MP4: Page Manager II

Vishnuvasan Raghuraman
UIN: 234009303
CSCE611: Operating System

Assigned Tasks

Main: Completed.

System Design

The objective of this machine problem is to extend the management of page tables to accommodate virtual memory and develop a virtual memory allocator. Within this machine problem, we allocate page table pages within mapped memory, specifically memory above 4 MB. These frames are managed by the process frame pool.

Dealing with large address spaces : Due to the limited size of the directly mapped memory, it's impractical for larger address spaces. Therefore, we utilize the process memory pool for allocating memory for page table pages. Additionally, when paging is activated, the CPU generates logical addresses. Thus, to establish mappings between logical addresses and frames and to modify entries within the page directory and page table pages, we employ a method known as 'Recursive Page Table Lookup'.

In the *Recursive Page Table Lookup* method, the final entry in the page directory directs to the beginning of the page directory itself. Both the page directory and page table pages store physical addresses.

To access a page directory entry, we utilize the following logical address format:

$$|1023:10|\ 1023:10|\ \text{offset}:12|$$

The Memory Management Unit (MMU) employs the first 10 bits (value 1023) to index the page directory and look up the Page Directory Entry (PDE). PDE number 1023, the final entry, points to the page directory itself. Subsequently, the MMU treats the page directory as a regular page table page. Next, the MMU utilizes the second 10 bits to index into the (presumed) page table page for the Page Table Entry (PTE). Since these bits also hold the value 1023, the resulting PTE once again points to the page directory itself. Once more, the MMU treats the page directory as a frame, using the offset to index into the physical frame.

For accessing a page table page entry, the logical address format is as follows:

$$|\ 1023:10\ |\ X:10\ |\ Y:10\ |\ 0:2\ |$$

The MMU utilizes the first 10 bits (value 1023) to index the page directory and look up the Page Directory Entry (PDE). PDE number 1023 directs to the page directory itself. Despite being the directory, the MMU treats it similarly to any other page table page.

Subsequently, the MMU utilizes the second 10 bits (value X) to index into the (assumed) page table page to retrieve the Page Table Entry (PTE), which in reality corresponds to the Xth PDE. The offset

now serves to index into the (supposed) physical frame, which actually refers to the page table page associated with the Xth directory entry. Hence, the remaining 12 bits are available to index into the Yth entry within the page table page. This approach enables the manipulation of a page directory entry or page table page entry stored in virtual memory.

To enable Virtual Memory support for Page Tables, the page table must possess awareness of all created virtual memory pools to distinguish between legitimate memory accesses and invalid accesses. This entails maintaining a list of all virtual memory pools within the page table. Additionally, when a page fault occurs, the system must verify if the accessed address is legitimate. Furthermore, previously allocated pages need to be released. To facilitate these functionalities, we implement the following functions: *register_pool', 'is_legitimate', and 'free_page'*. We utilize a linked list structure to store information about virtual memory pool regions available for allocation. Each node of this linked list contains details such as the base address, size of the memory pool region, the number of regions, and the available virtual memory pool size.

We have developed a straightforward Virtual Memory Allocator capable of allocating and de-allocating virtual memory in page-size multiples. With the *allocate* function, we can reserve a specific region of virtual memory within the designated virtual memory pool. Similarly, the *de-allocate* function allows us to release a previously allocated region of virtual memory.

Our implementation utilizes an array to track the allocation and de-allocation of virtual memory regions. Each element of this array is of type *alloc_region_info*, which is a structure holding essential information such as the base address and length of the allocated virtual memory region.

Code Description

A Silicon(M2) Mac was used for this assignment and the native environment setup for such a machine from Piazza was followed. The below files were modified for this particular MP.

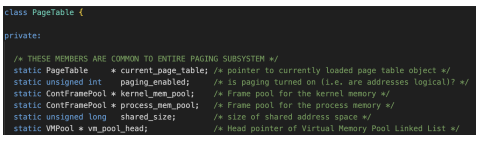
- vm_pool.H - In this file, the class VMPool declares data structures necessary for managing the virtual memory pool.
- vm_pool.C - This file contains functions such as allocate(), release(), is_legitimate(), etc.,

The below files from the previous MP(MP3) were used for this MP4 as a continuation. They were modified based on the requirements for the current MP.

