CSC3150 Assignment3 Report

Introduction

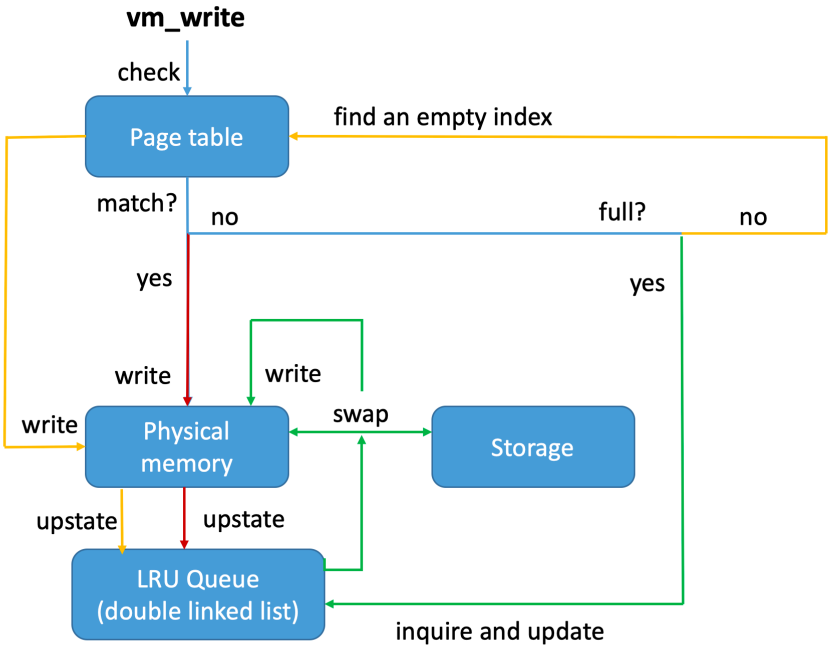
In this assignment, a simple virtual memory simulation is implemented in a kernel function of GPU that has single thread, limit shared memory and global memory. The source code of this virtual memory implementation is attached with this report, which is in the file *"virtual_memory.cu"*.

Specification of the GPU Virtual Memory:

- Secondary memory (global memory) - 128KB
- Physical memory (shared memory) - 48KB (32KB for data buffer and 16KB for page table setting)
- Page size - 32 bytes
- Page table entries - 1024 (32KB / 32 bytes)
- Page replacement policy for page fault: **LRU** (Least Recently Used)

Program design

vm_write



The above logic graph shows the basic logic of the implementation of *vm_write*.

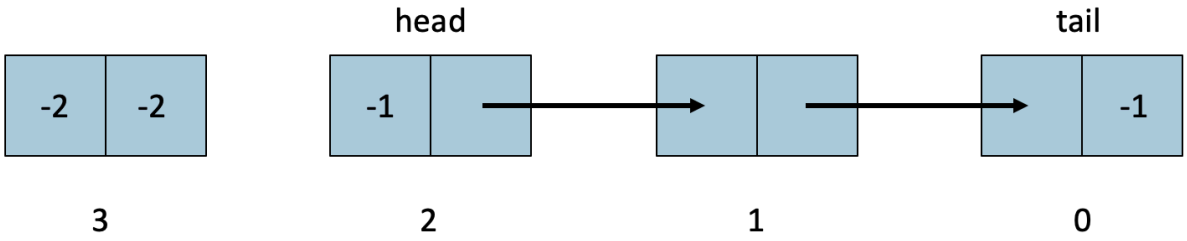
In the graph, the modules page table and physical memory are specified in the introduction. Module Storage relates to Secondary memory. What needs to be emphasized is that, the module Queue is a data structure used for implementing LRU. It uses double linked list to construct the queue.

There are 16KB memory space preset for page table setting but only half of it has been used, so there could be extra two integers for every index to record. To implement double linked list, we utilized these two integers - one integer represents the left element (next-accessed index) and the other represents the right element (previous-accessed index). If there is no left element, which means it is the head of the list, then fill with -1; if there is no right element, which means it is the tail of the list, then fill with -1 as well. All unused index were initialized with left = -2 and right = -2.

For example, if we access index 0, 1, 2 in order at beginning, then the corresponding record in index 0, 1, 2, 3 is as following:

index	**0**	**1**	**2**	**3**
left	1	2	-1	-2
right	-1	0	1	-2

Visualize this double linked list (queue):



Based on this setting, we could quickly get the least recent used index and swap it out when page replacements happen. Also, when some data in VM was accessed, we could easily update the queue by just change several numbers. These operations could be found in the code.

After the LRU queue has been built, we could go through the flow of the program.

