CSC3150 proj3 report

Zhouliang Yu

November 2021

1 Environment

I use the system CentOS in TC301, with default editor
CUDA version 10.1.105
NVDIA GeForce GTX 1060 6GB

2 how to compile

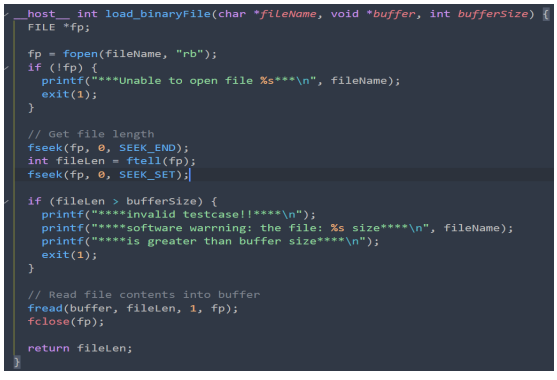
in my program I write a makefile. so users can simply compile by type "make" and clean the ".o" ".exe" file by "make clean"

users are able to run my program by type "./vm" in command line

3 How to Design My Program

the whole process of my basic part program is:

3.1 load data from the binary data fiel and input buffer



```
__host__ int load_binaryFile(char *fileName, void *buffer, int bufferSize) {
  FILE *fp;

  fp = fopen(fileName, "rb");
  if (!fp) {
    printf("***Unable to open file %s***\n", fileName);
    exit(1);
  }

  // Get file length
  fseek(fp, 0, SEEK_END);
  int fileLen = ftell(fp);
  fseek(fp, 0, SEEK_SET);

  if (fileLen > bufferSize) {
    printf("****invalid testcase!!****\n");
    printf("****software warrning: the file: %s size****\n", fileName);
    printf("****is greater than buffer size****\n");
    exit(1);
  }

  // Read file contents into buffer
  fread(buffer, fileLen, 1, fp);
  fclose(fp);

  return fileLen;
}
```

3.2 vm_write function

Use the write function to write the data from the input buffer to the physical memory. If the physical memory is full,we will use the LRU mechanism to decide which page should be removed from the physical memory to the disk storage and put the current page into that frame. actually in this senerio the least least recent used one is the one at the top: at the slot: PAGE_ENTRIES

```
__device__ void vm_write(VirtualMemory *vm, u32 addr, uchar value) {
  /* Complete vm_write function to write value into data buffer */
  u32 page_num = addr / 32;
  u32 page_offset = addr % 32;
  u32 frame_num;

paging_for_write(vm,  page_num,  page_offset,  frame_num, value);
```

```
__device__ void paging_for_write(VirtualMemory *vm, u32 page_num, u32 page_offset, u32 frame_num, u32 value){
  /** check page fault*/
  if(page_fault(vm, page_num)) {
    frame_num = vm->invert_page_table[vm->PAGE_ENTRIES];

  /** check if frame is full*/
if (vm->invert_page_table[frame_num] != 0x80000000)
  {
    /** if is full move to the storage*/
  move_to_storage(vm, frame_num);
  }


    vm->invert_page_table[frame_num] = page_num;
  }
  else{
    frame_num = frame_number_page(vm, page_num);
  }
  vm->buffer[frame_num * 32 + page_offset] = value;

  /** change from invalid to valid*/
  int frame_id = find_frame_num_frame_table(vm, frame_num);
    int tempt = vm->invert_page_table[vm->PAGE_ENTRIES + frame_id];
  for (int i = frame_id; i < vm -> PAGE_ENTRIES - 1; i ++) {
    vm->invert_page_table[i + vm->PAGE_ENTRIES] = vm->invert_page_table[i + vm->PAGE_ENTRIES + 1];
  }
  vm -> invert_page_table[2 * vm->PAGE_ENTRIES - 1] = tempt;
}
```

paging_for_write function we first check if there is page fault in the page table. if yes then we set the frame number as the the top one in the pagenumber area. than we check wether the frame table is full by checking wether the corrsponding slot of frame number is equal to 0x80000000 if it is full we move the elemnt from physical memory to storage because the top one is the least recent used one. else we find the corresponding pagenumber. in the end we write the value into the buffer and change the value of corresponding page number to be valid.

