Virtual Memory Management Simulation

Yohandi [SID: 120040025]

Program Design

We were instructed to design a virtual memory management simulation for this task. As background information, virtual memory is a method that enables the execution of non-memory-resident processes. The fact that programs can be greater than physical memory is a significant advantage of this design. We were to implement a simple virtual memory in a GPU kernel function using a single thread, limited shared memory, and global memory for this project.

Each array in the program has a total of 1024 unsigned 32-bit integers. The first two arrays, namely `page_table` and `invert_page_table`, show whether or not a storage frame is in memory. Since each element has 32 bits of space, two storage frames can be represented. Consequently, each frame structure consists of 16 bits, with 4 bits assigned for the valid/invalid bit and the remaining 12 bits for the frame's page number in memory. Notice that 12 bits are already sufficient, given each frame runs from 0 to 2^{12}. With the help of the arrays, we can search the needs in the virtual memory.

Aside from those arrays, a second array is also constructed to store the storage address of each memory element. This array is used during the swapping operation to restore data from memory to storage. In this software, the last array serves as the doubly-linked list for LRU implementation. Each entry in this array represents the location of the memory buffer corresponding to the corresponding index. The first 16 bits of each element indicate the following element, whereas the remaining bits indicate the previous element.

As a template is already given, we only want to modify the `vm_read`, `vm_write`, and `vm_snapshot` functions from the provided template. Therefore, the file `virtual_memory.cu` will contain three primary functions, omitting the initiation code. As stated, those are `vm_read`, `vm_write`, and `vm_snapshots`.

vm_read

For `vm_read`, we need to partition it into two possible cases: the called address is in physical memory, or it is not in virtual memory. For the first case, we do not need to take into consideration the replacement of primary memory with secondary memory. Therefore, our application initially determined if the address could be located in the page table. If it is located, we can increase the page fault pointer. Moreover, we also want to save the index of the page table, which is also the frame address of the physical memory. Then, depending on the previous condition, the LRU is updated, and the output is returned. For the second case, however, we need to consider a condition whether there is an empty space in the memory or not. We update the flag and persist the index if there is an empty space. We then update the LRU, the page table, and the physical memory immediately. In comparison, the situation where physical memory is full will be somewhat more complicated: 1. The LRU table is examined to obtain the index of the frame that has been utilized the least. 2. Using the index, update the LRU table by setting the matching address and incrementing the remaining addresses, but not for the empty frame. 3. We save the removed data in the secondary memory. Transfer the needed information from secondary memory to physical memory and update the page table.

vm_write

For `vm_write`, Like with the read function, there will be two cases to think about. First, when the physical memory already has the address in question. The flow of the code is pretty similar to the read, except that the new value will be added to the requested address in the buffer. When the requested address is not in physical memory, the second case is the same as the read function. We also need to think about what happens if the physical memory still has room or is already full. If it still has empty space, you only need to update the LRU, the buffer, and the page table. There is no need to replace it. But if the space is full, we have to store the frame data from the last time it was used. Give the requested address to the physical memory and change the value. Last, make sure the page table and the LRU set are up to date.

vm_snapshots

The `vm_snapshots` function's only job is to put all the information from storage into the results array. In short, the program will loop as many times as the size of the input and perform the exact same procedure as the previously described `vm_read`. However, not only it runs the same commands at the same time, but it also saves the information to results. The results will later be printed into a single bin file named `snapshot.bin`.

Bonus

The task is divided into four threads (this question uses CUDA threads to simulate processes) to complete, that is, thread with $pid = 0$ is responsible for reading and writing $addr\%4 = 0$ task, thread with $pid = 1$ is responsible for reading and writing $addr\%4 = 1$ task, and so on. The four threads perform the same and whole process of testcase four times in total. This means that the addresses of the four threads are the same during read and write operations. We overwrite the next process with the content of the previous process.

Program Execution

Program Environment

Note that in order to execute the program, it is already assumed that the user has already set up an access to the school's computation resources.

Login to Cluster

- Open a new terminal and type `ssh {Student_ID}@CSC4005_cluster` to connect to the cluster remote login with SSH protocol.
- Enter a password to login.

Transfer Files to Cluster

- Open a new terminal and locate the files that are going to be transferred.
- Type `scp {file} {Student_ID}@CSC4005_cluster:.` In here, file can be in the format of `FILENAME.zip` if multiple files are to be sent. This will sent files directly to `nfsmnt/{Student_ID}` destination.

