CSC3150 Assignment3 Report

Student ID: 119010104

Name: Hu Yiyan

Environment

I only specify the environment I use.

- Operating System

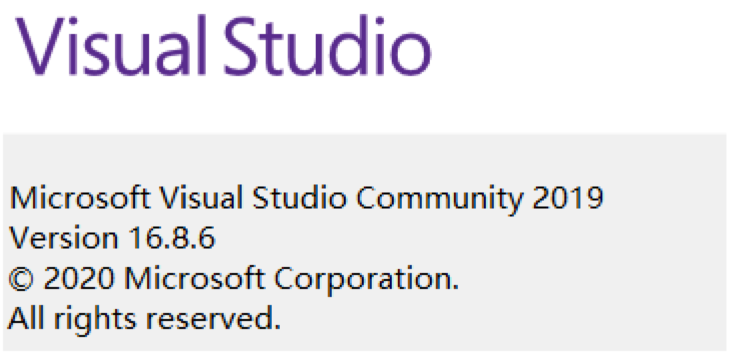
  Windows 10

```
PS C:\Users\11957> [System.Environment]::OSVersion.Version

Major  Minor  Build  Revision
-----  -----  -----  --------
10     0      19042  0
```

- Visual studio version

  Visual Studio 2019 Community



Microsoft Visual Studio Community 2019
Version 16.8.6
© 2020 Microsoft Corporation.
All rights reserved.

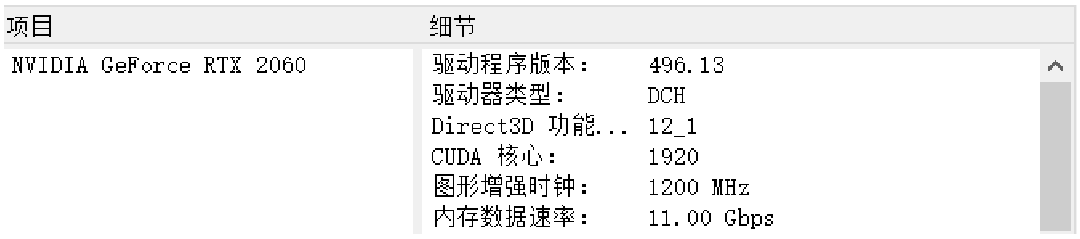
- CUDA version

  version 11.5

```
PS C:\Users\11957> nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2021 NVIDIA Corporation
Built on Mon_Sep_13_20:11:50_Pacific_Daylight_Time_2021
Cuda compilation tools, release 11.5, V11.5.50
Build cuda_11.5.r11.5/compiler.30411180_0
```

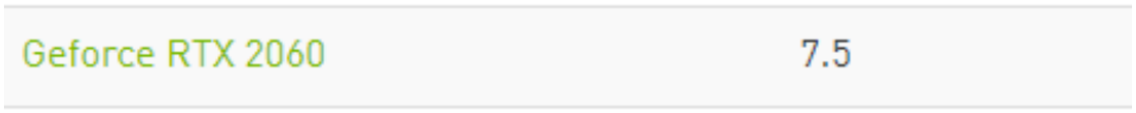
- GPU Information

  GTX 2060



项目	细节	
NVIDIA GeForce RTX 2060	驱动程序版本：	496.13
	驱动器类型：	DCH
	Direct3D 功能...	12_1
	CUDA 核心：	1920
	图形增强时钟：	1200 MHz
	内存数据速率：	11.00 Gbps

Compute capacity: 7.5



Geforce RTX 2060	7.5

Execution

Method 1: Use visual studio

You can set up the project in the visual studio and run the program. All you need to do is press down the *Ctrl+F5*.

