MP4: Page Manager II

Sushant Vijay Shelar
UIN: 733001479
CSCE611: Operating System

Assigned Tasks

Part I: Implementation of Large Address Space Support - Completed
Part II: Enhancement of the PageTable Class to Manage Virtual Memory Pools - Completed
Part III: Development of a Virtual Memory Allocator - Completed

System Design

The virtual memory pool manager for this machine problem is constructed using code from previous Machine Problems (MP3 and MP2). The development of the memory manager is a multi-stage process, with each stage comprising several smaller tasks. As per the instructions in the handout, these tasks are categorized into three main sections:

1. Extending the page table management to handle large quantities and sizes of address spaces. This involves transferring directory and page table pages to the process memory pool.
2. Preparing the page table to support virtual memory.
3. Implementing a straightforward virtual memory allocator and integrating it with the new and delete operators.

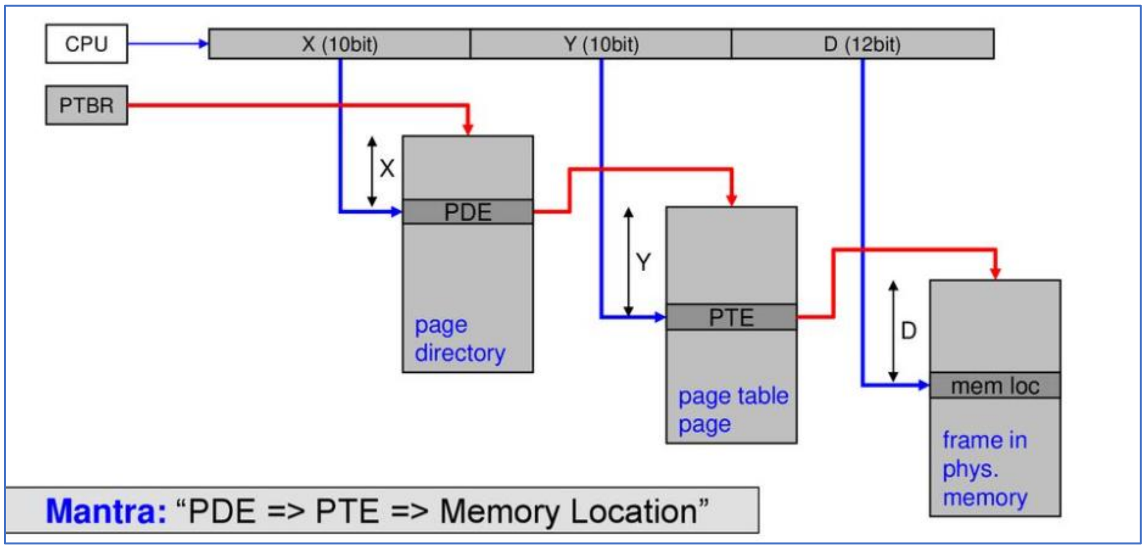


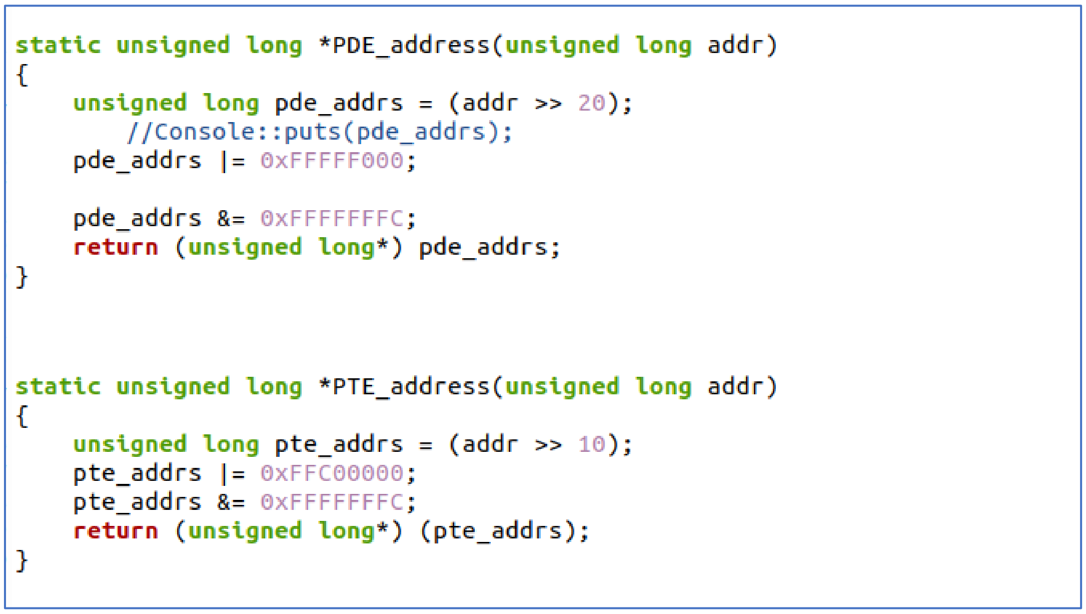
Figure 1: Address Translation in x86
(Source: https://slideplayer.com/slide/16460223/)

Code Description

- I made changes only to the "page_table.C", “vm_pool.c”, “page_table.H”, “vm_pool.H” files for this assignment.
