Assignment 2

Your task is to implement a program that simulates Virtual Memory. It is going to receive a sequence of instructions and should simulate them in the given order. You will receive this sequence of instructions in a single **`input.txt`** file with each instruction on a separate line. After executing an instruction, you should add its result to a **`log.txt`** file.

Once complete, generated log file should be submitted along with the solution file in a single ZIP archive. Solution can be written in any programming language of your choice.

Below is the list of instructions that your program should accept and execute. It is guaranteed that all instructions and addresses are valid.

Command	Description	Log output
`Init V M S P`	Will always be the first instruction of the sequence. `V` – size of the virtual memory in Kilobytes. `M` – size of the physical memory in Kilobytes. `S` – size of the swap area in Kilobytes. `P` – size of a single page in Kilobytes.	`Initialized`
`Read A`	Read from the virtual address `A`. If the page to which `A` belongs is not in operating memory, a page fault occurs. Page fault must be handled **before reading** the value. `A` – virtual address in Bytes, non-negative integer less than V. `X` – an integer value, previously written to the address `A` or 0 if `A` has not been modified before.	`Value at address A is X`
`Write A X`	Writes value `X` to the virtual address `A`. If the page to which `A` belongs is not in operating memory, a page fault occurs. Page fault must be handled **before writing** the value. Page always becomes **dirty** once written on it. `A` – virtual address in Bytes, non-negative integer less than `V`. `X` – an integer value that needs to be written to the address `A`.	`Written X to address A`
`Exit`	Last instruction, should finish the simulation, dump the state of memory and swap.	`*see the sample`
All page faults should be logged as well. Any transaction between swap and main memory should be reflected in the output on a separate line. `A` – virtual address in Bytes, non-negative integer less than `V`.		`Page fault at A`
Before moving a page from swap to the memory, once main memory is full, a page should be chosen to be evicted. Implement the **second chance** algorithm for this purpose. `P` – index of the page evicted. `F` – index of the page frame released.		`Evicting page P from frame F`
In case if evicted page is **dirty**, swap area has to be updated. `S` – index of the block in the swap area, where the page P is stored.		`Saving page P to swap block S`
Finally, a page fault might result in moving a page from the swap area to the main memory. Initially, since swap is empty, page loads directly into the memory. `P` – index of the page loaded, for which page fault has occurred. `S` – index of the block in the swap area, where the page P was stored. `F` – index of the destination page frame.		`Loading page P [from swap block S] to frame F`

Below is the sample input/output.

Virtual memory – 20 KB, operating memory – 5 KB, swap – 10 KB, page size – 1 KB.

Sample input (input.txt)	**Sample output (log.txt)**	**Clarification**
`Init 20 5 10 1`	`Initialized`	

`Read 150`	`Page fault at 150`
	`Loading page 0 to frame 0`
	`Value at address 150 is 0`
		Initially all values in the memory are 0. Loading to the frame 0

`Read 12500`	`Page fault at 12500`
	`Loading page 12 to frame 1`
	`Value at address 12500 is 0`
		Loading page 12 to the frame 1.

`Read 12501`	`Value at address 12501 is 0`	No page loading here, as page 12 is already loaded.
`Write 12999 10`	`Written 10 to address 12999`	Marking page 12 as dirty, updating the value.
`Read 12999`	`Value at address 12999 is 10`	Reading from updated value.

`Write 13000 55`	`Page fault at 13000`
	`Loading page 13 to frame 2`
	`Written 55 to address 13000`
		First byte of the page 13 set to 55. Since it is a new page, loading to the frame 2.

`Write 1000 77`	`Page fault at 1000`
	`Loading page 1 to frame 3`
	`Written 77 to address 1000`
		Page 1 loaded to the frame 3, updated and marked as dirty.

`Read 2000`	`Page fault at 2000`
	`Loading page 2 to frame 4`
	`Value at address 2000 is 0`
		Reading from the next page and loading it to the frame 4. Now, all 5 frames are full.

`Read 3000`	`Page fault at 3000`
	`Evicting page 0 from frame 0`
	`Loading page 3 to frame 0`
	`Value at address 3000 is 0`
		Trying to load a new page. Since all frames are full now, we should evict one using the second chance algorithm. As all pages have R bit set to 1, after one round of clearing R bits, the very first page 0 is chosen to be evicted.

`Read 4000`	`Page fault at 4000`
	`Evicting page 12 from frame 1`
	`Saving page 12 to swap block 0`
	`Loading page 4 to frame 1`
	`Value at address 4000 is 0`
		In a similar manner evicting the next page without R bit – page 12. As it was modified before, we need to store it in the first empty swap area.

`Read 12999`	`Page fault at 12999`
	`Evicting page 13 from frame 2`
	`Saving page 13 to swap block 1`
	`Loading page 12 from swap block 0 to frame 2`
	`Value at address 12999 is 10`
		Page 13 was modified, saving it to next empty swap block 1. Loading page 12 from swap to the frame 2, and freeing swap block 1.

`Read 5000`	`Page fault at 5000`
	`Evicting page 1 from frame 3`
	`Saving page 1 to swap block 0`
	`Loading page 5 to frame 3`
	`Value at address 5000 is 0`
		Again, page replacement, page 1 is dirty, storing it in the first empty swap block 0.

`Exit`	`Memory – 3, 4, 12, 5, 2.` `Swap – 1, 13, 0, 0, 0, 0, 0, 0, 0, 0.`	State of both memory and swap is printed on separate lines. Since page size is 1KB, for this sample, memory contains 5 frames, and swap has 10 blocks. Dumping their current state.