The Chinese University of Hong Kong, Shenzhen

CSC 3150

Operating System

Report for Project 3

Author:
Qiu Weilun

Student Number:
120090771

November 7, 2022

Contents

1 Environment

The entire program is compiled and tested on the cluster provided. Therefore, the working environment would be the cluster environment.

1. **Operating System Version**

   CentOS Linux release 7.5.1804

2. **GPU Information**

   Nvidia Quadro RTX 4000

3. **CUDA Version**

   11.7

4. **CPU Information**

   Intel(R) Xeon(R) Silver 4210R CPU @ 2.40GHz

   20 Cores, 40 Threads

5. **Memory**

   100GB RAM

2 Execution

The program is executed on the cluster. There is a `slurm.sh` file in both the `source` and `bonus` files. The file contains the instruction to compile and run the `main.cu` file. To execute the program, please first apply the following command to get to the correct directory. The following example illustrate the case of entering the `source` directory.

```
[120090771@node21 Assignment_3_120090771]$ cd source
[120090771@node21 source]$
```

After that, the program can be executed by using the following command.

```
[120090771@node21 source]$ sbatch ./slurm.sh
Submitted batch job 17747
```

The above information indicates that the job has been submitted to the cluster and the corresponding compilation information as well as the final results will be writen to the `output.out` file.

The same procedure can be applied to execute the program in bonus part.

3 Design

The main task in project 3 is to simulate the virtual memory in an operating system. Basic components which are needed to be simulated in the project includes **virtual memory**, **physical memory**, **secondary memory**, **page table** and an additional **swap table**. The two significant machnisms that is required to implement are the **paging** and **swapping** process. According to the requirement, the size of the virtual memory is 160 KB. The size of the physical memory is 48 KB (32 KB for data access and 16 KB for page table setting). The size of the secondary memory is 128 KB.

3.1 Virtual Memory

Virtual memory is an important part of the memory management system. The virtual memory seperates the logical memory and the physical memory. It provides the users with more convenience than the system without it by temporarily transferring data from the physical memory to the secondary memory. The machnism enables the users to use the computer without worrying the amount of memory available in the physical memory. In this project, the maximum size of the physical memory for data access would be 160 KB, which is exactly the sum of the capacity of physical memory (32 KB) and the secondary memory (128 KB). Since virtual memory is an abstraction of the logical memory which is not ”real” memory, it would not be implemented explicitly in the project as an array or other data structures.

3.2 Paging

Since virtual memory is not “real”, a machanism is needed to match the virtual memory with the physical memory. A page is usually refered to as the unit of data tranferring in memory management system. The paging machanism provides a way of matching virtual memory and physical memory that could resolve the problem of **external fragmentation**. In paging, the physical memory would be partitioned into fix-sized blocks called “frames”. The logical memory (virtual memory) would be partitioned into same sized blocks, which is called “pages”. In this way, pages could be matched to the frames in the physical memory. The matching can be implemented via **page table**.

In classical page table, the page numbers would be the indices (or keys) of the table and the frame numbers are the values in the table. In this project, such page structure is not applied. Instead, an **inverted page table** is applied. In an inverted page table, frame numbers are the indices and the page numbers are the values in the table. In more general case, an inverted page table would also include the information of different processes, such as the process ID. However, in the first part of this project, there is only one process in the memory management system. Therefore, it is no need to maintain this information. In the bonus part, there are 4 threads launched in the main routine, and thus, the information of process id would be recorded in the table.

In detailed implementation, the total number of page entries would be the same as the number of frame number. The page size is 32 bytes. The total capacity of the physical memory size would be 32 KB. Therefore, the number of pages would be $\frac{32\times10^{10}}{32} = 1024$. According to the coding template, the array which is used to simulate the behavior of inverted page table is initialized as:

```
extern __shared__ u32 pt[];
```

Each entry in the array occupies 32 bits (4 bytes). The size of the entire table is set as 16 KB. Therefore, total number of entries that the table could have would be $\frac{16\times10^{10}}{4} = 4096$, which is 4 times as large as the required 1024. Therefore, in the detailed implementation, the empty 3072 entries would be used for other usege. For example, the first 1024 entries are used to store the control bit (such as the valid/invalid bit). The information stored in the array `pt[]` is shown in Figure 1.