As the logic graph shows, when *vm_write* was called, it firstly checked the page table to get the current information of the page table, like whether there is a matching index or empty index, or whether the queue has the head and tail, the index will be recorded if it exists. Four variables were set for this purpose: *target_index*, *head_index*, *tail_index*, *empty_index* which is initially set as -1 to represent false:

```
int target_index, head_index, tail_index, empty_index;
target_index = empty_index = head_index = tail_index = -1;
int left, right;
for (int i = 0; i <vm->PAGE_ENTRIES; ++i){
      if (target_index < 0 && vm->invert_page_table[i] == base) target_index =
i; // find the corresponding index
      if (head_index < 0 && vm->invert_page_table[i + 2 * vm->PAGE_ENTRIES] ==
-1) head_index = i;  // find the most recent index
      if (tail_index < 0 && vm->invert_page_table[i + 3 * vm->PAGE_ENTRIES] ==
-1) tail_index = i;  // find the least recent index
      if (empty_index < 0 && vm->invert_page_table[i] == 0x80000000)
empty_index = i; // find the empty index
      if (target_index >= 0 && head_index >= 0) break;  // to reduce running
time, break in advance if the necessary indexes have been found
}
```

If there is a matching index, then write the input into the data buffer (physical memory) :

```
if (target_index >= 0) {
  // refresh the linked-list based on priority
  left = vm->invert_page_table[target_index + 2 * vm->PAGE_ENTRIES];
  right = vm->invert_page_table[target_index + 3 * vm->PAGE_ENTRIES];
  if (left != -1) {   // target_index is not the head_index
    vm->invert_page_table[left + 3 * vm->PAGE_ENTRIES] = right; // left.next =
right
    vm->invert_page_table[target_index + 2 * vm->PAGE_ENTRIES] = -1;  //
target.previous = -1
    if (right != -1)    // target_index is not the last
      vm->invert_page_table[right + 2 * vm->PAGE_ENTRIES] = left; //
right.previous = left
    vm->invert_page_table[head_index + 2 * vm->PAGE_ENTRIES] = target_index;
// head.previous = target
    vm->invert_page_table[target_index + 3 * vm->PAGE_ENTRIES] = head_index;
// target.next = head
  }
  printf("Read: %d; page faults: %d\n", (int)addr, vm->pagefault_num_ptr[0]);
  return vm->buffer[target_index * vm->PAGESIZE + offset];
}
```

If there is no matching index, increment the counting on page faults:

```
vm->pagefault_num_ptr[0] += 1;  // count the page fault
```

If there is empty index:

```
if (empty_index >= 0) {   // the page table is not empty
```

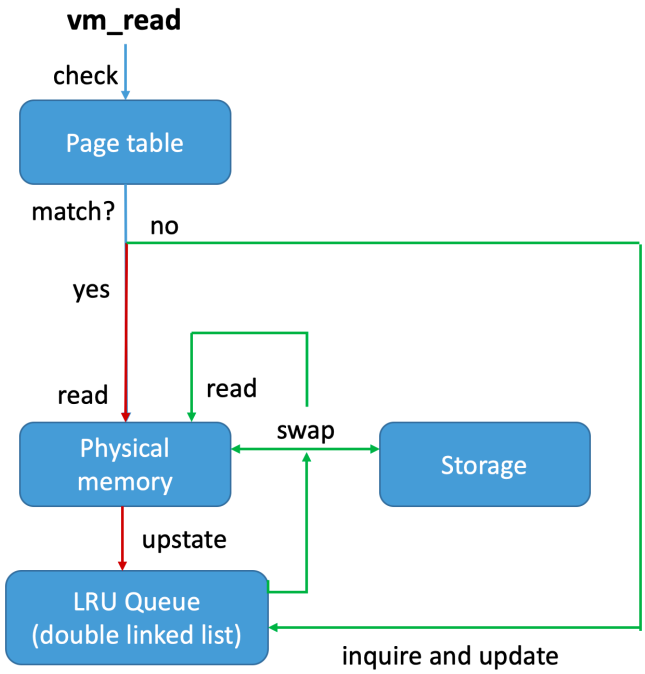
```
    vm->invert_page_table[empty_index] = base;  // edit the new index of the
page table
    vm->invert_page_table[empty_index + vm->PAGE_ENTRIES] = 1;  // change the
status
    vm->buffer[empty_index * vm->PAGESIZE + offset] = value;  // write into
buffer
    printf("write: %d; page faults: %d\n", (int)addr, vm-
>pagefault_num_ptr[0]);
    if (head_index >= 0) {    // head_index exists
        vm->invert_page_table[head_index + 2 * vm->PAGE_ENTRIES] =
empty_index;
        vm->invert_page_table[empty_index + 2 * vm->PAGE_ENTRIES] = -1;
        vm->invert_page_table[empty_index + 3 * vm->PAGE_ENTRIES] =
head_index;
    }
    else {  // it is the first accessed index
        vm->invert_page_table[empty_index + 2 * vm->PAGE_ENTRIES] = -1;
        vm->invert_page_table[empty_index + 3 * vm->PAGE_ENTRIES] = -1;
    }
}
```