3.3 vm_read function

first find the required pages in the physical memory, if the target page is not in the physical memory we will use LRU to swap the page in storage with the least recent used slot in physical memory

```
__device__ uchar vm_read(VirtualMemory *vm, u32 addr)
{
  /* Complate vm_read function to read single element from data buffer */
  u32 page_num = addr / 32;
  u32 page_offset = addr % 32;
  u32 frame_num;

  paging_for_read(vm,  page_num,  page_offset,  frame_num);
```

```
__device__ void paging_for_read(VirtualMemory *vm, u32 page_num, u32 page_offset, u32 frame_
/** check page fault*/
if (page_fault(vm, page_num)) {
    frame_num = vm -> invert_page_table[vm->PAGE_ENTRIES];

    /** move to memory*/
    u32 original_page_num = vm->invert_page_table[frame_num];
    for (int i = 0; i < 32; i ++) {
    vm->storage[original_page_num * 32 + i] = vm -> buffer[frame_num * 32 + i];
    vm->buffer[frame_num * 32 + i] = vm -> storage[page_num * 32 + i];
  }
  vm -> invert_page_table[frame_num] = page_num;
  }else {
    frame_num = frame_number_page(vm, page_num);
  }

int frame_id = find_frame_num_frame_table(vm, frame_num);
int tempt = vm->invert_page_table[vm->PAGE_ENTRIES + frame_id];
  for (int i = frame_id; i < vm -> PAGE_ENTRIES - 1; i ++) {
    vm->invert_page_table[i + vm->PAGE_ENTRIES] = vm->invert_page_table[i + vm->PAGE_ENTRIES
  }
  vm -> invert_page_table[2 * vm->PAGE_ENTRIES - 1] = tempt;

}
```

paging_for_read function we first check if there is page fault in the page table. if yes then we set the frame number as the the top one in the pagenumber area. if there is page fault then we move the data from the storage to the physical memory according to the given page and frame numbers

else we find the corresponding frame number according to the given page number. It is implemented by searching through the page table and check if the

page number storage in each entry of the page table is equal to the given page number. If found, return the frame number

then we change the valid bit from invalid to valid

3.4 vm_snapshot

dump the contents to snapshot it load the elements of virtual memory buffer to results buffer

for each iteration if there is page fault then we move the data from the storage to the physical memory according to the given page and frame numbers

else we find the frame number in pagetable

then we move tge result to the result buffer, and change the valid bit from invalid to valid

```
__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 i = 0; i < input_size; i++){
    u32 page_num = i /32;
    u32 frame_offset = i % 32;
    u32 frame_num;

paging_snapshot(vm, results,  offset, input_size,  page_num,  frame_num,  frame_offset);

  }
}
```

```
__device__ void paging_snapshot(VirtualMemory *vm, uchar *results, int offset,
                          int input_size, int page_num, int frame_num, int frame_offset){
if (page_fault(vm, page_num)) {
      frame_num = vm -> invert_page_table[vm -> PAGE_ENTRIES];
      u32 original_page_num = vm->invert_page_table[frame_num];
    for (int i = 0; i < 32; i ++) {
    vm->storage[original_page_num * 32 + i] = vm -> buffer[frame_num * 32 + i];
    vm->buffer[frame_num * 32 + i] = vm -> storage[page_num * 32 + i];
  }
  vm -> invert_page_table[frame_num] = page_num;
    }else{
      frame_num = frame_number_page(vm, page_num);
    }

/** move to the results buffer*/
 for (int i = 0; i < 32; i++){
    results[page_num * 32 + i] = vm -> buffer[frame_num * 32 + i]; // load element from vm buffer to re
  }

/** change from invalid to valid*/
int frame_id = find_frame_num_frame_table(vm, frame_num);
    int tempt = vm->invert_page_table[vm->PAGE_ENTRIES + frame_id];
  for (int i = frame_id; i < vm -> PAGE_ENTRIES - 1; i ++) {
    vm->invert_page_table[i + vm->PAGE_ENTRIES] = vm->invert_page_table[i + vm->PAGE_ENTRIES + 1];
  }
  vm -> invert_page_table[2 * vm->PAGE_ENTRIES - 1] = tempt;


}
```