- I've reused the "cont_frame_pool.C" and "cont_frame_pool.H" files from a previous submission.

Part I: Recursive Page Table Lookup

- The central aspect of the current Machine Problem is the recursive page table lookup. It leverages the structure of the page directory and page table.
- The offset is employed to access either of them since their contents ultimately represent frame numbers, effectively tricking the CPU into believing that the page directory or the page table corresponds to real pages.
- As explained in MP4 handout, to access the page directory, the following address format is used: 1023|1023|X|offset. Similarly, to access the page table page, the following address format is utilized: 1023|X|Y|offset.
- As mentioned in MP4 handout, “PDE address(unsigned long addr)” & “PTE address(unsigned long addr)” functions are implemented. It returns *pte and *pde.

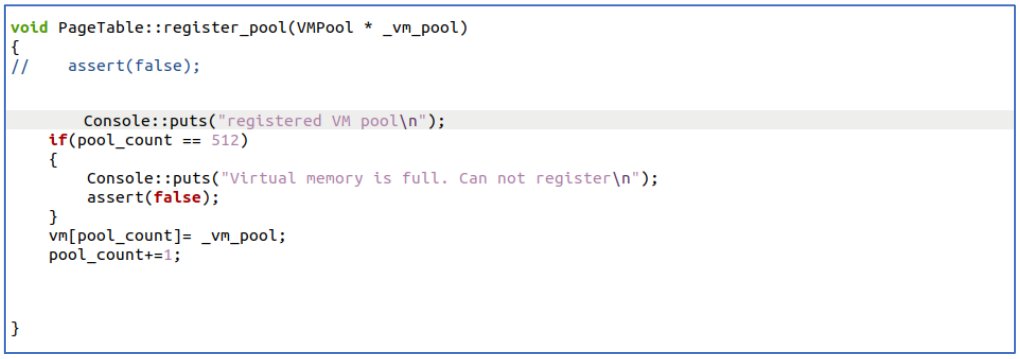


```
static unsigned long *PDE_address(unsigned long addr)
{
    unsigned long pde_addrs = (addr >> 20);
        //Console::puts(pde_addrs);
    pde_addrs |= 0xFFFFF000;

    pde_addrs &= 0xFFFFFFFC;
    return (unsigned long*) pde_addrs;
}



static unsigned long *PTE_address(unsigned long addr)
{
    unsigned long pte_addrs = (addr >> 10);
    pte_addrs |= 0xFFC00000;
    pte_addrs &= 0xFFFFFFFC;
    return (unsigned long*) (pte_addrs);
}
```

