CSC3150 Operating Systems

Project 3 report

Name: Xiao Nan

Student ID: 119010344

Date: 2021/11/10

The Chinese University of Hong Kong, Shen Zhen

0. Introduction:

In homework 3 basic part I implemented the paging memory system using a 16KB inverted page table for accessing the data stored in the 32KB shared memory and 128KB global memory (simulated as main memory and disk storage in computer architecture) on NVIDIA CUDA GPU cards. In the bonus part I implemented another page table supporting concurrent executions of threads on GPU.

1. Design of the program:

The homework 3 basic part consists of 2 sub-modules: virtual memory module (virtual_memory.h, virtual_memory.cu) and user program module (user_program.cu).

The user program module essentially simulates a user program executing using the memory provided in the virtual memory module. It first writes data into the virtual memory from the input data block in the global memory and then read the data written previously (both with page replacement policy set as LRU by default).

The virtual memory module implements vm_write and vm_read function. Vm_write is used for writing data into the backend storage. It first looks up the page table (16KB in the shared memory region) to check if the data can be written to main memory for convenience by checking whether the virtual page number would match. If matching the data would be written to the main memory location specified by the index of the page entry. If not matching or accessing free frame the corresponding block of data would be fetched from global memory to the main memory (32KB shared memory region) and then the input data would be written in the main memory. In the latter case if the main memory is full, the memory system would select an entry in page table using LRU policy, push the page of data back to the global memory, pull the target page of data in the global memory into the 32KB shared memory, and then update the page table with the new page and the new count. Vm_read is used for reading data from the backing store. It also first searches through the page table to try matching virtual page number. If matching, the data in the 32KB shared memory would be returned directly. If not matching or accessing free frame, the data would be searched in the global memory using the address provided and then be written in the main memory. In the latter case if the main memory is full the memory system would search through the page table to find the LRU page, push that back to the global memory, and then pull up the page being read in the global memory.

The LRU policy is implemented by 4-byte integer count (in shared memory) and utilizing the first several bytes of the count. The 4-byte count is stored in inverted_page_table[3*PAGE_SIZE], which increments every time a read/write operation takes place. The first word of each page table entry is divided into 2 parts: Valid bit part and count part. The valid bit part, the most significant byte of the entry, indicates whether the entry is currently being used. The count part, the least significant 7 bytes, counts the degree to which the entry is being recently used. After every read/write operation, the incremented count would be updated to the entry in the page table which had just been read or written, so that whenever LRU comparisons are invoked, the page entry which has the minimum count would be the entry to replace.

In the bonus part I added the process id to each page table entry to identify which process is using the current entry. I scheduled the 4 processes concurrently by using __syncthreads() and conditional judgements on threadIdx.x to synchronize the execution of the 4 threads (one executing after another). For LRU replacement and free frame filling I also enabled recording process id information.

2. Runtime environment:

2.1. Linux version:

```
[cuhksz@TC-301-37 xxx]$ cat /proc/version
Linux version 3.10.0-957.21.3.el7.x86_64 (mockbuild@kbuilder.bsys.centos.org) (gcc version 4.8.5 20150623 (Red Hat 4.8.5-36) (GCC) ) #1 SMP Tue Jun 18 16:35:19 UTC 2019
```

2.2. CUDA version:

```
[cuhksz@TC-301-37 xxx]$ nvcc --version
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
```

2.3. GPU information:

```
[cuhksz@TC-301-37 xxx]$ lshw -C display
WARNING: you should run this program as super-user.
  *-display
       description: VGA compatible controller
       product: GP106 [GeForce GTX 1060 6GB]
       vendor: NVIDIA Corporation
       physical id: 0
       bus info: pci@0000:01:00.0
       version: a1
       width: 64 bits
       clock: 33MHz
       capabilities: vga_controller bus_master cap_list rom
```

```
          configuration: driver=nvidia latency=0
          resources: irq:142 memory:f0000000-
   f0ffffff memory:c0000000-cfffffff memory:d0000000-
   d1ffffff ioport:3000(size=128) memory:f1080000-f10fffff
     *-display
          description: VGA compatible controller
          product: UHD Graphics 630 (Desktop)
          vendor: Intel Corporation
          physical id: 2
          bus info: pci@0000:00:02.0
          version: 00
          width: 64 bits
          clock: 33MHz
          capabilities: vga_controller bus_master cap_list rom
          configuration: driver=i915 latency=0
          resources: iomemory:400-3ff irq:140 memory:4000000000-
   4000ffffff memory:e0000000-efffffff ioport:4000(size=64)
WARNING: output may be incomplete or inaccurate, you should ru
   n this program as super-user.
```