Method 2: Use Windows Command or Windows Powershell

If you use the Windows 10 OS and have the Cuda installed on your PC, you may run the program by typing commands in the Windows command window or Windows Powershell. The following figure shows the steps to run the program in Windows Powershell.

```
PS C:\Users\11957\Desktop\Ass3\Source> nvcc -rdc=true -o main virtual_memory.cu
user_program.cu main.cu
virtual_memory.cu
user_program.cu
main.cu
main.cu(132): warning #2464-D: conversion from a string literal to "char *" is
deprecated

main.cu(151): warning #2464-D: conversion from a string literal to "char *" is
deprecated

main.cu(132): warning #2464-D: conversion from a string literal to "char *" is
deprecated

main.cu(151): warning #2464-D: conversion from a string literal to "char *" is
deprecated

  正在创建库 main.lib 和对象 main.exp
PS C:\Users\11957\Desktop\Ass3\Source> ./main
input size: 131072
pagefault number is 8193
```

Method 3: Use Makefile

You may test my program in the Centos environment (which is the environment of the computers in the TC301). In this case, you can simply type: `make` to run the program. The following figure shows the steps to run the program.

```
[cuhksz@TC-301-35 Source]$ make
nvcc -rdc=true -o main virtual_memory.cu user_program.cu main.cu
main.cu(132): warning: conversion from a string literal to "char *" is
deprecated

main.cu(151): warning: conversion from a string literal to "char *" is
deprecated

main.cu(132): warning: conversion from a string literal to "char *" is
deprecated

main.cu(151): warning: conversion from a string literal to "char *" is
deprecated

./main
input size: 131072
user program: size is 131072
write starts
read starts
pagefault number is 8193
```

Note: you may encounter a wired situation: `Makefile:2: *** missing separator.  Stop.` To solve this, you should change the format of the Makefile. You can simply change every indent in the Makefile to `Tab` (This situation happens we I test on the computer in the TC301, I think the problem is caused by different indent syntax on different OS).

Design

1. Overall Idea: Virtual Memory, Physical Memory and Disk Memory

> In the Assignment 3, we are required to simulate a mechanism of virtual memory. Virtual memory allows the execution of processes that are not completely in memory, in other words, the logical memory address can exceed the bound of physical memory. To implement the virtual memory, we can use table and disk (secondary) storage.
>
> **Virtual Memory**
>
> The logical address of a datum is its address in the virtual memory. The virtual memory is a conceptual (not physical) memory space, it may exceed the size of physical memory limit. In this assignment, to make the implementation easier, the size of the virtual memory is bounded by the size of the disk (secondary) memory (128KB).
>
> **Physical Memory**
>
> The physical memory corresponds to the main memory of a computer. When performing the read and write operations, we should (1) search for the page in the physical memory, (2) do other operations if we cannot find the page in the physical memory. In this assignment, the physical memory is of 32KB, which is less than the virtual (logical) memory size.