Otherwise the page table is full, then do the page replacement on the least recent used index, which is dequeued by the linked list we implemented:

```
else {            // the page table is full -- page swap
      int new_tail = vm->invert_page_table[tail_index + 2 * vm->PAGE_ENTRIES];
      // swap out to the storage
      for (int i = 0; i < vm->PAGESIZE; ++ i) {
        vm->storage[vm->invert_page_table[tail_index] * vm->PAGESIZE + i] =
vm->buffer[tail_index * vm->PAGESIZE + i];
      }
      vm->buffer[tail_index * vm->PAGESIZE + offset] = value;
      printf("write: %d; page faults: %d\n", (int)addr, vm-
>pagefault_num_ptr[0]);
      vm->invert_page_table[tail_index] = base; // edit the new index of the
page table
      vm->invert_page_table[new_tail + 3 * vm->PAGE_ENTRIES] = -1;  //
new_tail.next = -1
      vm->invert_page_table[tail_index + 2 * vm->PAGE_ENTRIES] = -1;  //
become head
      vm->invert_page_table[tail_index + 3 * vm->PAGE_ENTRIES] = head_index;
      vm->invert_page_table[head_index + 2 * vm->PAGE_ENTRIES] = tail_index;
    }
```

The above is all the implementation about *vm_write*.

vm_read



As the logic graph shows, *vm_read* is quite similar to *vm_write*. It does not need to do empty check but in the swap step, it should do both swapping out and swapping in:

```
// swap out
for (int i = 0; i < vm->PAGESIZE; ++ i) {
    vm->storage[vm->invert_page_table[tail_index] * vm->PAGESIZE + i] = vm-
>buffer[tail_index * vm->PAGESIZE + i];
}
// swap in
for (int i = 0; i < vm->PAGESIZE; ++ i) {
    vm->buffer[tail_index * vm->PAGESIZE + i] =  vm->storage[base * vm-
>PAGESIZE + i];
}
```

And *vm_read* should return the read data at last:

```
return vm->buffer[target_index * vm->PAGESIZE + offset];
```

All the other code of *vm_read* is almost the same as the *vm_write*, since they are somehow like reverse operation for each other. Please refer to source code for details.

vm_snapshot

For snapshot, just simply do the *vm_read* in the same order of input:

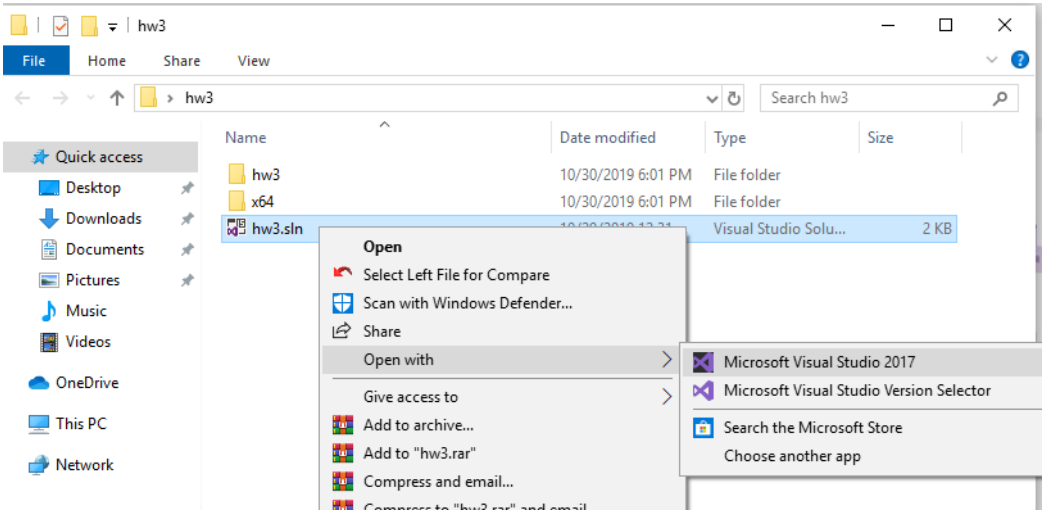
```
__device__ void vm_snapshot(VirtualMemory *vm, uchar *results, int offset,
                            int input_size) {
   printf("Start of snapshot:\n");
   for (int i = 0; i < input_size; ++i){
     results[i] = vm_read(vm, i); // read from VM in order
     printf("In snapshot - ");
   }
}
```

Running environment

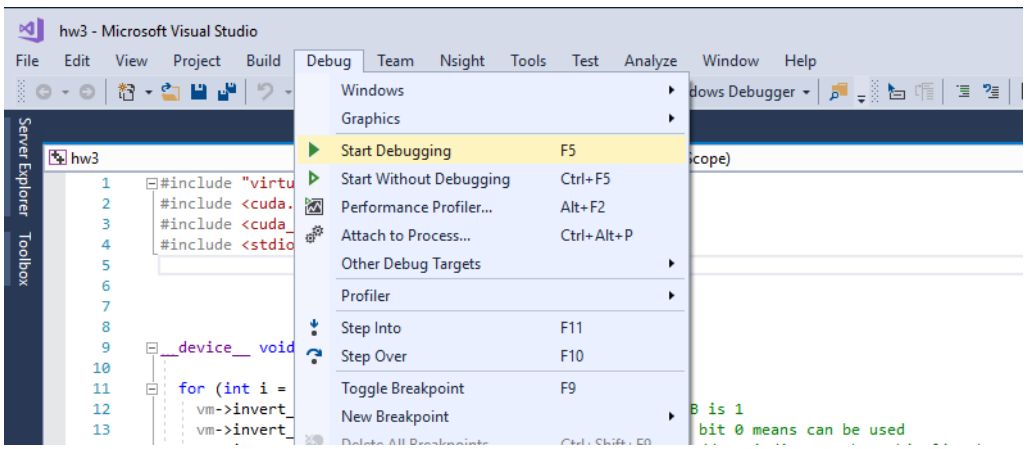
- Windows 10 Enterprise
- Microsoft Visual Studio 2017
- CUDA 9.2
- NVIDIA Geforce GTX 1060 6GB
- Compute capacity: 6.1