3. Steps to execute my program:

In command line, you need to first unzip the .zip file. Make sure that the .zip file is under the current directory:

```
[cuhksz@TC-301-37 xxx]$ unzip Assignment_3_119010344.zip
```

3.1 Normal part:

“cd” to the “Source” execution directory, and type “make hw2” to compile the source code. Make sure that the main.cu, virtual_memory.cu, virtual_memory.h, user_program.cu are under the current directory:

```
[cuhksz@TC-301-37 xxx]$ cd Assignment_3_119010344/Source/
[cuhksz@TC-301-37 Source]$ make
```

Then, type “./main.out” to execute the program.

```
[cuhksz@TC-301-37 Source]$ ./main.out
```

3.2 Bonus program:

"cd" to the "Bonus" execution directory. Type "make" to compile the bonus programs. Make sure that the main.cu, virtual_memory.cu, virtual_memory.h, user_program.cu are under the current directory:

```
[cuhksz@TC-301-37 xxx]$ cd Assignment_3_119010344/Bonus/
[cuhksz@TC-301-37 Source]$ make
```

Then execute the bonus program:

```
[cuhksz@TC-301-37 Source]$ ./main.out
```

4. Page fault number and reasons:

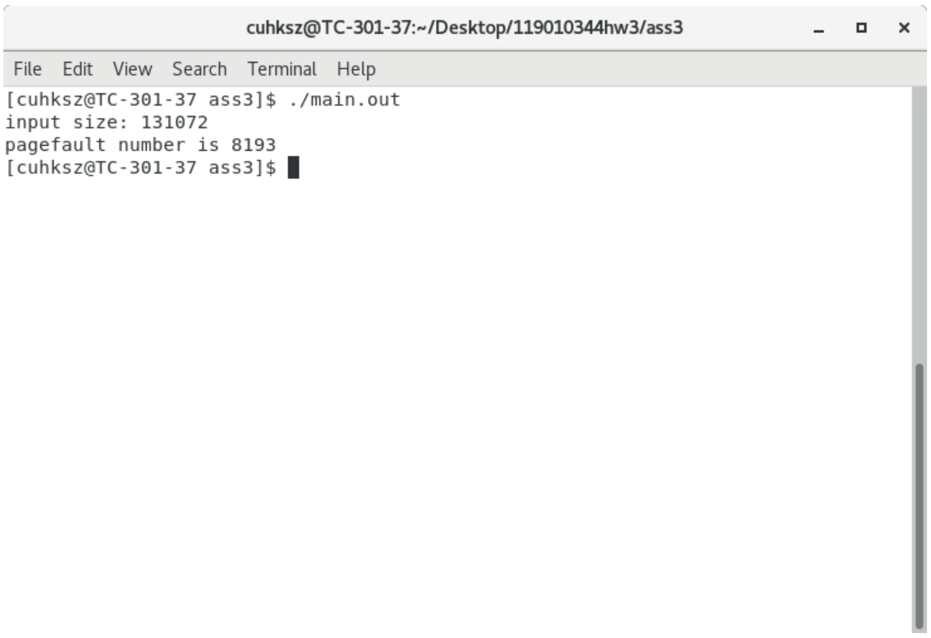
The page fault number for the normal program is 8193. The reason is that each stage of the program, first writing, median reading and final reading contribute to 4096, 1 and 4096 page faults respectively. The first writing part of the program (writing 128KB data to the global memory) would write 4 groups of 32KB data, each corresponding to 1024 pages and hence causing 1024 page faults (since the main memory size is limited to 1024 pages with each page 32B) and in total causing 4096 page faults. In the median reading part bytes from 131072 to 98304 are read with page hits, but byte 98303 is read with page fault replacing the last page (page 1023), contributing to 1 page fault. The final reading part reads from the beginning of the global memory to the end, which generates 4096 page faults since at this time the shared memory doesn't contain data in the first 32KB of the global memory (instead containing approximately the last 32KB of that).

The bonus program turns out to generate 32772 page faults. The execution of the first program contributes to 8193 page faults (the same as the normal program). The second time since the thread id is different, all pages from the previous thread would be invalid, hence the new thread would replace all the entries in the page table belonging to the previous thread that are least recently used even though the same writing/reading address is detected. Similar procedures would take place for the later 3 threads executing concurrently, resulting in 8193*4=32772 page faults in total.