Execution Steps

- After login to cluster, extract the zip files if the files are zipped.
- It is required that in the destination there are exactly 6 files: `data.bin`, `main.cu`, `slurm.sh`, `user_program.cu`, `virtual_memory.cu`, and `virtual_memory.h`.
- To submit a batch, type `sbatch ./slurm.sh`. After execution is done, new files named `result.out` and `snapshot.bin` will be printed in the same destination. `result.out` shows the compile and run result.
- Another method is by typing both `nvcc --relocatable-device-code=true main.cu user_program.cu virtual_memory.cu -o test` to directly compile the CUDA script using nvcc compiler and `srun ./test` to run the compiled execution file. Note that, this step produces the same result with the previous step; however, `result.out` will not be printed as the content will be printed directly to the terminal.

Transfer Files from Cluster

- If the current terminal is still in the cluster root, type `exit` to exit from the cluster.
- Type `scp 120040025@CSC4005_cluster:~/{file} .` to retrieve the file back to the current folder.

Bonus

For bonus part, execution is performed exactly same with the normal part.

Result

When changing a test case, kindly edit the `user_program.cu` file and transfer it back to cluster.

Test Case #1

```c
// Test Case 1
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);
```

In this case, a number of 8193 pagefaults are expected. In first `vm_write` part, the number of `input_size` jobs, which is 2^{17}, take $\frac{2^{17}}{32} = 4096$ pages. Meaning 4096 page faults. After that, in the `vm_read` part, we realize that the size of physical memory is only 2^{15} bytes while `vm_read` is executed for $32769 = 2^{15} + 1$ different address. Meaning 1 extra page fault. In the last part, page fault will keep happening as `vm_snapshot` loops from 0 to `input_size` without any offset. This implies that another 4096 page faults are counted. This is also due to the physical memory that did not store the first 2^{15} data. Adding those results up, we have $4096 + 1 + 4096 = 8193$ page faults.

```
[120040025@node21 ~]$ srun ./test
input size: 131072
pagefault number is 8193
```

results

```
address   00 01 02 03 04 05 06 07  08 09 10 11 12 13 14 15   AscII  unsigned  bigendian
00000000  37 0c 60 51 38 5a 25 0b  13 2a 2a 01 0c 07 05 2f   7.`Q8Z%..**..../
00000010  62 0b 3a 2b 3e 35 2b 3c  0e 27 60 3b 04 53 2a 3a   b.:+>5+<.'`;.S*:
00000020  5e 5a 5b 31 1f 1b 0b 01  14 04 01 20 0b 05 1e 08   ^Z[1□□..□.. ..□.
00000030  44 57 02 1d 5c 61 28 05  23 58 3f 5a 46 39 63 3f   DW.□\a(.#X?ZF9c?
00000040  62 59 3f 1c 44 49 1c 57  1c 51 46 5a 55 63 32 34   bY?□DI□W□QFZUc24
00000050  56 33 21 1d 2f 18 21 21  0b 30 16 50 38 15 2a 35   V3!□/□!!.0□P8□*5
00000060  3d 38 20 1c 50 3b 43 08  5b 24 31 4c 23 62 4f 48   =8 □P;C.[$1L#bOH
00000070  01 3f 64 63 57 54 20 31  1f 05 1d 56 4d 16 5a 26   .?dcWT 1□.□VM□Z&
```

first 8 lines of snapshot.bin

```
0001ff80  62 44 64 33 1d 56 33 58  2c 5b 14 23 17 1e 20 1c   bDd3□V3X,[□#□□ □
0001ff90  05 30 4e 62 5c 29 32 32  55 61 63 31 23 5a 2b 55   .0Nb\)22Uac1#Z+U
0001ffa0  39 2a 23 55 50 26 19 4b  1c 60 09 02 1a 5c 52 52   9*#UP&□K□`..□\RR
0001ffb0  28 3b 4f 53 33 51 54 23  4d 53 24 40 18 4e 30 20   (;OS3QT#MS$@□N0
0001ffc0  48 22 44 03 17 2c 4d 33  28 56 04 41 1d 25 62 14   H"D.□,M3(V.A□%b□
0001ffd0  30 1c 03 62 08 26 55 55  48 14 64 5f 31 2f 1a 14   0□.b.&UUH□d_1/□□
0001ffe0  20 2e 16 37 59 33 39 1c  58 0c 2c 10 31 29 24 60    .□7Y39□X.,.1)$`
0001fff0  45 5a 2d 1c 4f 1d 0c 33  30 3f 2d 61 3d 17 44 5d   EZ-□O□.30?-a=□D]
```

last 8 lines of snapshot.bin

Test Case #2

```c
// Test Case 2
for (int i = 0; i < input_size; i++)
  vm_write(vm, 32*1024+i, input[i]);

for (int i = 0; i < 32*1023; i++)
  vm_write(vm, i, input[i+32*1024]);

vm_snapshot(vm, results, 32*1024, input_size);
```