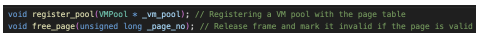
- cont_frame_pool.C
- page_table.H
- page_table.C

The functions implemented are given as below:

page_table.H: : The data structures for the frame pool were declared. Additionally, 2 more functions were defined as shown in the image below.



(a) VM Pool Head pointer



(b) New Function header

Figure 1: page_table.H modifications; VM Pool Linked List Header (a) and New Function header (b).

page_table.C : register_pool: This function is employed to register a virtual memory pool with the page table. A Linked List has been utilized to manage the list of virtual memory pools. Newly created virtual memory pools are added to the end of the list.

```
void PageTable::register_pool(VMPool * _vm_pool)
{
    // registering the initial VM pool
    if(PageTable::vm_pool_head == NULL){
        PageTable::vm_pool_head = _vm_pool;
    }
    // registering subsequent VM pools
    else{
        VMPool * temp = PageTable::vm_pool_head;
        for(;temp->vm_pool_next!=NULL;temp=temp->vm_pool_next);
        // adding pool to tje end of the linked list
        temp->vm_pool_next = _vm_pool;
    }
    Console::puts("Successfully Registered VM pool\n");
}
```

page_table.C : free_page: This function is used to release a frame and mark the corresponding page table entry as 'invalid'. It involves extracting the page directory index and page table index from the page number. Subsequently, we construct the logical address and access the page table entry to retrieve the frame number. The frame is then released from the process memory pool, and the page table entry is marked as 'invalid'. To ensure all outdated TLB entries are removed, the TLB is flushed by reloading the page table.

```
void PageTable::free_page(unsigned long _page_no)
{
    // extracting page directory index - First 10 bits
    unsigned long page_dir_ind = ( _page_no & 0xFFC00000) >> 22;
    // extracting page table index using mask - next 10 bits
    unsigned long page_table_ind = (_page_no & 0x003FF000 ) >> 12;
    // PTE Address = 1023 | PDE | Offset
    unsigned long * page_table = (unsigned long *) ( (0x000003FF << 22) | (page_dir_ind << 12) );
    // obtaining frame number for releasing
    unsigned long frame_no = ( (page_table[page_table_ind] & 0xFFFFF000) / PAGE_SIZE );
    // releasing frame from process pool
    process_mem_pool->release_frames(frame_no);
    // marking PTE as invalid
    page_table[page_table_ind] = page_table[page_table_ind] | 0b10;
    // flushing TLB by reloading page table
    load();
    Console::puts("Successfully Freed page!\n");
}
```

page_table.C : handle_fault: This method is responsible for handling page-fault exceptions of the CPU. It begins by examining the error code of Exception 14 to determine if a 'page not present' fault occurred. Subsequently, it retrieves the page fault address from register CR2 and the page directory address from register CR3. Calculating the page directory index and page table index from the page fault address, it then iterates through the list of virtual memory pool regions to determine if the page fault address exists within any of these regions.