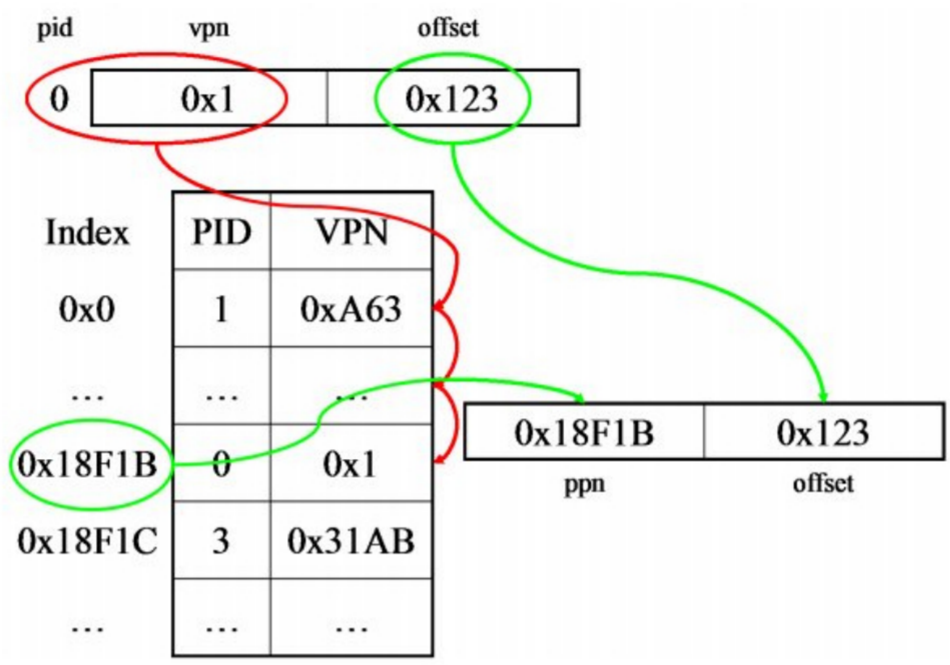
Disk (Secondary) Memory

The disk (secondary memory) is larger than the physical memory. However the access speed to access this part of memory is much slower than that of the physical memory. The disk can be regarded as the back-up storage of the physical memory. If we cannot find the desired page in the physical memory, we can access the disk storage to find that page. In this assignment, the disk (secondary) memory is 128KB. For the logical address is bounded by disk memory size, we can establish a one-to-one relationship between each byte of the disk memory and the logical address.

Page Table

To implement the virtual memory, we need to match the logical address of a datum to a page in the physical memory. The page table can achieve this mapping function.

A page table consists of page table entries. Each page entry corresponds to a page in the physical memory. Each entry also stores the logical address information (virtual page number, valid bit and dirty bit etc.). The mechanism of the page table is shown in the figure below.



In this assignment, each page has 32 bytes, the page table has 1-24 entries (the physcial memory also has 1024 entries, there is an bijection relationship between page table and the physical memory pages).

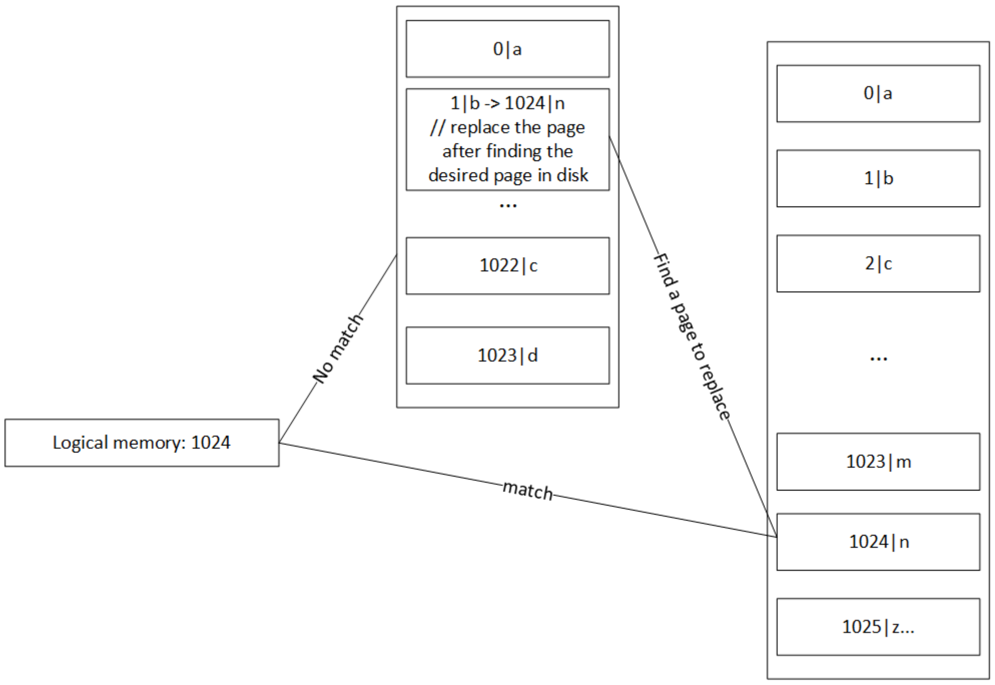
Page Replacement

When the desired page is absent in the physical memory, the OS will perform a certain operation to find that page. In this assignment, we need to implement the page replacement to realize this function. In case that the page is not in the physical memory, we will find the corresponding page in the disk memory and replace the least recently used page in the physical memory.

The logic of the page replacement is:

`Find LRU page in physical memory -> Swap the content of the page into the corresponding disk memory -> find the page in the disk memory that corresponds to the logical address of the datum -> swap the page in the disk memory to the desired page in the physical memory -> perform read or write -> update page` `table`

The following figure demonstrates the page replacement (**This figure does not show the step of swapping page from physical memory to disk memory**, but I think it is enough to demonstrate the idea).



