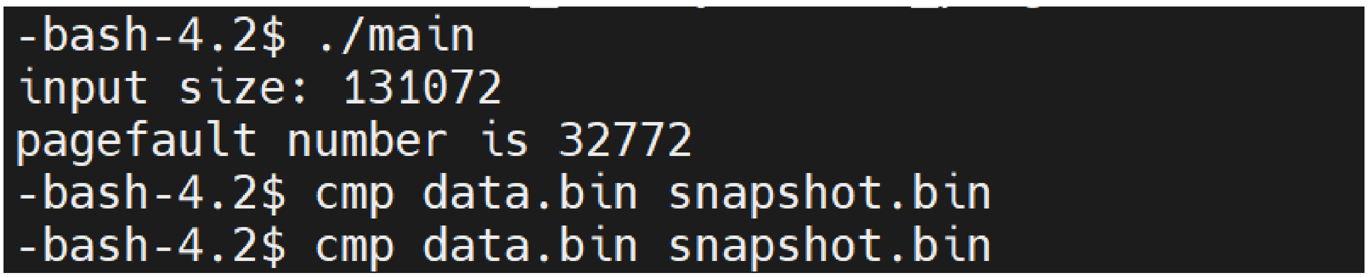


```
-bash-4.2$ ./main
input size: 131072
pagefault number is 8193
-bash-4.2$ cmp data.bin snapshot.bin
-bash-4.2$
```

The first step of the given user program is to write all content in input buffer to disk in order. The input size is 131072 bytes, which is 10244 *pages. The physical memory can only contain up to 1024 pages. According to the implementation, processor will load pages one by one from physical index 0 to 1023. When page table becomes full, processor swaps least recently used page out and loads. The swap process occurs from physical index 0 to 1023, and to 0 again, so and so forth. Therefore, page faults occur in the writing process is $10244*=4096$. Finally pages at virtual index 3072 to 4095 is in physical memory.
Then user program read pages from virtual index 4095 to 3071. The first 1023 pages will not cause page

faults. Only one page faults occur when accessing to page at virtual index 3071, so the first entry of page table is swapped out. In the process of snapshoting, the process similar to writing is repeated again, because all of the first 1024 pages are not in the physical memory. This contributes to 4096 page faults.
Therefore, the total page fault number is 8193.

bonus



```
-bash-4.2$ ./main
input size: 131072
pagefault number is 32772
-bash-4.2$ cmp data.bin snapshot.bin
-bash-4.2$ cmp data.bin snapshot.bin
```

Because a page table entry not belonging to one thread will be considered to have no relationship with given vpn, even if this entry actually has the same vpn, the latter thread will repeat the whole process of the former thread. So the total page fault number is $8193*4=32772$.

VI. Learning Points

This project helps me practice paging management. In this process, I gain much more insights and get more familiar with memory level and paging mechanism under single or multiple threads.