If the page fault is detected in the page directory, a free frame from the process pool is allocated for the page directory entry, marked as 'valid' using bitwise operations. To prevent further page faults, an available free frame is allocated from the process pool for the corresponding page table entry, and the page is marked as 'valid'. Additionally, the corresponding page table is initialized by allocating a free frame from the process pool and marking all entries as 'invalid'. If the fault occurred in the page table, a free frame from the process pool is allocated for the page table entry and marked as 'valid'. Finally, the page fault handler concludes by returning.

```
void PageTable::handle_fault(REGS * _r)
{
    unsigned int ind = 0;
    unsigned long err_code = _r->err_code;
    // get page fault address using CR2 register


    // page fault occurs if page is not present
    if ((err_code & 1) == 0)
    {
        unsigned long fault_addr = read_cr2();
        // get page directory address using CR3 register
        unsigned long * page_dir = (unsigned long *)read_cr3();
        // get page directory index: first 10 bits
        unsigned long page_dir_ind = (fault_addr >> 22);
        // get page table index using mask: next 10 bits
        // 0x3FF = 001111111111 - retain only last 10 bits
        unsigned long page_table_ind = ((fault_addr & (0x3FF << 12)) >> 12);
        unsigned long * new_page_table = NULL;
        unsigned long *new_pde = NULL;

        // checking if logical address is valid and proper
        unsigned int present_flag = 0;

        // Iterating through the VM Pool regions
        VMPool * temp = PageTable::vm_pool_head;

        for(;temp!=NULL;temp=temp->vm_pool_next){
            if(temp->is_legitimate(fault_addr) == true){
                present_flag = 1;
                break;
            }
        }

        if((temp != NULL) && (present_flag == 0)){
            Console::puts("Not a legit address!! \n");
            assert(false);
```

```
        // checking where page fault has occured
        if ((page_dir[page_dir_ind] & 1) == 0)
        {
            int ind = 0;
            new_page_table = (unsigned long *)(process_mem_pool->get_frames(1) * PAGE_SIZE);
            // PDE = Page Directory Entry
            // PTE = Page Table Entry
            // PDE address = 1023 | 1023 | Offset
            unsigned long * new_pde = (unsigned long *)( 0xFFFFF << 12 );
            new_pde[page_dir_ind] = ( (unsigned long)(new_page_table)| 0b11 );

            // setting flags for each page - PTEs marked as invalid
            for(ind=0;ind<1024;ind++){
                // Set user level flag bit
                new_page_table[ind] = 0b100;
            }
            // Handling invalid PTE scenario to avoid raising another page fault
            new_pde = (unsigned long *) (process_mem_pool->get_frames(1) * PAGE_SIZE);
            // PTE address = 1023 | PDE | Offset
            unsigned long * page_entry = (unsigned long *)( (0x3FF << 22) | (page_dir_ind << 12) );

            // marking PTE valid
            page_entry[page_table_ind] = ( (unsigned long)(new_pde) | 0b11 );
        }
        else{
            // Page fault occured in page table page - PDE is present, but PTE is invalid
            new_pde = (unsigned long *) (process_mem_pool->get_frames(1) * PAGE_SIZE);
            // PTE address = 1023 | PDE | Offset
            unsigned long * page_entry = (unsigned long *)( (0x3FF << 22)| (page_dir_ind << 12) );
            page_entry[page_table_ind] = ( (unsigned long)(new_pde) | 0b11 );
        }
    }

    Console::puts("handled page fault\n");
}
```

vm_pool.H: In the class VMPool, the following data structures for managing the virtual memory pool were declared:

- unsigned long base_address: Represents the logical start address of the virtual memory pool.
- unsigned long size: Denotes the size of the virtual memory pool in bytes.
- unsigned long num_regions: Indicates the number of virtual memory pools.
- unsigned long available_mem: Represents the size of virtual memory available within the pool.
- struct alloc_region_info *vm_regions: A pointer to the allocation list of virtual memory regions.
- ContFramePool *frame_pool: A pointer to the frame pool that provides frames for the virtual memory pool.
- VMPool *vm_pool_next: A pointer to the next virtual memory pool in the linked list.

In the `alloc_region_info` structure, we store information about each region in virtual memory:

- unsigned long base_address: Represents the base address of the virtual memory region.
- unsigned long length: Denotes the length of the virtual memory region.

```
// structure used for holding information of each region in virtual memory
struct alloc_region_info{
   unsigned long base_address;
   unsigned long length;
};

/*--------------------------------------------------------------------------*/
/* V M  P o o l  */
/*--------------------------------------------------------------------------*/

class VMPool { /* Virtual Memory Pool */
private:
   /* -- DEFINE YOUR VIRTUAL MEMORY POOL DATA STRUCTURE(s) HERE. */
   unsigned long base_address;
   unsigned long size;
   unsigned long num_regions; // no of VM regions
   unsigned long available_memory; // size of mem region available
   struct alloc_region_info * vm_regions; // ptr to VM region allocation
   ContFramePool * frame_pool;
   PageTable * page_table;

public:
   VMPool * vm_pool_next;  // ptr to vm pool linked list
```

vm_pool.C : VMPool: This constructor initializes all the necessary data structures for managing the virtual memory pool within the 'VMPool' class. It registers the virtual memory pool with the page table. The base address and length of the virtual memory region are stored in the 'alloc_region_info' structure. The variables 'num_regions' and 'available_mem' are used to keep track of the number of virtual memory regions allocated and the available virtual memory in the pool, respectively.

```
VMPool::VMPool(unsigned long  _base_address,
               unsigned long  _size,
               ContFramePool *_frame_pool,
               PageTable     *_page_table) {

    base_address = _base_address;
    size = _size;
    frame_pool = _frame_pool;
    page_table = _page_table;
    vm_pool_next = NULL;
    num_regions = 0;

    page_table->register_pool(this);   // registering the vm pool

    // first entry storing base address and page size
    alloc_region_info * region = (alloc_region_info*)base_address;
    region[0].base_address = base_address;
    region[0].length = PageTable::PAGE_SIZE;
    vm_regions = region;

    num_regions += 1;

    available_memory -= PageTable::PAGE_SIZE; // calculating available vm

    Console::puts("Constructed VMPool object.\n");
}
```