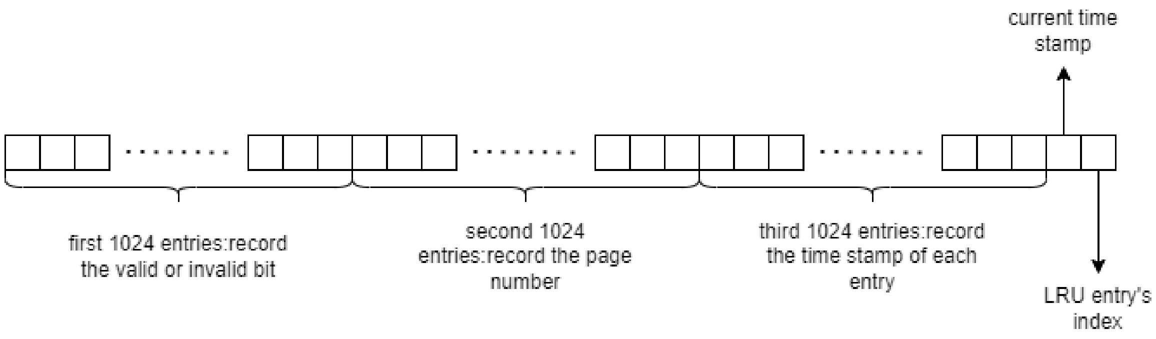

current time stamp
first 1024 entries:record the valid or invalid bit
second 1024 entries:record the page number
third 1024 entries:record the time stamp of each entry
LRU entry's index

Figure 1: implementation of inverted page table

The second 1024 entries store the information of the one-to-one mapping between the page number and the frame number. According to the illustration in Figure 1, assume that the frame number is i with constraint $0 \leq i \leq 1023$, the corresponding page number would be stored in position $i+1024$ in the array. The next 1026 entries would be used for the implementation of the least recently used algorithm (LRU) which is applied as the page replacement strategy, which would be discussed in the next section.

3.3 Page Fault & Latest Recently Used Algorithm

According to the requirement of this assignment, the least recently used algorithm is selected as the page replacement strategy of the memory management system.

Page replacement occurs when all the frames in the physical memory are filled with elements and the accessed page is not in the page table. When conducting page replacement, the program will first look for an empty position in the secondary memory. Then, the least recently used frame in the physical memory would be swapped out to the empty position, leaving an empty position in the physical memory. Finally, the program would look for where the original accessed page is in the secondary memory (since the page is not in the physical memory) and swap in the page to the physical memory.

In this project, the entire procedure is implemented by recording the time stamp of each frame. The general rule is that when a frame is read or write, the time stamp of the frame will be updated to the most recently used time stamp. When page fault occurs and page replacement is needed, the program called a sub-routine `find_LRU(&vm)` to find the frame's index whose time stamp is the smallest. The page corresponds to that frame would be the page to be swapped out. The current time stamp value as well as the current least recently used frame number would also be updated every time when a `vm_read` or `vm_write` operation is performed. They would be stored in the last two entries in the array as illustrated in Figure 1.

After selecting the least recently used page, the next step would be swapping out it and swapping in the accessed page. Detailed design is discussed in the next section.

3.4 Swapping

Swapping is an operation between the physical memory and the secondary memory. The least recently used page in the main memory would be moved to a backing store. In addition, the page which cannot be found in the physical memory would be moved from secondary memory to physical memory.

To implement swapping, a new array is needed to store the information of the mapping between entries in the secondary memory and the corresponding virtual page number (vpn). Therefore, a swap table is used in this project. The index of the table would be the index for the entries in the secondary memory. The secondary memory size is 128 KB according to the requirement. Since the page size is set as 32 byte in this project, the total number of entries in the swap table would be $\frac{128\times10^{10}}{32} = 4096$. The values in the table are the correponding vpn.