2. Implementation Details & Flowchart

Details: Special relationships

1. Page table & physical memory

   `PageTableEntry_index = PhysicalMemory_frame_index`

2. Logical address & disk memory

   `LogicalAddress = DiskMmeory_byte_index`

3. Page table & physical address

   `PhysicalAddress = VirtualPageNumber << page_offset + offset`

4. Logical address & page table

   `VirtualPageNumber = LogicalAddress >> page_offset`

   Page table entry stores the VirtualPageNumber

Details: Page table

Page table contains page entries, each entry contains the following information:

1. Valid bit: indicates whether the page is in the physical memory.
2. Dirty bit: indicates whether the page in the physical memory is consistent with the disk memory

   note:

   read operation -> don not change dirty bit

   write operation -> set dirty bit dirty

swap from disk memory to physical memory -> only operates when dirty bit dirty

swap from physical memory to disk memory -> set dirty bit clean

3. VPN (Virtual page number): `VirtualPageNumber = LogicalAddress >> page_offset`

Details: Dumping into the snapshot

This part is quite simple, you just need to use the `vm_read()` function and the templated provided on the blackboard.

Details: LRU page finding

When implementing the page replacement, we need to find the LRU page. I implement finding the LRU page by using a counter array.

The counter array records how long the page has not been accessed

```
__device__ __managed__ u32 ptCounter[1024] = {0};
// counter array, each element of the array corresponds to a page in the
physical memory
// eg: buffer[a*32] matches ptCounter[a]
```

Once the page in the physical memory is referred, the counter of that page need to be reset to 0. The other pages need to increase by 1.

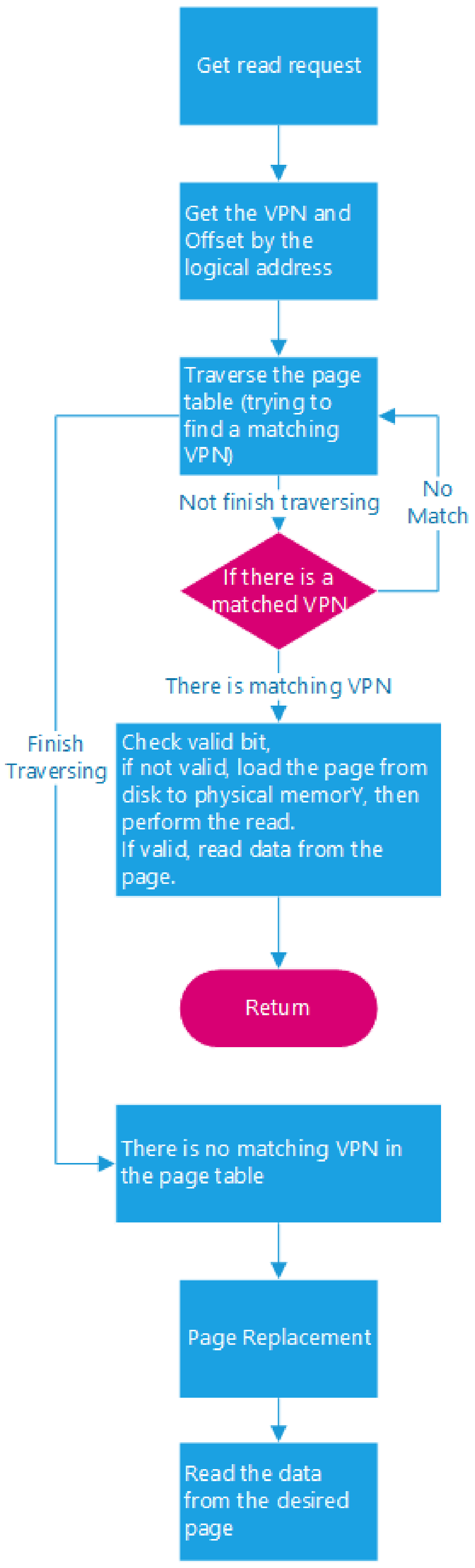
```
for (int k = 0; k < size; k++) { // increment counter
    vm->pageTableCounter[k] += 1;
}
vm->pageTableCounter[i] = 0; // reset counter of the accessed page
```

