Additional notes on Assignment 3

Q: How big is the virtual memory? The address in the test case is 128KB max, does it mean that our memory and addressing only uses 128KB of supported virtual memory?

A: The test case provided to you is simple and only writes to 128KB, but the physical memory + secondary memory has 160KB in total, so please support 160KB addressing space. Our grading test cases will assume that you support a 160KB virtual memory.

Q: Can we make a fixed mapping between pages in virtual memory and secondary memory to facilitate swap operation in case of write and read page faults?

A: This may not pass in some tests. In fact, Linux, for example, maintains a space in the swap partition, which records the history of swaps. To simplify your implementation, you could use an additional swap table and note the vpn and corresponding disk location each time a swap happens. When a read page fault occurs, take the page num and go to the swap table to find out where the last swap was to. If you can't find it in the swap table, just return an error.

Q: Where does the Swap table occupy space?

A: In the real world, it is in the disk. In this question, to ease the work of your calculation, the swap table can locate on the disk alone and does not occupy secondary memory, which means that all secondary memory (128KB) is still fully used for data swap.

Q: Can we have more test cases?

A: Here is a case that (1) write the data.bin to the VM starting from address 32*1024; (2) write (32KB-32B) data to the VM starting from 0; (3) read out VM[32K, 160K] and output the content to snapshot.bin, which should be the same with data.bin. And the page fault number should be 4096+1023+4096=9215

```
// test case 2

__device__ void user_program(VirtualMemory *vm, uchar *input, uchar *results,

  int input_size) {
```

```
// write the data.bin to the VM starting from address 32*1024

for (int i = 0; i < input_size; i++)

   vm_write(vm, 32*1024+i, input[i]);

// write (32KB-32B) data   to the VM starting from 0

for (int i = 0; i < 32*1023; i++)

   vm_write(vm, i, input[i+32*1024]);

// readout VM[32K, 160K] and output to snapshot.bin, which should be the same with data.bin

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

}
```

The test case can be found in BB now.

About Bonus

After these days of discussion with some of you about the bonuses, including but not limited to the following understandings and implementations will be considered correct and give points.

We first recall that the basic part of test case1, is to first write into memory 128KB data.bin, then read 32KB+1B data, and finally snapshot out the content of the memory at address 0-128K, i.e. snapshot.bin should be consistent with data.bin.

Version 1: 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.

Version 2: The four threads perform the same and whole process of testcase1 four times in total. This means that the addresses of the four threads are the same during read and write operations. (The same task is done four times by four threads) In this version, the next process overwrites the content of the previous process.

Version 3: As in version 2, four threads perform the same operation four times, but the disk is scaled to 512KB so that the four threads can write on different pages of the virtual machine without overwriting each other. When the final snapshot is made, the result buffer stores the 128KB written by the last thread.

The page number will be up to your implementation.