vm_pool.C : allocate: This method allocates a region of memory from the virtual memory pool. Initially, it compares the requested allocation size with the available virtual memory pool size. Then, it calculates the number of pages required for the allocation. Subsequently, the method calculates the base address and length of the new virtual memory region and stores its details in the data structure 'vm_regions', adding it to the allocated regions array. After that, it recalculates the available virtual memory pool size and increments the variable 'num_regions'. Upon success, the method returns the virtual address of the start of the allocated region of memory. In case of failure, the method returns zero.

```
unsigned long VMPool::allocate(unsigned long _size) {
    unsigned long pages_ct = 0;
    if( _size > available_memory ){
        Console::puts("VMPool::allocate - Not enough vm space available.\n");
        assert(false);
    }
    pages_ct = (_size / PageTable::PAGE_SIZE) + ((_size % PageTable::PAGE_SIZE) > 0 ? 1 : 0);   // calculating number of pages to be allocated
    // storing details of new vm region
    vm_regions[num_regions].base_address = vm_regions[num_regions-1].base_address +  vm_regions[num_regions-1].length;
    vm_regions[num_regions].length = pages_ct * PageTable::PAGE_SIZE;
    // calculating available memory
    available_memory -= pages_ct * PageTable::PAGE_SIZE;
    num_regions += 1;
    Console::puts("Successfully allocated region of memory.\n");

    // returning the allocated base_address
    return vm_regions[num_regions-1].base_address;
}
```

vm_pool.C : release: This method releases a region of previously allocated virtual memory. It takes the start address argument and iterates through the virtual memory regions that have been allocated to identify the region to which the start address belongs. Next, it calculates the number of pages to be freed and iteratively frees each page. Afterward, it updates the virtual memory allocation information by removing the array element that holds information about the region to be released. Finally, it recalculates the available virtual memory pool size and decrements the variable 'num_regions'.

```
void VMPool::release(unsigned long _start_address) {
    int ind = 0;
    int region_no = -1;
    unsigned long page_ct = 0;

    // iterating and finding the region the start address belongs to
    for(ind=1;ind<num_regions;ind++){
        if(vm_regions[ind].base_address==_start_address){
            region_no = ind;
        }
    }
    // calculating the number of pages to free
    page_ct = vm_regions[region_no].length / PageTable::PAGE_SIZE;
    while(page_ct > 0){
        // freeing the page
        page_table->free_page(_start_address);
        // incrementing the start address by page size
        _start_address = _start_address + PageTable::PAGE_SIZE;
        page_ct = page_ct - 1;
    }
    // deleting the virtual memory region information
    for(ind=region_no;ind<num_regions;ind++){
        vm_regions[ind] = vm_regions[ind+1];
    }
    available_memory += vm_regions[region_no].length;  // calculating the available memory
    num_regions -= 1;   // decrementing the number of regions

    Console::puts("Released region of memory.\n");
}
```

vm_pool.C : is_legitimate: This method is employed to verify if the address is within a region that is currently allocated. It checks if the address argument lies between the bounds of 'base_address' and ('base_address' + 'size'). Upon success, it returns a boolean true. In case of failure, indicating that the address is not valid, it returns a boolean false.

```
bool VMPool::is_legitimate(unsigned long _address) {
    Console::puts("Checked whether address is part of an allocated region.\n");
    if((_address < base_address) || (_address > (base_address + size))){
        return false;
    }
    return true;
}
```

cont_frame_pool.C : release_frame_range: This method is utilized to release all the frames within a specific frame pool. A correction has been made in the logic of this method. In the previous implementation, after confirming that the first frame state is 'Head of Segment (HoS)', we didn't iterate through the frames until we encountered a frame that is either 'Free' or 'HoS'. This resulted in errors such as 'Frame state is not HoS'.