In this case, a number of 9215 pagefaults are expected. In first `vm_write` part, the number of `input_size` jobs, which is 2^{17}, take $\frac{2^{17}}{32} = 4096$ pages. Meaning 4096 page faults. After that, in the second part, a number of 32×1023 `vm_write` are executed. This results in $\frac{32 \times 1023}{32} = 1023$ page faults. In the last part, page fault will keep happening as `vm_snapshot` loops from 0 to `input_size` with offset (loop over all contents written in the first part). This implies that another 4096 page faults are counted. Add those results up, we have an expected number of $4096 + 1023 + 4096 = 9215$ page faults.

```
[120040025@node21 ~]$ srun ./test
input size: 131072
pagefault number is 9215
```

results

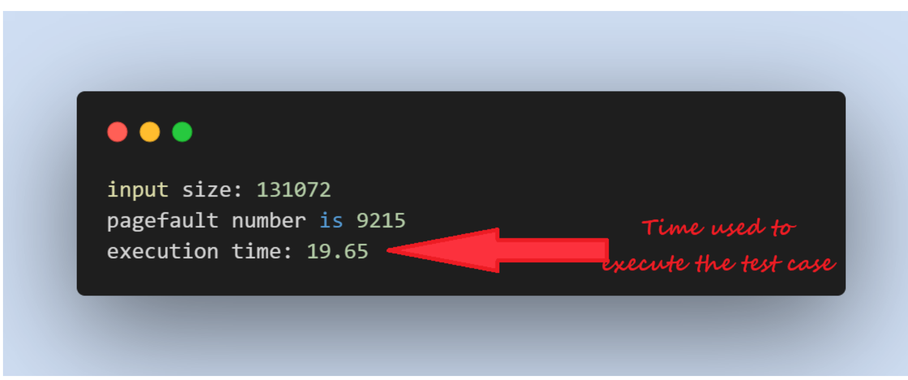
```
address   00 01 02 03 04 05 06 07  08 09 10 11 12 13 14 15   AscII  unsigned  bigendian
00000000  37 0c 60 51 38 5a 25 0b  13 2a 2a 01 0c 07 05 2f   7.`Q8Z%..**..../
00000010  62 0b 3a 2b 3e 35 2b 3c  0e 27 60 3b 04 53 2a 3a   b.:+>5+<.'`;.S*:
00000020  5e 5a 5b 31 1f 1b 0b 01  14 04 01 20 0b 05 1e 08   ^Z[1□□..□.. ..□.
00000030  44 57 02 1d 5c 61 28 05  23 58 3f 5a 46 39 63 3f   DW.□\a(.#X?ZF9c?
00000040  62 59 3f 1c 44 49 1c 57  1c 51 46 5a 55 63 32 34   bY?□DI□W□QFZUc24
00000050  56 33 21 1d 2f 18 21 21  0b 30 16 50 38 15 2a 35   V3!□/□!!.0□P8□*5
00000060  3d 38 20 1c 50 3b 43 08  5b 24 31 4c 23 62 4f 48   =8 □P;C.[$1L#bOH
00000070  01 3f 64 63 57 54 20 31  1f 05 1d 56 4d 16 5a 26   .?dcWT 1□.□VM□Z&
```

first 8 lines of snapshot.bin

```
0001ff80  62 44 64 33 1d 56 33 58  2c 5b 14 23 17 1e 20 1c   bDd3□V3X,[□#□□ □
0001ff90  05 30 4e 62 5c 29 32 32  55 61 63 31 23 5a 2b 55   .0Nb\)22Uac1#Z+U
0001ffa0  39 2a 23 55 50 26 19 4b  1c 60 09 02 1a 5c 52 52   9*#UP&□K□`..□\RR
0001ffb0  28 3b 4f 53 33 51 54 23  4d 53 24 40 18 4e 30 20   (;OS3QT#MS$@□N0
0001ffc0  48 22 44 03 17 2c 4d 33  28 56 04 41 1d 25 62 14   H"D.□,M3(V.A□%b□
0001ffd0  30 1c 03 62 08 26 55 55  48 14 64 5f 31 2f 1a 14   0□.b.&UUH□d_1/□□
0001ffe0  20 2e 16 37 59 33 39 1c  58 0c 2c 10 31 29 24 60    .□7Y39□X.,.1)$`
0001fff0  45 5a 2d 1c 4f 1d 0c 33  30 3f 2d 61 3d 17 44 5d   EZ-□O□.30?-a=□D]
```

last 8 lines of snapshot.bin

Bonus



Reflection

This task has enhanced my understanding of the computer's paging system. Initially, we were taught CUDA programming. Second, we studied how memory functions in computers, particularly how primary memory and secondary memory function. This is the primary objective of the task. We discovered that the memory employs a page table to maintain track of the connections between physical and secondary storage. On the other hand, we better understand what occurs when we write miss, write hit, read miss, and read hit. When a read or write error occurs, the memory must be replaced. Finally, we discover that there is a technique named Least Recently Used. This approach is used in main memory to determine which data has been accessed the least frequently so that it can be switched out for other requested data.