Collection of QA

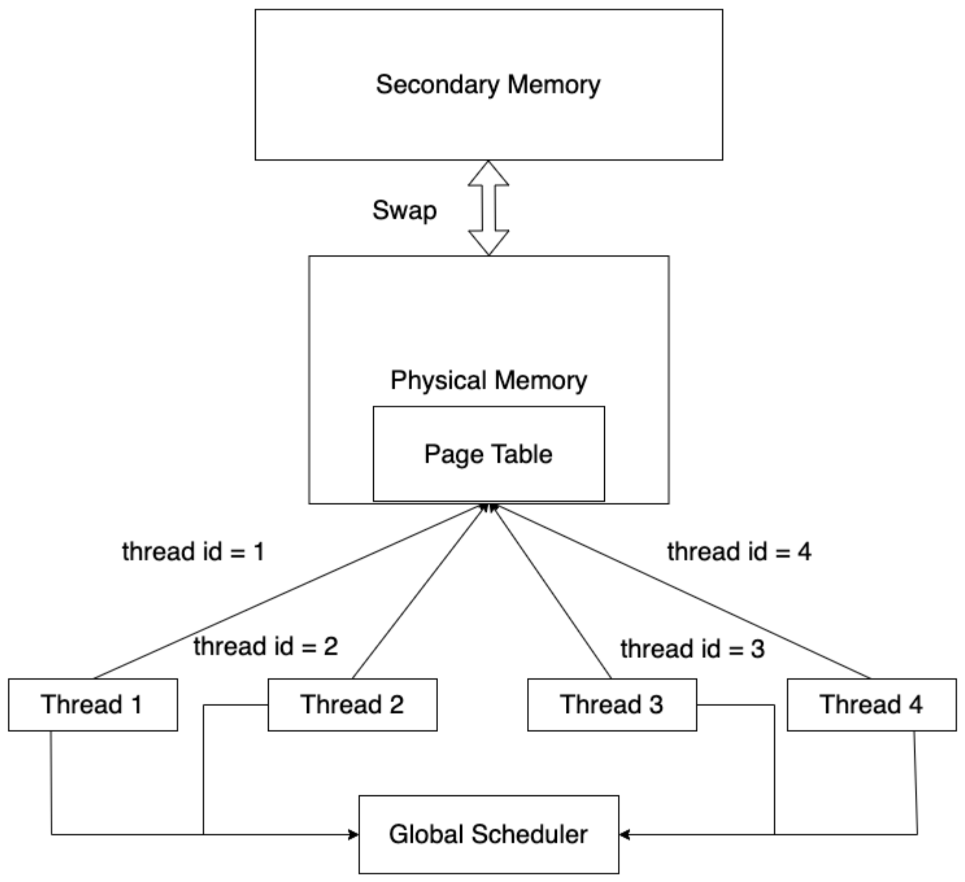
Q: About how to support multi-threads:
One possible tip: you can add a pointer to the vm structure that is placed in global memory and points to the next thread to be run. You can use a global scheduler placed in global memory.

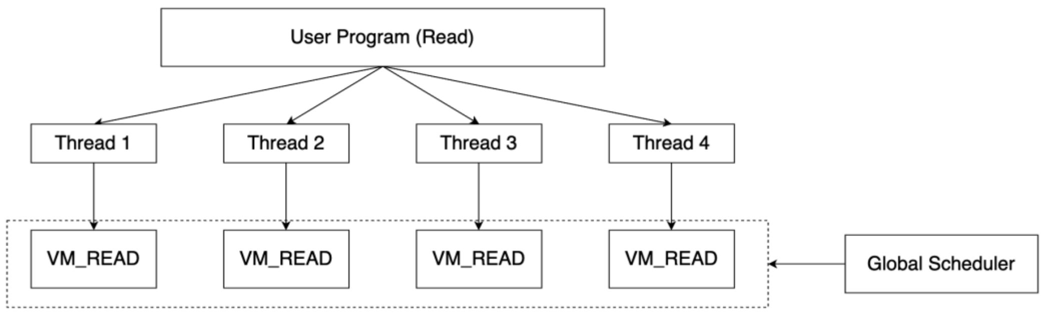
Q: VM is not large enough since we have 4 threads.
A: It depends on your implementation. The secondary memory can be 512KB in bonus.

Q: Because of parallelism, can't we guarantee that each of the four threads page faulted several times? Unless our test case "user_program" can be given a dead order of execution, we can't tell from the page fault whether it was written correctly or not, we can only see the respective snapshot.
A: We will check the total page fault number and the snapshot. (write_binaryFile is a host function, so there would be only one snapshot file)

Q: How many page tables do we need to maintain? What about the security issues and integrity issues?
A: We are aware that the previous illustration figure of the bonus is needed to be corrected. Only one inverted page table is needed. The intention of this assignment is not to discuss too many security/integrity issues. You can design your program to make each thread use one block (128KB) of the disk without confliction.

Q: Can we have a template or test case for bonus part?
A: **In bonus part, you only need to pass the first test case of the basic part (Our grading test cases will be similar).** And hence we don’t provide the template of the bonus.


Secondary Memory
Swap
Physical Memory
Page Table
thread id = 1
thread id = 2
thread id = 3
thread id = 4
Thread 1
Thread 2
Thread 3
Thread 4
Global Scheduler


User Program (Read)
Thread 1
Thread 2
Thread 3
Thread 4
VM_READ
VM_READ
VM_READ
VM_READ
Global Scheduler