5. Program output screenshots:

5.1 Normal program:

The results for the page fault number are shown in the figure 5.1:



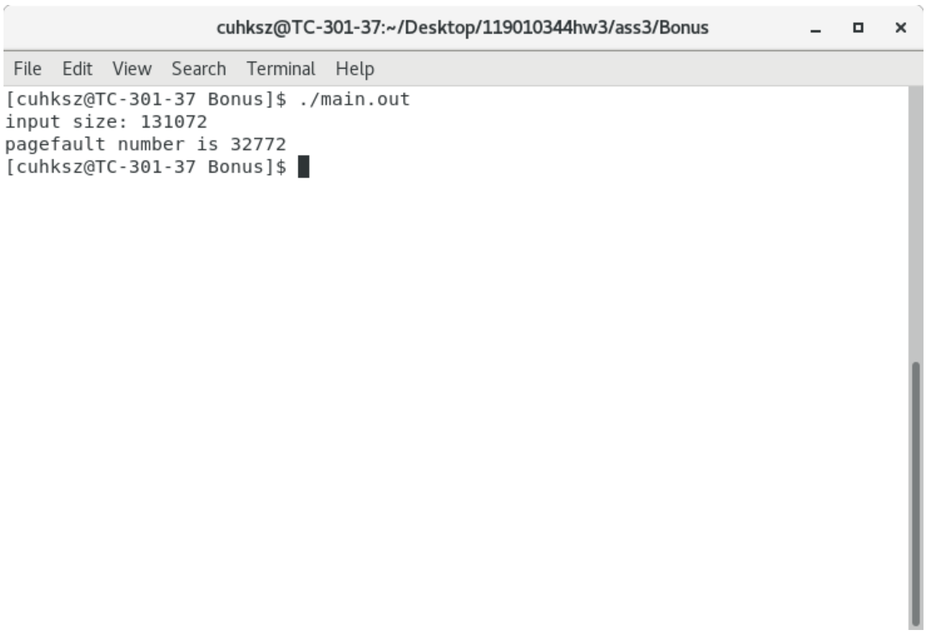
cuhksz@TC-301-37:~/Desktop/119010344hw3/ass3
File Edit View Search Terminal Help

```
[cuhksz@TC-301-37 ass3]$ ./main.out
input size: 131072
pagefault number is 8193
[cuhksz@TC-301-37 ass3]$ 
```

Figure 5.1: Page fault number for normal program

5.2 Bonus:

The results for the page fault number for the bonus program are shown in figure 5.2:


cuhksz@TC-301-37:~/Desktop/119010344hw3/ass3/Bonus
File Edit View Search Terminal Help

```
[cuhksz@TC-301-37 Bonus]$ ./main.out
input size: 131072
pagefault number is 32772
[cuhksz@TC-301-37 Bonus]$
```

Figure 5.2: Page fault number for bonus program

6. Problems that I've met and how I solved them:

In this project the first problem that I came across is that the execution time of the page reading/writing is too long. Since previously for every page table entry I maintain a count which continuously changes as the number of read/writes increase. Then I found that its too time-consuming since in such case each read/write, regardless of whether or not the page should be replaced, would require reading through the page table to update the count in every entry. Then I decided to use a global 4-byte integer count dynamically incrementing after read/write, which reduces the count updating task overhead a lot when reading/writing with replacement and hence saves a lot of time.

In addition, when debugging the program I came across a problem that took me a lot of time to find out: the bits for storing the count of each entry (representing LRU) wasn't enough. Previously I used only the least significant 20 bits of the first word of each entry for storing counts. Then in the bonus program I find it not enough since the maximum possible count (reflected by the number of read/writes required to be done) exceeds the range of a 20-bit number (since the bonus program requires more than 2^20 read/write operations, indicating the count number that might be achieved). Then I solved it by assigning more bits (28 bits) for the count.

7. Things I've learnt from the project:

In this project I learnt deeply how the page table works with the main memory and the disk and how powerful the LRU page replacement policy can be in managing data read or write. I also learnt deeply the structure of the page table and how it might change as the practical needs arise (e.g. the needs of multi-threaded applications). In addition, I understood the basic memory hierarchy of NVIDIA CUDA GPU (the differences between global and shared memory) and learnt how GPU would schedule threads in a block inside a grid. I also learnt how to synchronize the execution of threads in CUDA GPU when running multi-threaded applications.