Report for CSC3150 assignment 3

Feng Yutong 120090266

Design

- Intro

  This program implements the memory management. It supports translation between logical memory and physical memory, physical memory and disk memory, page table, swap table and RLU.

  This program use CUDA programming. Bonus support multi-threads.

- Implementation
  - Abstract Data Type representation
    - Logical Memory Adrress (160k): consisting of physical memory (32k) and disk memory (128k)
    - Disk memory (128k): `vm->storage`
    - Physical memory (32k): `vm->buffer`
    - Inverted page table (16k): `vm->invert_page_table` (actual use: 8k)
    - LRU stack: a stack implemented by link, with its head pointing to the least recent used page, tail pointing the most recent used page
    - Swap table (160k/32b): store the location of swapped out page
  - Theoretical time complexiy
    - Page table lookup: O(L)
    - LRU stack look up and update: O(1)
    - Swap page: O(1)
  - Function implement
    - `vm_init`
      - Initialize page table
      - Initialize LRU table
    - `LRU_get`
      - Return the index (frame index in this design) of the least recent used page
      - Update the index to stack top (the same index will be used for the page coming in)
    - `swap_in_page(page_number)`
      - Pop out the queried page location
    - `swap_out_page(LRU_page_number, location)`
      - Update the LRU page location in disk storage

- `lookup_pt(page_number`
  - Return the frame index of a given page number
  - Return 0x80000000 if not found in page table
- `vm_read`: read single element from main memory
  1. Lookup page table. If in page table, return the value stored in corresponding buffer address.
  2. If not in page table, use `LRU_get` to get the least recent used page. Then call `swap_in_page(page_number)` get the disk location of the page, swap value stored in disk and main memory and call `swap_out_page(LRU_page_number, location)` to swap out the LRU page to disk to the location.
  3. Update page table and LRU table.
  4. Repeat step 1.
- `vm_write`: write value to main memory
  1. Lookup page table. If in page table, write the value in the corresponding buffer address.
  2. If not in page table, use `LRU_get` to get the least recent used page. Then call `swap_in_page(page_number)` get the disk location of the page, swap value stored in disk and main memory and call `swap_out_page(LRU_page_number, location)` to swap out the LRU page to disk to the location.
  3. Update page table and LRU table.
  4. Repeat step 1.
- `vm_snapshot`
  - Call `vm_read` to dump value in results
- core part of `vm_write` and `vm_read`

```
int pn = addr >> 5, offset = addr & 31;
int ret = lookup_pt(vm, pn);
if(ret == 0x80000000){ //not found

   (*(vm->pagefault_num_ptr))++;

  int idx = LRU_get(vm); // idx of PT
  u32 replace_page = vm->invert_page_table[idx]; // frame addr

  u32 pos_in = swap_in_page(vm, pn);

  for(int i = 0; i < 32; i++) {
      uchar tmp = vm->buffer[(replace_page << 5) + i];
      vm->buffer[(replace_page << 5) + i] = vm->storage[(pos_in << 5) + i];
      vm->storage[(pos_in << 5) + i] = tmp;
    }

  swap_out_page(vm, vm->invert_page_table[idx + vm->PAGE_ENTRIES], pos_in);
  vm->invert_page_table[idx + vm->PAGE_ENTRIES] = pn;
}
```

- 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 page_number%4==0 task, thread with pid==1 is responsible for reading and writing page_number%4==1 task, and so on. (Since the pragram swap data in unit of page, so here use page_number instead of address). This implementation is easy, just set vm->thread_id and check at the beginning of `vm_write` and `vm_read`, also modify `vm_snapshot` to only read corresponding page.

Environment and execution

- Environment
  - OS: CentOS Linux release 7.5.1804 (Core)
  - NVIDIA-SMI 515.65.01
  - Driver Version: 515.65.01
  - CUDA Version: 11.7
- Execution Modify `user_program` and run `bash slurm.sh`. It will print out page fault number in terminal and dump a binary file `snapshot.bin`. Use vim to check for correctness. Same is for bonus.

Output

- For given test1
  - Sample

```
for (int i = 0; i < input_size; i++)
        vm_write(vm, i, input[i]);
printf("write page fault: %d\n",*(vm->pagefault_num_ptr));
for (int i = input_size - 1; i >= input_size - 32769; i--) {
    int value = vm_read(vm, i);
}
printf("read page fault: %d\n", *(vm->pagefault_num_ptr));

vm_snapshot(vm, results, 0, input_size);
printf("snapshot page fault: %d\n", *(vm->pagefault_num_ptr));
```

  - Output

```
input size: 131072
write page fault: 4096
read page fault: 4097
snapshot page fault: 8193
pagefault number is 8193
```

    We can see that the page fault number is 4096 + 1 + 4096 = 8193

- For given test2

- Sample

```
for(int i = 0; i < input_size; i++)
        vm_write(vm, 32*1024+i, input[i]);
printf("write page fault: %d\n",*(vm->pagefault_num_ptr));

  for(int i = 0; i < 32*1023; i++)
        vm_write(vm, i, input[i+32*1024]);
printf("write page fault: %d\n",*(vm->pagefault_num_ptr));

  vm_snapshot(vm,results,32*1024,input_size);
printf("snapshot page fault: %d\n", *(vm->pagefault_num_ptr));
```

- Output

```
input size: 131072
write page fault: 4096
write page fault: 5119
snapshot page fault: 9215
pagefault number is 9215
```

We can see that the page fault number is 4096 + 1023 + 4096 = 9215 `snashot.bin` is the same with `data.bin` for two cases.

- bonus Same output as program since it's first version design

Learning

When implementing this project, I first confused the logical memory, main memory with disk memory. Then I learned that logical memory can be seen as a combination of the latter two memory in tutorial. Thus, I cannot use 16k page table to record all physical address of a given page. I build a swap table. At the beginning, I used a data structure similar to vector to implement the update of swap table(when swapping in, put in; when swapping out, put out), however, the time complexiy is very high since page number can be up to 5120. Thus, I use a array to store the location. If page is not in disk memory, then set -1. The index of the location is page number. This reduced to O(1) The order of swap also needs paying attention. Actually, a page table storing `PAGE_ENTRIES` (1024) can just uses 8K or less (near 4k): only store page number and valid bit, with index of the array as frame index. This program uses 8k for the valid bit is not a bit but a unit32_t, 0x80000000 as invalid and frame index as valid.

In addition, I first used counter to implement LRU, then it takes O(page_table_size) to update the counter. Therefore, I implement a LRU stack. Each time a page is read/write, I add it to the top (vm->tail), LRU page can be get at bottom (vm->head), then delete it. It is support by a data structure `LRU_node` which has a `LRU_node` nxt pointer and an index.

For future improvement, we can build a hash page table to achieve O(1) lookup. Due to time limit, I just use a inverted page table.