In the initilization stage of the swap table, all the values in the table are set to

6000. The reason is that this project support a total size of 160 KB virtual memory and total number of pages would be $\frac{160\times10^{10}}{32} = 5120$. Therefore, every page number would not exceed 5120. It is reasonable to set the values of the empty enties to be 6000. From section 3.3, when page fault occurs and the page table is already full, the program would search through the swap table to see whether there is entry with a value of 6000. If yes, the program would load the data in the least recently used page to the entry it has found in swap table. After that, the program search through the swap table and looks for the entry with value equals the accessed page number. Once the entry is found, the data in it will be loaded to the physical memory and the page table would also be updated since page in the frame has already changed.

Unlike the page table which is stored in the main (physical) memory, the swap table is usually stored in the disk. Therefore, the swap table in this project would also locate on the disk alone without occupying the secondary memory. In actual implmentation, it would be allocate to the global memory in CUDA GPU, and is initialized using keywords `__device__` and `__managed__`.

3.5 Design of Read and Write Function

In the following section the detailed procedure of functions `vm_read()`, `vm_write()` and `vm_snapshot()` will be discussed.

The `vm_write()` function generally simulate the process of writing to the memory. The input of the function consists of three parameters, `VirtualMemory *vm`, `u32 addr` and `uchar value`. The parameter `vm` consists of all the information of the current state of virtual memory. The input `addr` would be the access logical address. The variable `value` is the value needed to be written to the memory.

The function would first calculate the page number according to the given address. Using the result, the program would search in the inverted page table to look for the frame index which contains the page. If the frame is found, the program will then check the valid/invalid bit. If the bit is set as invalid, the frame is mapped to the page, but the frame in physical memory is empty. This situation occurs when no data is stored in the frame. The program will then check the swap table to see whether the page is in the secondary memory and load the data to the physical memory. If it is not found in the secondary memory, that means the page does not contain any data and the program would directly write to the physical memory and update the page table. When the valid bit is set as valid, the page is already in the physical memory and the program directly write to the frame. If the frame is not found, the program would check the valid/invalid bit in the inverted page table to see whether there is empty frame in the physical memory. If there is an empty frame, the program will map the page to the empty frame, update the page table and write directly in the physical memory where the empty frame locates. If all the frames are occupied, the program would swap the LRU page to the disk and write to the selected frame after the swapping procedure.

The `vm_read()` has almost the same procedure of the `vm_write()` function, except that the function do not write to the memory. Instead, it load the data from the memory to output. As for the function `vm_snapshot()`, it is implmented by calling the function `vm_read()` and read the content of the memory bit by bit.

3.6 Design of Bonus Part

The bonus part is also implemented in the `bonus` file. In this project, the bonus part is implemented based on the **second** version of the bonus task illustration.

The code in the bonus part is almost the same as the one that is implemented in the main task. The only different is that the program launches 4 threads in the kernel function. This is implemented using the following code.

```
dim3 grid(1, 1), block(4, 1);
mykernel<<<grid, block, INVERT_PAGE_TABLE_SIZE>>>(input_size);
```

Since the 4 threads would run concurrently when the program is executed, it is important to deal with race condition. According to the requirement of the bonus task, the execution of the `vm_read()` and `vm_write()` should be a non-preemptive priority scheduling. Therefore, the function `__syncthreads()` is applied in the `mykernel()` function. The `__syncthreads()` function may guarantee that each thread will synchronously run to the same position in the function. The threads which run faster may wait for the slower ones. Using this method, the scheduling can be maintained.