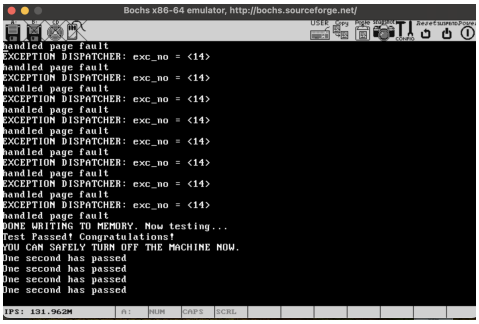
In the modified implementation, after confirming that the first frame state is 'HoS', we iterate through the frames until we encounter a frame that is 'Free' or 'HoS'. During this iteration, we mark all the frames we traverse as 'Free' until we reach a frame that meets the specified conditions.

```
void ContFramePool::release_frame_range(unsigned long _first_frame_no){
    unsigned long index = _first_frame_no + 1;
    // Get the state of frame
    if( get_state(_first_frame_no - base_frame_no) == FrameState::HoS ){
        set_state(_first_frame_no, FrameState::Free);
        nFreeFrames = nFreeFrames + 1;  // incrementing number of free frames
        while( get_state( index - base_frame_no) != FrameState::Free ){
            set_state(index, FrameState::Free); // setting state to free
            nFreeFrames = nFreeFrames + 1;  // incrementing number of free frames
            index = index + 1;
        }
    }
    else{
        Console::puts("ContframePool::release_frame_range - Cannot release frame. Frame state is not HoS.\n");
        assert(false);
    }
}
```

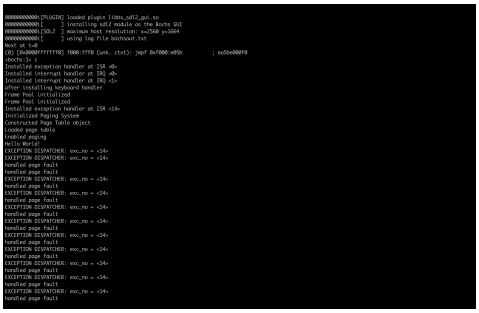
Testing

A few testcases were carried out by making modifications to the *kernel.C* file in the part of the code given below. This code block was changed based on the different test cases.

- Testing page table: Fault Address = 4MB



(a) Bochs Emulator



(b) Terminal

Figure 2: Fault address=4MB; Bochs Emulator (a) and Terminal (b).

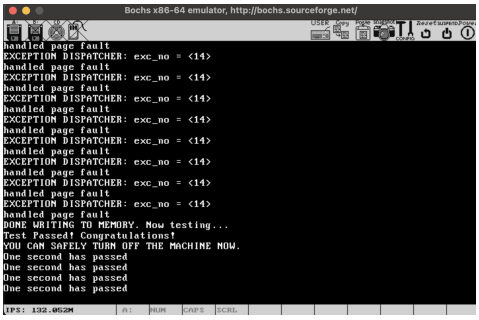
```c
#define FAULT_ADDR (4 MB)
/* used in the code later as address referenced to cause page faults. */
#define NACCESS ((1 MB) / 4)
/* NACCESS integer access (i.e. 4 bytes in each access) are made starting at address FAULT_ADDR */
```

Figure 3: Fault address=4MB - Code

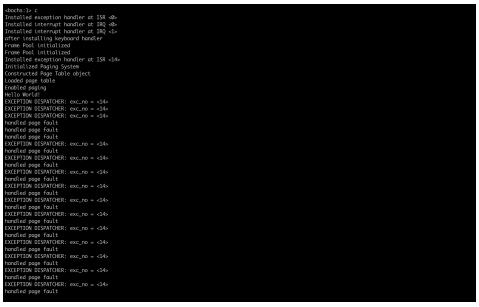
Expected Result: Page table memory references are generated, and all page faults are effectively managed, ensuring that data reads and writes are equivalent.

Actual Result: Page table memory references are generated, and all page faults are appropriately handled, ensuring that data reads and writes are consistent.