During page replacement, we should find the LRU page. I do linear search on the counter array to find the LRU page.

```
__device__ int FindLRUPageIndex(VirtualMemory* vm) {
    u32 max;
    max = vm->pageTableCounter[0];
    int index = 0;
    for (int i = 0; i < 1024; i++) {
        if (vm->pageTableCounter[i] > max) {
            max = vm->pageTableCounter[i];
            index = i;
        }
    }
    return index;
}
```

Flowchart: The implementation logic

- Read operation


Get read request
Get the VPN and Offset by the logical address
Traverse the page table (trying to find a matching VPN)
Not finish traversing
No Match
If there is a matched VPN
There is matching VPN
Finish Traversing
Check valid bit, if not valid, load the page from disk to physical memorY, then perform the read. If valid, read data from the page.
Return
There is no matching VPN in the page table
Page Replacement
Read the data from the desired page

- Write Operation

  The logic of the write operation is similar to that of the the read operation. The only difference is that write operations writes to the physical memory instead of return a value from the physical memory. For convenience, I will not show the flowchart of the write operation.

3. Bonus: Brief Design Illustration

In the bonus part, we are required to launch 4 threads and execute them in sequence, which involves implementation of thread creation and mechanism to manage threads.

Thread Creation

We can initialize multiple threads in the same block.

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

Note that we cannot initialize the threads in different blocks. Different blocks cannot share memory identified by the `_shared_` keyword.

Thread Managing Mechanism

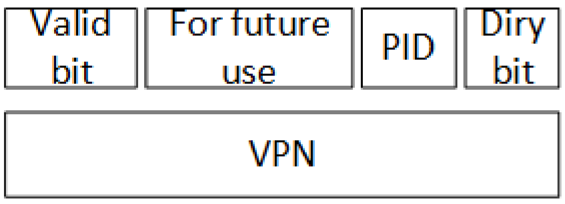
- Shared memory & page table

  The 4 thread will share the memory. In particular, they need to share the same page table. To prevent the processes mess up the memory, we can modify the page table to distinguish the memory between processes.

  We can add PID component to each page table entry to indicate that the page entry is belong to a certain thread. Now each page table entry contains:

1. VPN
2. PID: the thread id
3. Valid bit and dirty bit

The page table structure is shown in the following figure.



- Thread piriority

  The priority relationship of the threads: thread 0 > thread 1 > thread 2 > thread 3, thus the sequence to execute the threads: thread 0->thread 1->thread 2->thread 3. To make the thread execute in sequence without memory conflict, I use the `__syncthreads()`.

  The `__syncthreads()` command is a **block level** synchronization barrier. That means it is safe to be used when all threads in a block reach the barrier. If several thread programs are put in a sequential order, putting `__syncthreads()` between thread programs indicates that the program can only go to the next thread program when the previous thread finishes the execution (reaches the barrier). I put `__syncthreads()` between the thread programs to make the thread executes in the desired sequence. Below is the structure of my program to execute the threads.

```
if (threadIdx.x == 0) { // thread #0
        // thread 0 program
    }
    __syncthreads(); // barrier
    if (threadIdx.x == 1) { // thread #1
        // thread 1 program
    }
    __syncthreads(); // barrier
    if (threadIdx.x == 2) { // thread #2
        // thread 2 program
    }
    __syncthreads(); // barrier
    if (threadIdx.x == 3) { // thread #3
        // thread 3 program
    }
}
```

Page Fault Illustration

1. Pagefault Output

I use the Windows Powershell to display the pagefault output.

Basic Part

```
PS C:\Users\11957\Desktop\Ass3\Source> ./main
input size: 131072
pagefault number is 8193
```

Bonus Part

I add `printf()` function in the main.cu to display which thread is currently being executed. In this case the terminal will display the thread exection information.

```
PS C:\Users\11957\Desktop\Ass3\Bonus> ./main
input size: 131072
now is the thread:0
now is the thread:1
now is the thread:2
now is the thread:3
pagefault number is 32772
```

2. Pagefault Output Illustration

When the pagefault occur?

1. Page entry: invalid
2. Page replacement

Basic part: 8193 page faults

To check the page fault, we need to check `user_program()`, the `user_program()` consists of three parts: write part, read part and the snapshot part.