4 Result Analysis

The result of the first test case of the main task of this prject is 8193 with the following information in the file `output.out`.

```
main.cu(92): warning #2464-D: conversion from a string literal to "char
    *" is deprecated

main.cu(111): warning #2464-D: conversion from a string literal to "
   char *" is deprecated

main.cu(92): warning #2464-D: conversion from a string literal to "char
    *" is deprecated
```

```
main.cu(111): warning #2464-D: conversion from a string literal to "
    char *" is deprecated

input size: 131072
pagefault number is 8193
```

The user program of the first test case is shown as following.

```
#include "virtual_memory.h"
#include <cuda.h>
#include <cuda_runtime.h>

__device__ void user_program(VirtualMemory *vm, uchar *input, uchar *
    results,
                             int input_size) {
  for (int i = 0; i < input_size; i++)
    vm_write(vm, i, input[i]);

  for (int i = input_size - 1; i >= input_size - 32769; i--)
    int value = vm_read(vm, i);

  vm_snapshot(vm, results, 0, input_size);
}
```

The user program first write 131072 bytes to the memory. Since the page size is 32 bytes, and each frame is considered as empty at the beginning. Therefore, 1024 page faults occur when the user program is writing to the first 32 KB memory. When the value of i is larger than 32767, the physical memory is full. For the following write operation, there would be in total $\frac{131072-32768}{32} = 3072$ page faults. This is because page fault occurs when the address is a multiple of 32. In total, the number of page fault occurs in the `vm_write()` function in the first for loop is $1024 + 3072 = 4096$. As for the `vm_read()` function in the second for loop, there would be only one page fault. The reason is that after the first for loop, the pages in the physical memory would be the one with the last 1024 indices from page number 5120 to page number 4096. In the second loop, the program actually accesses the last 1024 pages and an extra byte. The extra byte would cause page fault since the page including that byte is currently not in the page table. At the end of the user program, function `vm_snapshot()` is called. The implementation of the function `vm_snapshot()` is shown as following.

```
__device__ void vm_snapshot(VirtualMemory *vm, uchar *results, int
    offset, int input_size) {
/* Complete snapshot function togther with vm_read to load elements
    from data
* to result buffer */
  for (int k = 0; k < input_size; k++) {
      results[k] = vm_read(vm, k + offset);
  }
}
```

The function read the data starting from the address k + offset, which is in the range $[0, 131071]$ in this case. Since the first 1024 pages are not in the page table currently and after they are accessed, the following pages may also suffer page fault. Therefore, all the pages would suffer from page faults and the total number of page fault at this stage is $\frac{131072}{32} = 4096$. In general, the number of page fault using the user program of the first test case is

$$4096 \text{ (first for loop)} + 1 \text{ (second for loop)} + 4096 \text{ (final snapshot)} = 8193,$$

which is exactly the result in the `output.out` file.

The second test case applies the following user program.

```
#include "virtual_memory.h"
#include <cuda.h>
#include <cuda_runtime.h>

__device__ void user_program(VirtualMemory *vm, uchar *input, uchar *
    results,
  int input_size) {
// write the data.bin to the VM starting from address 32*1024
for (int i = 0; i < input_size; i++)
  vm_write(vm, 32*1024+i, input[i]);
// write (32KB-32B) data  to the VM starting from 0
for (int i = 0; i < 32*1023; i++)
  vm_write(vm, i, input[i+32*1024]);
// readout VM[32K, 160K] and output to snapshot.bin, which should be
    the same with data.bin
vm_snapshot(vm, results, 32*1024, input_size);
}

// expected page fault num: 9215
```

The information in the `output.out` is shown as following.

```
main.cu(92): warning #2464-D: conversion from a string literal to "char
     *" is deprecated

main.cu(111): warning #2464-D: conversion from a string literal to "
    char *" is deprecated

main.cu(92): warning #2464-D: conversion from a string literal to "char
     *" is deprecated
```

```
main.cu(111): warning #2464-D: conversion from a string literal to "
    char *" is deprecated

input size: 131072
pagefault number is 9215
```