Steps to run the program

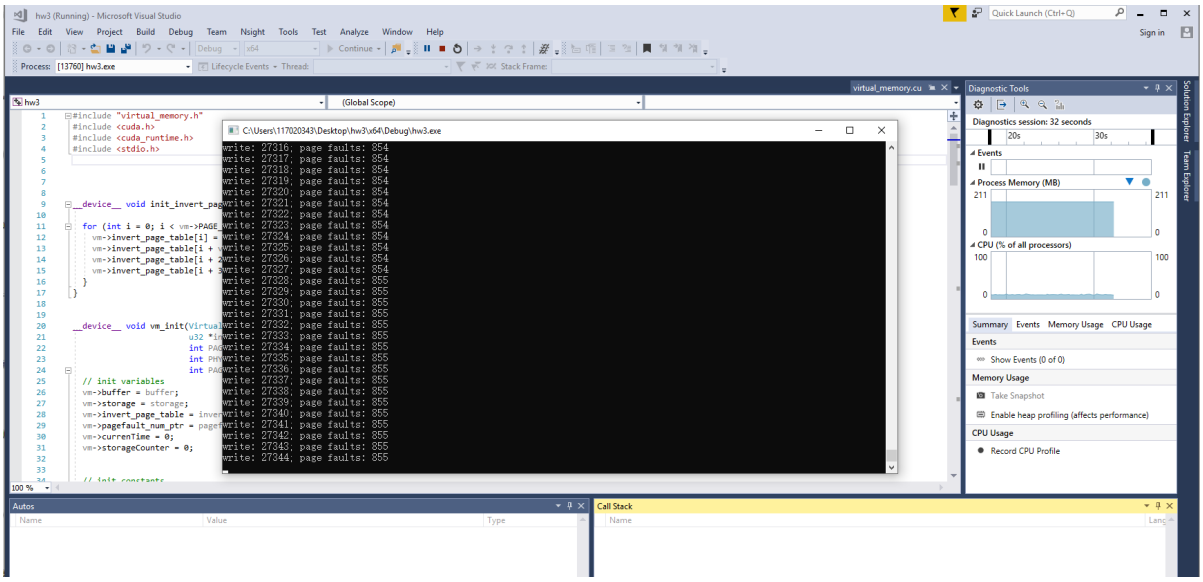
1. In the folder of '*hw3*', open '*hw3.sln*' with VS 2017:



2. In the VS 2017, select 'Debug' item in the menu and click 'Start Debugging' (or directly press F5):

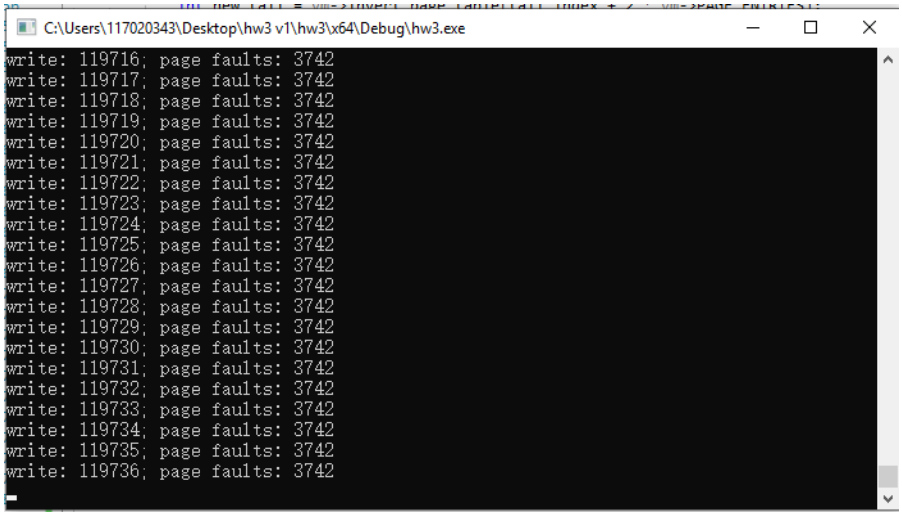


Then the program will start running:

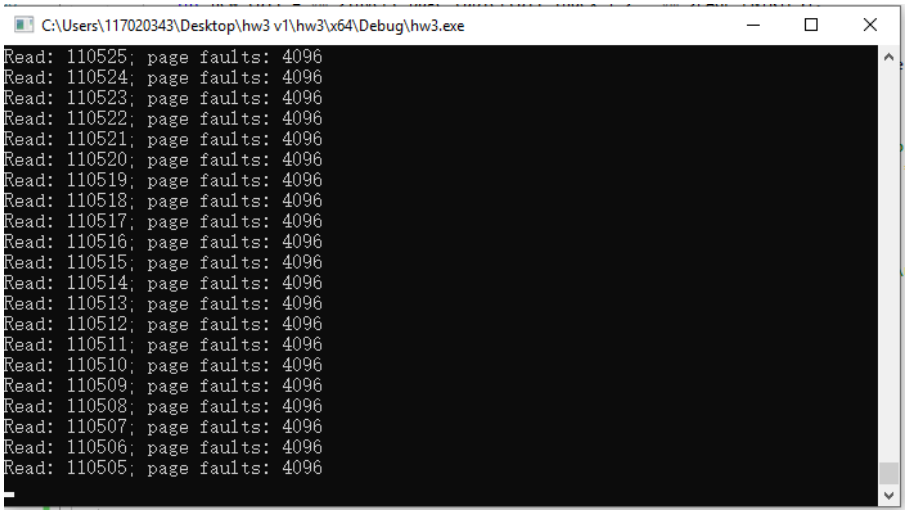


Sample output

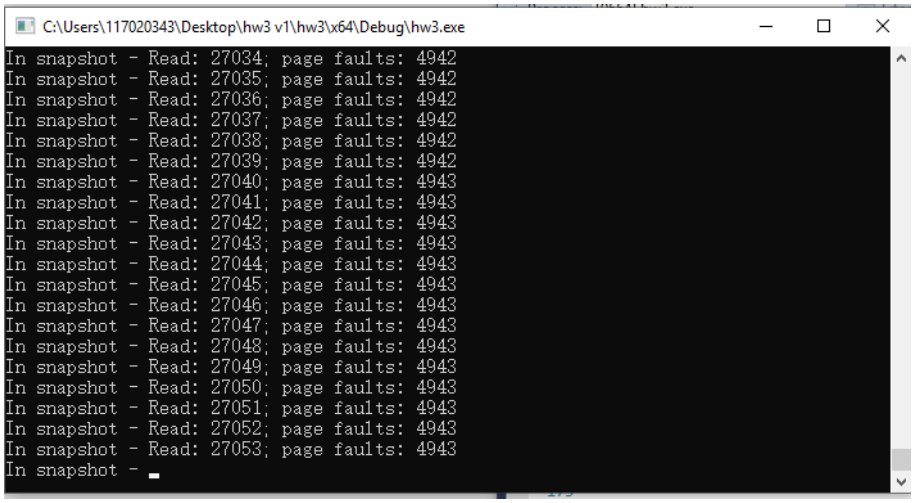
vm_write (part):



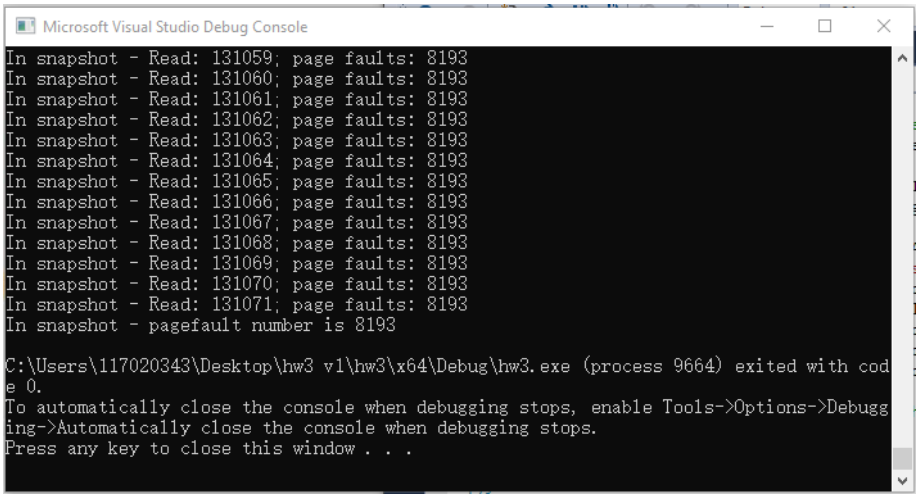
vm_read (part):