3.5 move_to_storage

according to the given frame number move from the physical memory to storage, use in process of swap

```
__device__ void move_to_storage(VirtualMemory *vm, u32 frame_num){
  u32 page_num = vm -> invert_page_table[frame_num];
  for (int i = 0; i < 32; i ++) {
    vm->storage[page_num * 32 + i] = vm->buffer[frame_num * 32 + i];
  }
}
```

3.6 find_frame_num_frame_table

as the top entry will store the frame number of the least recently used element and the bottom entry will store the frame number of the most recently used one. we are going to find the frame number in the frame table linearly one by one

```
int frame_id = find_frame_num_frame_table(vm, frame_num);
int tempt = vm->invert_page_table[vm->PAGE_ENTRIES + frame_id];
  for (int i = frame_id; i < vm -> PAGE_ENTRIES - 1; i ++) {
    vm->invert_page_table[i + vm->PAGE_ENTRIES] = vm->invert_page_table[i + vm->PAGE_ENTRIES + 1];
  }
  vm -> invert_page_table[2 * vm->PAGE_ENTRIES - 1] = tempt;

}
```

4 Output and Explanation

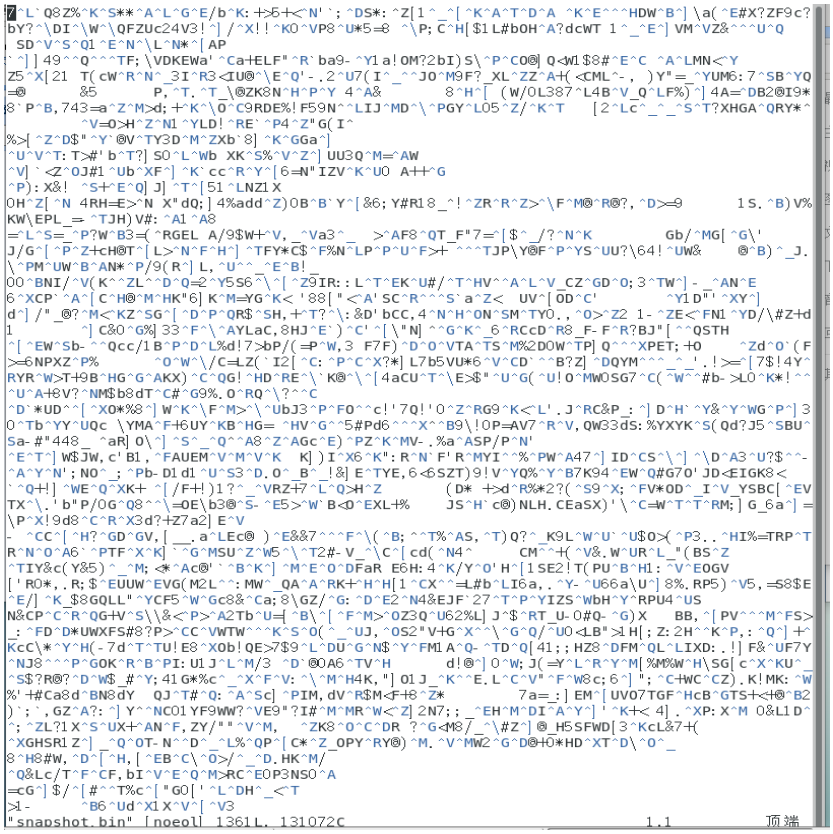
the input file is like:

4.1 the output

1. the pagefault number is:

```
nvcc -arch=sm_30 user_program.cu -rdc=true -c
nvcc -arch=sm_30 main.o virtual_memory.o user_program.o -rdc=true -o vm
(base) [cuhksz@TC-301-27 77]$ ./vm
input size: 131072
pagefault number is 8193
(base) [cuhksz@TC-301-27 77]$ cmp data.bin snapshot.bin
(base) [cuhksz@TC-301-27 77]$
```

2. the snapshot file is:



4.2 Why 8193 pagefaults

in write section there are 4096 fault since every page loaded from the input buffer acts as a new page for the physical memory and since there are 128K data with 32 bytes pagesize in each page, by calculation we can have 4096 page faults because there are 4096 pages

in read section there are 1 fault there are totally 32769 bytes read in the physical memory in the user program. so by simple calculation we have 1024 pages and 1 byte because each page contains 32 bytes. meanwhile as the data is written from the top to the bottom, the first 1024 pages will be found in the physical memory. so only the last bytes will generate page fault.

in the snapshot section the page fault is 4096 because the data are read from the top to the bottom. each one will generate fault so there are 4096 faults

so overall 8193 page faults will be generated.

5 Problem I faced and how to solve them

5.1 How to deal the case when physical memory is full

I had no idea at first How to deal with the case when physical memory is full

then I realize when physical memory is full, I can use the LRU mechanism to decide which page should be removed from the physical memory to the disk storage and put the current page into that frame. It's a process of swap to ensure new element can be added and the LRU can be removed

in my program I use LRU in both read and write process

5.2 How to get page number for data store in storage area

I confused about this process. Then I solved it by going to the storage search for the page number one by one util we find the right one

5.3 how to check if physical memory is full

by checking wether the corrsponding slot of frame number is equal to 0x80000000

6 What I have learnt

6.1 memory management

we divid our memory into global memory, shared memory. when we read the data we first search the page table to see if the corresponding page is in the physical memory or not. if the answer is yes, then we simply claim that the frame number of data is the index of that corresponding page number in inverted page table if not we use LRU to swap the page in the storge area with least recent used one in physical memory

6.2 process of page replacement in the program design

we can swap the least used page out of shared memory and swap it to the secondary storage then we swap the page into share memory for writting data

then we update the page table

when reading data from vuffer is the page exists in shared memory read it out otherwise replase the LRU set and swap the lru page out of shared memory and swap it in secondary storage then swap the designed page into shared memory for data reading

in the end we update the page table

7 bonus

7.1 how_to_execute

in my program I write a makefile. so users can simply compile by type "make" and clean the ".o" ".exe" file by "make clean"

7.2 output

```
cuhksz@TC-301-39 77]$ ./vm
nput size: 131072
he times of page table is:31.900000
agefault number is 1025
cuhksz@TC-301-39 77]$
```

7.3 launch 4 threads in kernel function, all threads concurrently execute it.

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

cudaStatus = cudaGetLastError();
if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "mykernel launch failed: %s\n",
          cudaGetErrorString(cudaStatus));
  return 0;
}
```

7.4 To avoid the race condition, threads execute vmread() vmwrite() should be a nonpreemptive priority scheduling, the priority of threads thread1 thread2 thread3 thread4.

```
__global__ void mykernel(int input_size) {

  // memory allocation for virtual_memory
  // take shared memory as physical memory
  __shared__ uchar data[PHYSICAL_MEM_SIZE];

  VirtualMemory vm;
  vm_init(&vm, data, storage, pt, &pagefault_num, PAGE_SIZE,
          INVERT_PAGE_TABLE_SIZE, PHYSICAL_MEM_SIZE, STORAGE_SIZE,
          PHYSICAL_MEM_SIZE / PAGE_SIZE);
  if(threadIdx.x == 0) {
    init_invert_page_table(&vm);
  }

 /** calling the __synthreads() instrincfunction to specify the synchronization points
  * __synthreads acts as barrier at which threads in the block must wait before any is
  LOCK();

  // call the write function
  for (int i = 0; i < input_size; i++) {
    vm_write(&vm, i, input[i]);
  }
  UNLOCK();

  //read
  for (int i = input_size - 1; i >= input_size - 32769; i--){
    LOCK();
    int value = vm_read(&vm, i);
    UNLOCK();
  }

  LOCK();
  vm_snapshot(&vm, results, 0, input_size);
  UNLOCK();
  return;
}
```

7.5 print times of page fault

```
  t = clock() - t;
  printf("the times of page table is:%f\n", ((float)t) / CLOCKS_PER_SEC);
  printf("pagefault number is %d\n", pagefault_num);

  return 0;
}
```