The second test case applies the same input file as the first test case. The difference is the order of reading from and writing to the memory. The user program first write data to the memory with address starting from 32768. Since the original values of the page numbers in each frame in the inverted page table is set to be from 0 to 1023 and the address 32768 is actually accessing the page with page number 1024. Therefore, in the first for loop, one page fault occurs in each iteration and in total $\frac{131072}{32} = 4096$ page faults occur. In the second for loop, the user program write to the memory with address from 0 to 32767. In the current page table, the value of page number is from 4096 to 5119. What the user program do is to write data to pages with page number 0 to 1022. Therefore, 1023 page faults occurs in this for loop. Finally, the user program call the `vm_snapshot()` function, which will read the data from the pages with page number from 1024 to 5119. Since the current page table stores the information of pages with page number from 0 to 1022 and the last entry stores the page with page number 5119. Therefore, the function call may lead to 4096 additional page faults. In general, the total number of page fault will be:

$$4096 \text{ (first for loop)} + 1023 \text{ (second for loop)} + 4096 \text{ (snapshot)} = 9215.$$

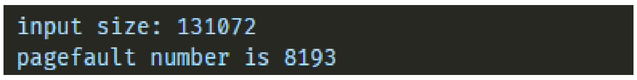
For the bonus part, the result would be 32772. The second version of bonus requires that the four threads perform the whole process of testcase 1. In this case, the next thread may overwrite the content of the previous thread. Since all the threads perform the same operations, their number of page faults would be the same. The total number of page faults would be $4 \times 8193 = 32772$. The output information of the execution is shown as following.

```
In file included from main.cu:4:0:
/usr/local/cuda/bin/../targets/x86_64-linux/include/device_functions.h
    :54:2: warning: #warning "device_functions.h is an internal header
    file and must not be used directly.  This file will be removed in a
     future CUDA release.  Please use cuda_runtime_api.h or
    cuda_runtime.h instead." [-Wcpp]
 #warning "device_functions.h is an internal header file and must not
     be used directly.  This file will be removed in a future CUDA
     release.  Please use cuda_runtime_api.h or cuda_runtime.h instead.
     "
  ^
main.cu(126): warning #2464-D: conversion from a string literal to "
    char *" is deprecated
```

```
main.cu(146): warning #2464-D: conversion from a string literal to "
    char *" is deprecated

In file included from main.cu:4:0:
/usr/local/cuda/bin/../targets/x86_64-linux/include/device_functions.h
    :54:2: warning: #warning "device_functions.h is an internal header
    file and must not be used directly.  This file will be removed in a
     future CUDA release.  Please use cuda_runtime_api.h or
    cuda_runtime.h instead." [-Wcpp]
 #warning "device_functions.h is an internal header file and must not
     be used directly.  This file will be removed in a future CUDA
     release.  Please use cuda_runtime_api.h or cuda_runtime.h instead.
     "
  ^
main.cu(126): warning #2464-D: conversion from a string literal to "
    char *" is deprecated

main.cu(146): warning #2464-D: conversion from a string literal to "
    char *" is deprecated

input size: 131072
pagefault number is 32772
```

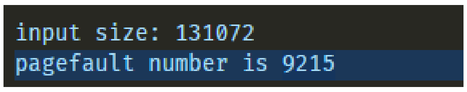
5 Screenshot of the Output

1. output of testcase 1 with one thread



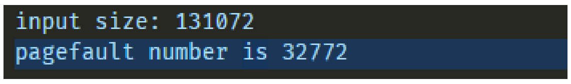
```
input size: 131072
pagefault number is 8193
```