- Testing page table: Fault Address = 16MB



(a) Bochs Emulator



(b) Terminal

Figure 4: Fault address=16MB; Bochs Emulator (a) and Terminal (b).

```
#define FAULT_ADDR (16 MB)
/* used in the code later as address referenced to cause page faults. */
#define NACCESS ((1 MB) / 4)
/* NACCESS integer access (i.e. 4 bytes in each access) are made starting at address FAULT_ADDR */
```

Figure 5: Fault address=16MB - Code

Expected Result: Page table memory references are generated, and all page faults are effectively managed, ensuring that data reads and writes are equivalent.

Actual Result: Page table memory references are generated, and all page faults are appropriately handled, ensuring that data reads and writes are consistent.

- Allocating and releasing multiple heap pools sequentially starting from the same memory location. - Refer Figure. 6 and 7

(a) Bochs Emulator

(b) Terminal

Figure 6: Heap Pool Testing; Bochs Emulator (a) and Terminal (b).

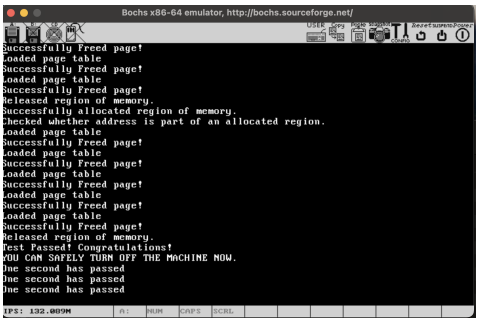
```
229  #else
230
231      /* WE TEST JUST THE VM POOLS */
232
233      /* -- CREATE THE VM POOLS. */
234
235      /* ---- We define the code pool to be a 256MB segment starting at virtual address 512MB -- */
236      VMPool code_pool(512 MB, 256 MB, &process_mem_pool, &pt1);
237
238      /* ---- We define a 256MB heap that starts at 1GB in virtual memory. -- */
239      VMPool heap_pool(1 GB, 256 MB, &process_mem_pool, &pt1);
240      VMPool heap_pool2(1 GB, 256 MB, &process_mem_pool, &pt1);
241
242      /* -- NOW THE POOLS HAVE BEEN CREATED. */
243
244      Console::puts("VM Pools successfully created!\n");
245
246      /* -- GENERATE MEMORY REFERENCES TO THE VM POOLS */
247
248      Console::puts("I am starting with an extensive test\n");
249      Console::puts("of the VM Pool memory allocator.\n");
250      Console::puts("Please be patient...\n");
251      Console::puts("Testing the memory allocation on code_pool...\n");
252      GenerateVMPoolMemoryReferences(&code_pool, 50, 100);
253      Console::puts("Testing the memory allocation on heap_pool...\n");
254      GenerateVMPoolMemoryReferences(&heap_pool, 50, 100);
255      GenerateVMPoolMemoryReferences(&heap_pool2, 50, 100);
```

Figure 7: Heap Pool Testing - Code

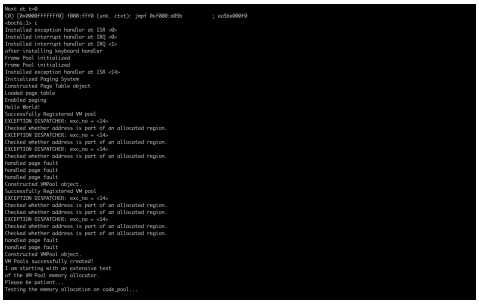
Expected Result: Memory was successfully allocated and released at the same memory location consecutively.

Actual Result: Memory was successfully allocated and released at the same memory location consecutively.

- Testing VM pools



(a) Bochs Emulator



(b) Terminal

Figure 8: VM Pool Testing; Bochs Emulator (a) and Terminal (b).

```
    /* BY DEFAULT WE TEST THE PAGE TABLE IN MAPPED MEMORY!
       (COMMENT OUT THE FOLLOWING LINE TO TEST THE VM Pools! */
// #define _TEST_PAGE_TABLE_

#ifdef _TEST_PAGE_TABLE_

    /* WE TEST JUST THE PAGE TABLE */
    GeneratePageTableMemoryReferences(FAULT_ADDR, NACCESS);

#else

    /* WE TEST JUST THE VM POOLS */

    /* -- CREATE THE VM POOLS. */
```

Figure 9: VM Pool Testing - Code

Expected Result: Virtual memory references are created, ensuring that the data written and read maintain equivalence.

Actual Result: Virtual memory references are generated, ensuring that the data written and read remain equivalent.