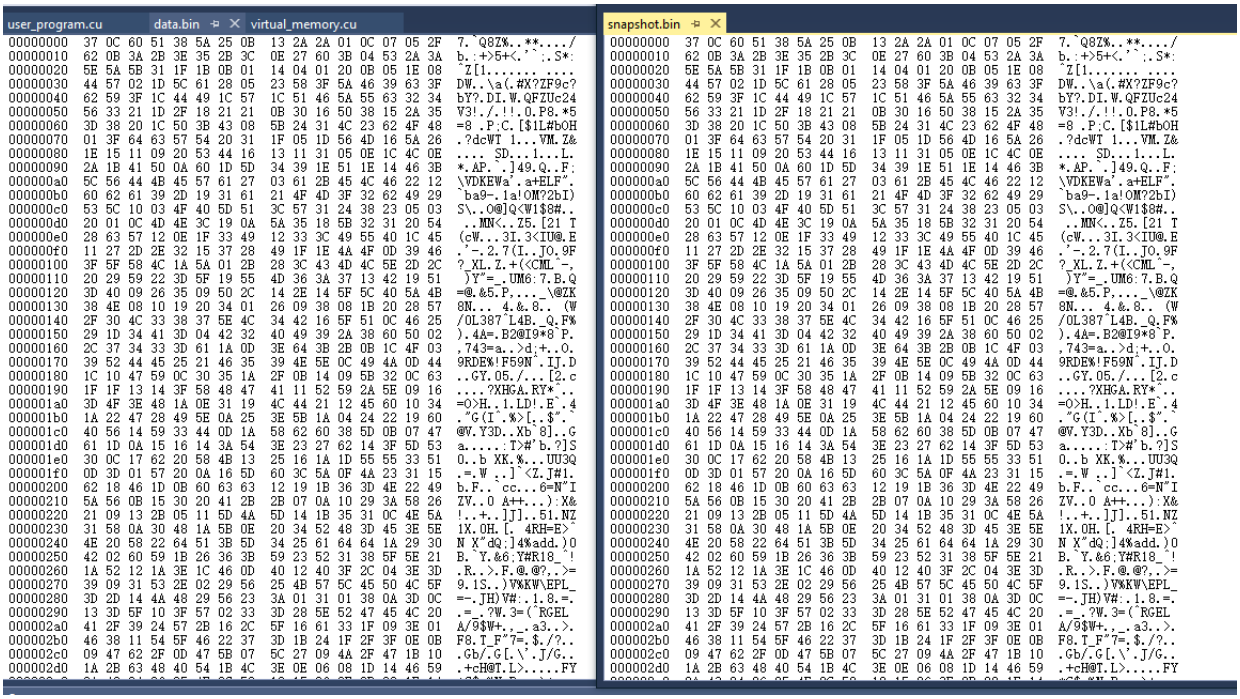
vm_snapshot (part):



Final output in console:



Output in file 'snapshot.bin', compared with input file 'data.bin' (part):