2. output of testcase 2 with one thread



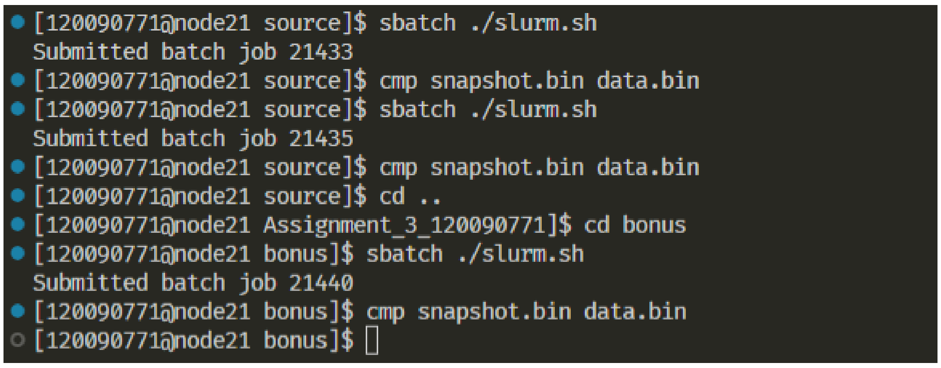
```
input size: 131072
pagefault number is 9215
```

3. output of testcase 1 with four threads (bonus)



```
input size: 131072
pagefault number is 32772
```

4. for further testing, compare the output file `snapshot.bin` with the input file `data.bin` using the linux command `cmp snapshot.bin data.bin`. In the above



```
● [120090771@node21 source]$ sbatch ./slurm.sh
  Submitted batch job 21433
● [120090771@node21 source]$ cmp snapshot.bin data.bin
● [120090771@node21 source]$ sbatch ./slurm.sh
  Submitted batch job 21435
● [120090771@node21 source]$ cmp snapshot.bin data.bin
● [120090771@node21 source]$ cd ..
● [120090771@node21 Assignment_3_120090771]$ cd bonus
● [120090771@node21 bonus]$ sbatch ./slurm.sh
  Submitted batch job 21440
● [120090771@node21 bonus]$ cmp snapshot.bin data.bin
○ [120090771@node21 bonus]$ ▯
```

figure, the job with number 21433 is the execution of testcase 1 using 1 thread. The job with number 21435 is the execution of testcase 2 using 1 thread and the job with number 21440 is the execution of testcase 1 using 4 threads. The result shows that all of the output files are the same as the input file. There is no information about the difference between the two files appear in the terminal after using `cmp` command, which indicates that the output information of the original program is correct.

6 Problems

In this section, some of the problems appear when implementing the project will be discussed.

Problem: The paging and swapping machanism is hard to understand.

Solution: The textbook actually provides a detailed illustration on how the paging and swapping work in operating systems. It would be a good idea to refer to the textbook. Maybe it is still hard to read it because it is quite long. However, if one have read the textbook carefully and patiently, he or she is able to figure out how these two machnisms work.

Problem: At the beginning, one may have no idea on how to implement the LRU algorithm in this project.

Solution: Actually, several ways on how to implement the algorithm is discussed on the textbook. Therefore, it is important to read the textbook carefully. Also, it might be helpful to draw some figures like **Figure 1** before coding because sometimes figures may help programmers to clear their minds.

Problem: How to launch 4 threads and let them execute the user program currently?

Solution: It would be useful to search on google for tutorials related to CUDA programming. Make use of resource is also important.

Problem: How to test the program?

Solution: Actually, the TAs of course CSC4005 have already sent the emails which describe the process of how to log in the cluster and change the password. For more detailed information, one could check the related project on github. It is actually a good idea to apply the cluster in CSC3150, which provide great convenience on program testing.

7 Things I Learnt from the Project

The most important thing I learnt from this project is the details related to virtual memory paging machanism. From the lecture, what we could learn is theory, which might be abstract. Without going through the detailed implementation of the memory management process, some significant problems may still be unknown to a newcomer of operating system. Implementing this project provides me with a more detailed view on the behavior of a memory system.

In addition, this project gives me a chance to practise using CUDA programming. Before this project, I have no experience in CUDA programming, which might be useful for my future study. From this assignment, I really learn something related to CUDA from google.

Besides, doing the programming project not only enhence my understanding on related theory, but also improve my ability to learn new things quickly. Technologies in computer science develop quickly and it would be of great necessity to equipped with the ability of quick learning as a comouter science student.

Finally, thank you for reading my long project report.