- Write part

```
for (int i = 0; i < input_size; i++) {
    vm_write(vm, i, input[i]);
}
```

`\because` Input size: 131072 -> 4096 pages -> 4*1024 pages->4 page table size

`\therefore` The above program fill the page table 4 times

`\beacause` At first all the entries in the page table is invalid, accessing the page table for the first time -> a page fault

`\therefore` The first 1024 iterations -> 1024 page faults

`\beacuse` After the first 1024 iterations, the page table is always full, and the new logical address cannot find the corresponding VPN in the page table.

`\therefore` Every iteration after the first 1024 iterations will call the page replacement function.

`\therefore` The latter 3*1024 iterations -> 3*1024 page faults.

`\therefore` This part gives 4096 page faults.

- Read part

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

`\beacause` iteration times: 32769 times->32*1024+1 times->page_size*1024 + 1

`\therefore` for the first 1024 access of the page entries, there will always be a read hit (since the reading process starts from the highest logical address, which is just written), and for the 1025th access of the page entry, there will be a page fault for the VPN is not in the page table.

`\therefore` this part results 1 pagefault

- Snapshot part

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

`\because` there are four 1024-pages access in total (we can regard accessing 1024-pages as one big access)

`\because` the snap shot part starts from the logical address 0, while the page table contains the VPNs of the high logical addresses.

`\therefore` the first 1024-pages access result 1024 page-replacement page faults (similar to that of the write part)

`\therefore` for the latter three 1024-pages accesses, each 1024-pages access results 1024 page replacement page faults (similar to that of the write part)

`\therefore` Snapshot part gives 4*1024 page faults->4096 page faults

In total: `#pagefaults = 4096+4096+1 = 8193`

Bonus part: 8193*4 page faults

The bonus part of program execute four threads, each thread executes the program of the basic part. The process of bonus part can be regarded as four processes of the basic part.

thread 0:

Page table entries are all invalid at first. The situation of thread 0 is the same as the situation of the process of the basic part -> 8193 page faults.

thread 1~3

Page table entries all contains the PID of the last thread, meaning that the entries are invalid for the current thread. In this case, the thread 1~3 yields the same situation of the process of the basic part. -> each thread generates 8193 page faults.

In total: `#pagefaults = 8193*4 = 32772`

Results & Output

I display the output of the programs on the Windows Powershell.

1. Output: Basic Part

```
PS C:\Users\11957\Desktop\Ass3\Source> ./main
input size: 131072
pagefault number is 8193
```

2. Output: Bonus Part

```
PS C:\Users\11957\Desktop\Ass3\Bonus> ./main
input size: 131072
now is the threadx:0
now is the threadx:1
now is the threadx:2
now is the threadx:3
pagefault number is 32772
```

Future Improvement

1. Problem: Page Table Counter Overflow

I use a counter array to count how long the page has not been accessed. The element of the array is of u32 type. If the counter value of the page exceeds `2^32-1`, then there will be an overflow.

To solve the problem, we should add more codes. When there is an overflow detected, we want to modify the counter values of physical memory. We should reset the values in the counter array from 0 to 1023.

2. Improvement: Page Replacement --Page Table Stack

If we use page table counter to implement LRU, each page replacement yields operation with O(n) time, which can be time consuming as the page table gets very large.

If the speed is the critical issue, we may want to implement an **stack** to indicate the sequence of the accessing time of each page table.

Each time we access a page, we move the corresponding node of that page in the stack to the top. Each time we need to find the LRU page, we go to the bottom of the stack. The operations using stack are all in O(n) time.

However, if the storage is more critical, I think we'd better to use the counter array.

3. Improvement: Hashed Inverted Page Table

It cost time to linearly search the page table. We may upgrade the page table by implementing a hashed inverted page table, reducing time overhead.

Conclusion

In this assignment, I have

1. Implement virtual memory (page table, LRU, page replacement ...) -> get better understanding of memory management.
2. Investigate different data structures when implementing the LRU -> Recall some knowledge of the data structure.
3. Debug a lot -> Become a printf() master and learn to use debugging tools such as nvcc adb.
4. Thinking of the memory management logic -> review the CSC3150 course content a bit.

   The code of this assignment is quite simple once I figure out the logic. I think I have learnt something.

Thanks for grading my assignment and reading this long report!