Page fault analysis

According to the final output in the console, the input size is 131072 (2^{17}) and the total page faults is **8193**.

```
__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 code above is the thet program in *user_program.cu*. By running this test program, it

1. writes on the virtual memory with 2^{17} unsigned characters from input buffer in order
2. reads the virtual memory for 32769 ($2^{15}+1$) unsigned characters in the reverse order
3. does the snapshot, which reads the virtual memory in order (same as 1) for 2^{17} unsigned characters.

Analyse each steps to find how page fault happened:

1. Initially, the page table is empty with 1024 available indexes and each index is corresponding to a page with 32 bytes. Since 1 uchar takes 1 byte, there are total 2^{17} bytes (4096 pages) written in this step. And there will not be two access on the same data address since the input is in order.
    1. For the first 1024 pages, each page just directly used a unused index to write the data into physical memory. Since using an new index caused 1 page fault, here total 1024 page faults were caused.
    2. After step 1, the page table was full, so the page replacement happened for the writing of the rest 3072 pages. Each page replacement caused 1 page, then total 3072 pages faults were caused.

    Therefore, for the 2^{17} times of *vm_write*, 4096 page faults happened.
2. After the *vm_write* of 4096 pages, the early 3072 pages have been swapped out to the storage (secondary memory), and the most recent 1024 pages are still in the physical memory. In this step, 32769 unsigned characters are read in reverse order, which consists of 1024 pages plus 1 byte.
    1. For the first 1024 pages, since the data was read in reverse order, the target data is the most recent data which is still in the physical memory. Thus the page table could directly access the target data and there is no page fault caused here.
    2. For the additional 1 single byte, it is not recent enough to be included in the page table, so the page replacement is required to swap the corresponding memory from storage into physical memory. This caused 1 page fault.

    Therefore, for the 32796 times of*vm_read*, 1 page fault happened.

3. For the *vm_snapshot*, the data was read in order for 2^{17} times. Since the first 1024 pages of data is the least recent indexed set, 1024 page replacements are required to get access to the data. After that, the same for the rest 3072 pages of data. A page replacement represents 1 page fault, so total 4096 page faults happend for the *vm_snapshot*.

Therefore, based on the above analysis, total $4096 + 1 + 4096 = 8193$ page faults is supposed to happen, which is the same as the counting result of the program.

Encountered problems

Problem

To implement the LRU algorithm, how to get the least recent used data set?

Solution

Observing that there is spare space in the memory for page table setting, I utilized that space to construct a simplified double linked list of indexes by recording the previous used index and the next used index for each index. This could function as a queue so that we could dequeue the least recent used indexed set when page replacement happens.

Problem

When debugging, the program exited unexpectedly.

Solution

Function *printf()* was used to keep track of the variables during program running and the output showed that all the variables were going well until the moment that the program exited. This may be caused by segment fault, which refers to the problem of memory overflow. I checked the code and found that, when transferring data to the physical memory, the base of the index was wrongly set as the base address of the input data. Finally I changed the base index to the corresponding index of the page table and the problem was solved.

What I learned

- Simple CUDA programming
  - Configuration of environment
  - CUDA memory hierarchy
  - Data transfer between memory
  - ...
- Mechanism of virtual memory
  - How does page table work
  - LRU algorithm of page replacement
  - Analysis based on page fault
  - ...

Bonus

Source code has been attached in folder "CSC3150_A3_Bonus"

☑ Launch 4 threads in kernel function

In *main()*, create 4 threads for kernel function *mykernel*:

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

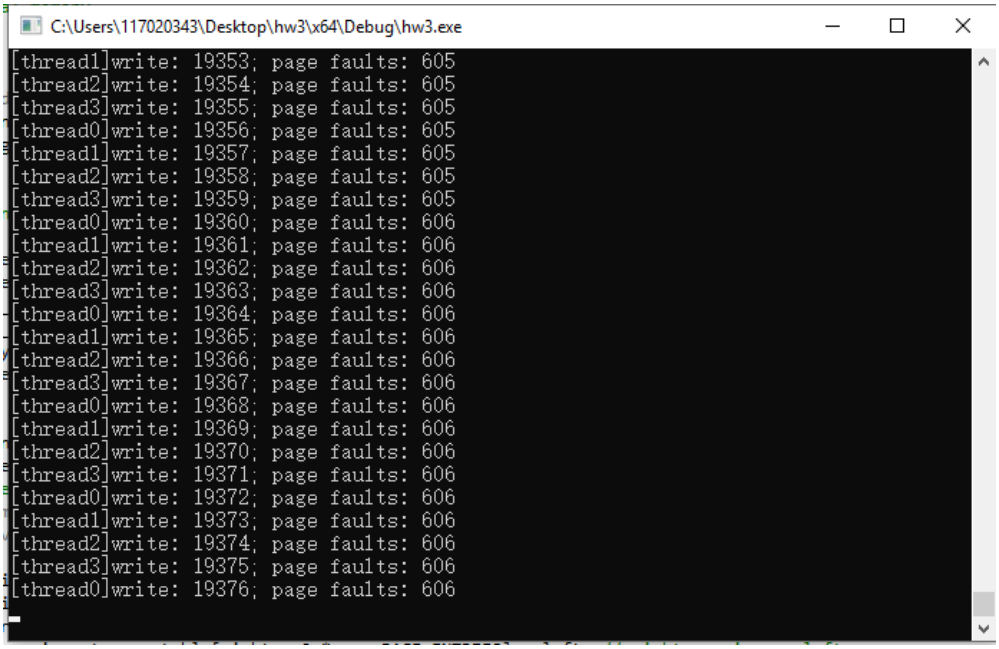
In function *mykernel*, identify the thread id and pass it into VM:

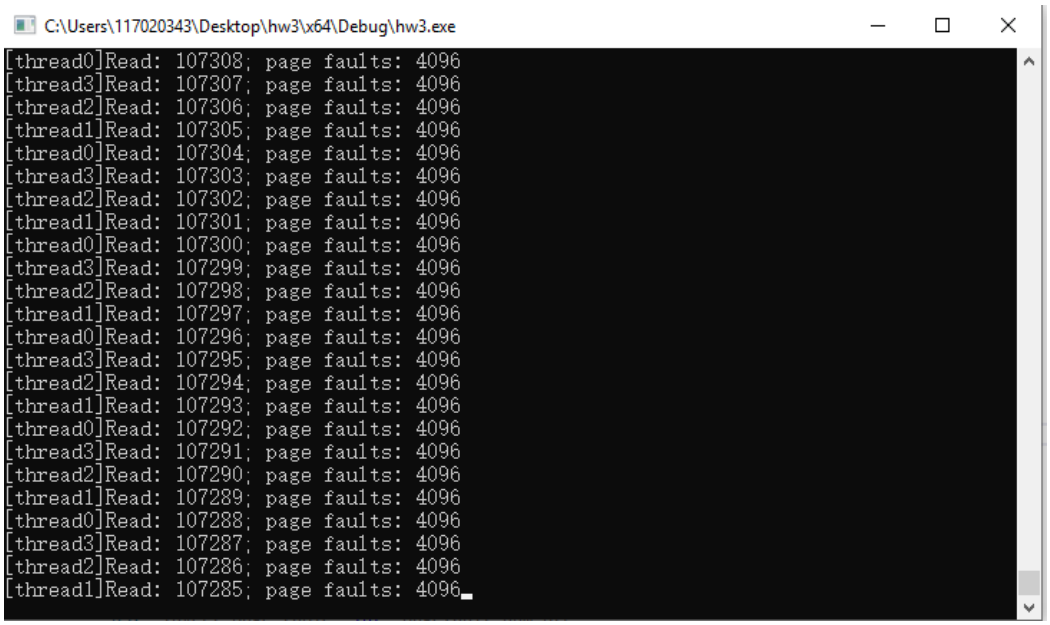
```
int i = blockDim.x * blockIdx.x + threadIdx.x;
```

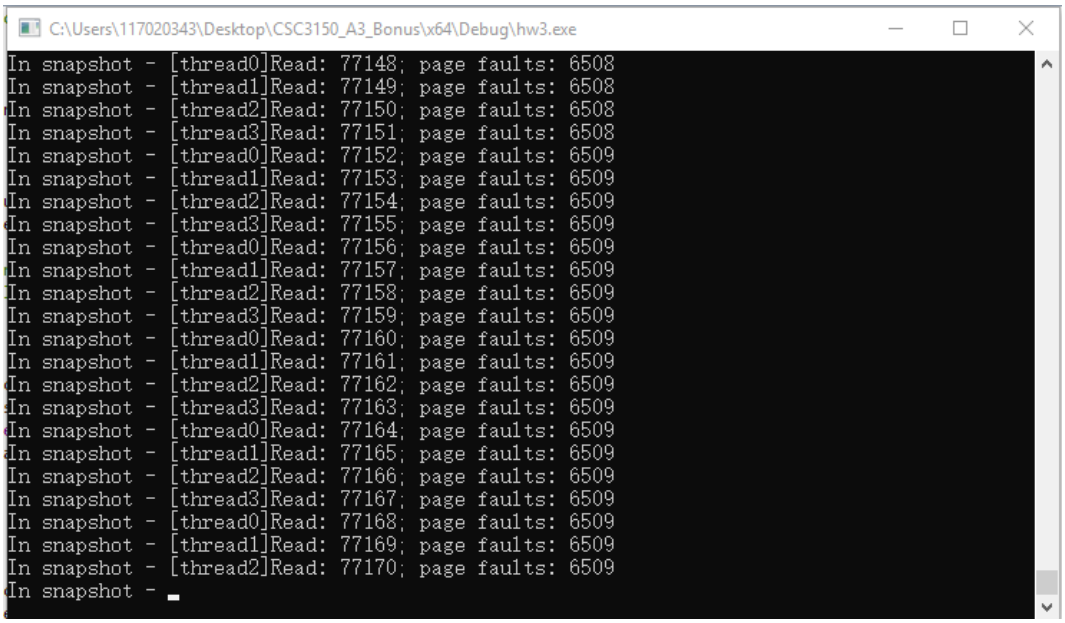
In each thread, for *vm_read, vm_write, vm_snapshot*, using thread id to identify the task, exit when the task is not the correct one:

```
if (addr % 4 != vm->tid) return ;
```

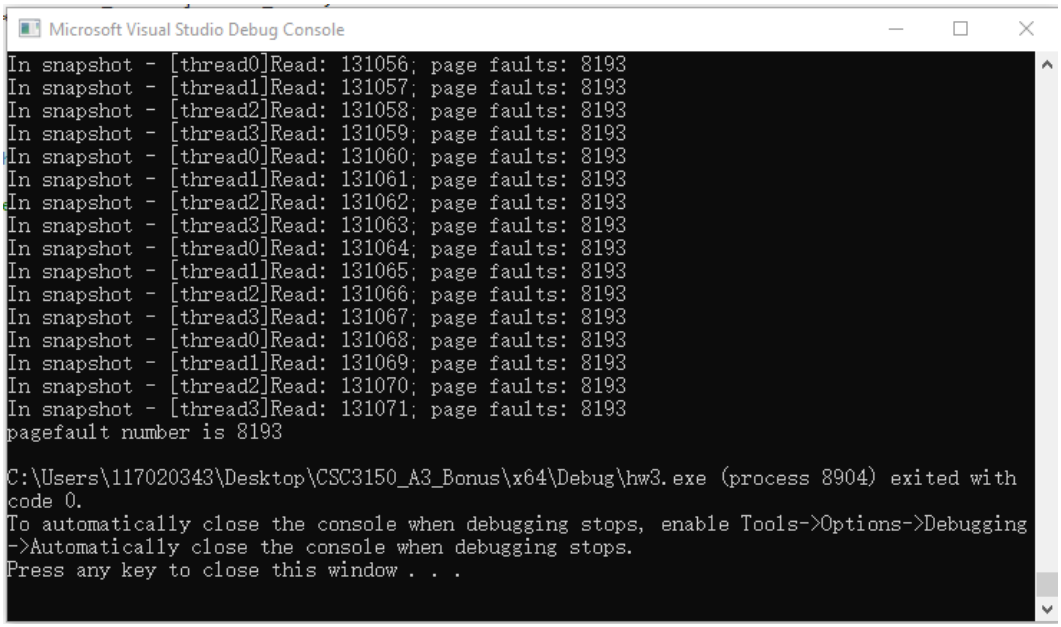
☑ Non-preemptive priority scheduling







☑ **Print the times of page fault of whole system before the program end**



- [x] **Modify paging mechanism to manage multiple threads.**

- [x] **Correctly dump the contents to “snapshot.bin”**