Figure 2: Functions for Page directory address and page table address

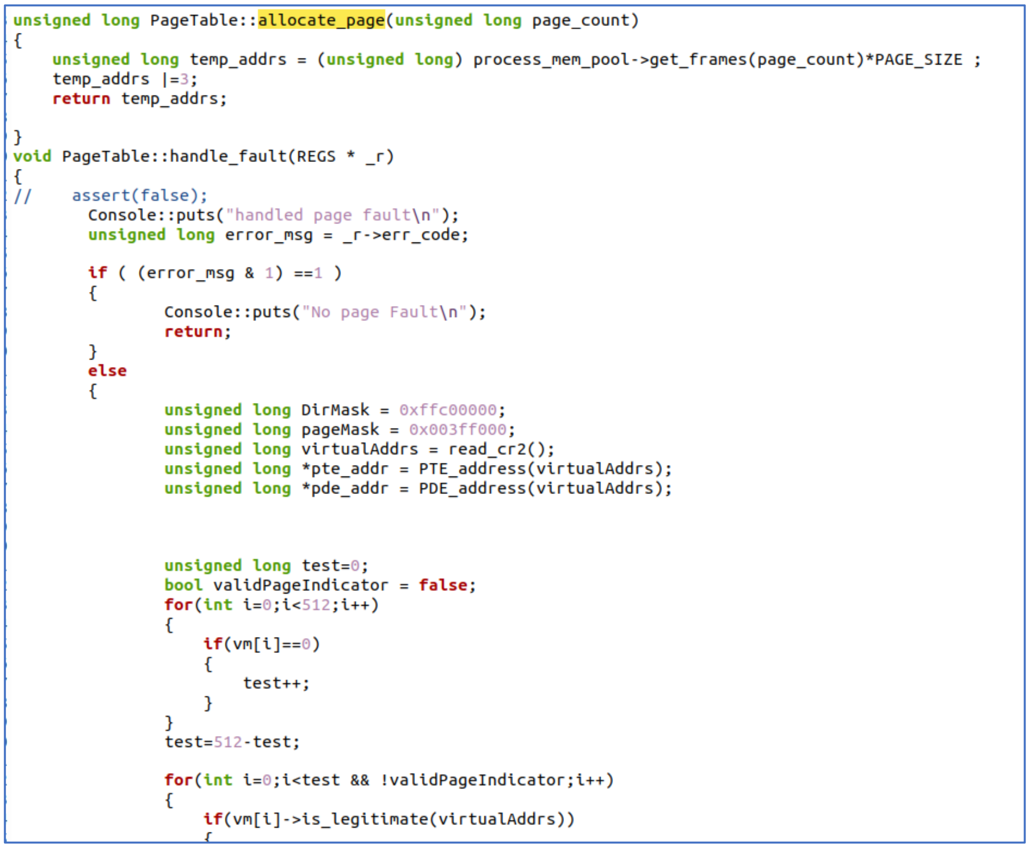
Part II: Preparing class PageTable to handle Virtual Memory Pools

- The page table class underwent modifications to incorporate additional APIs necessary for virtual memory management.
- The "register pool" API maintains a record of all memory pools established up to that point in an array. Another variable is utilized to keep a record of the registered virtual memory pools.
- Subsequently, the page fault handler was validated for the given address. Any address exhibiting an error should be located within the region where the virtual memory pool has been set up.
- If it doesn't function as expected, an error message is generated, and the operation halts. Additionally, the page table file now includes a "free page" API, which utilizes the provided frame number as an input to invoke the "release frame" API.



```
void PageTable::register_pool(VMPool * _vm_pool)
{
//    assert(false);


       Console::puts("registered VM pool\n");
    if(pool_count == 512)
    {
        Console::puts("Virtual memory is full. Can not register\n");
        assert(false);
    }
    vm[pool_count]= _vm_pool;
    pool_count+=1;



}
```

Figure 3: Register_Pool

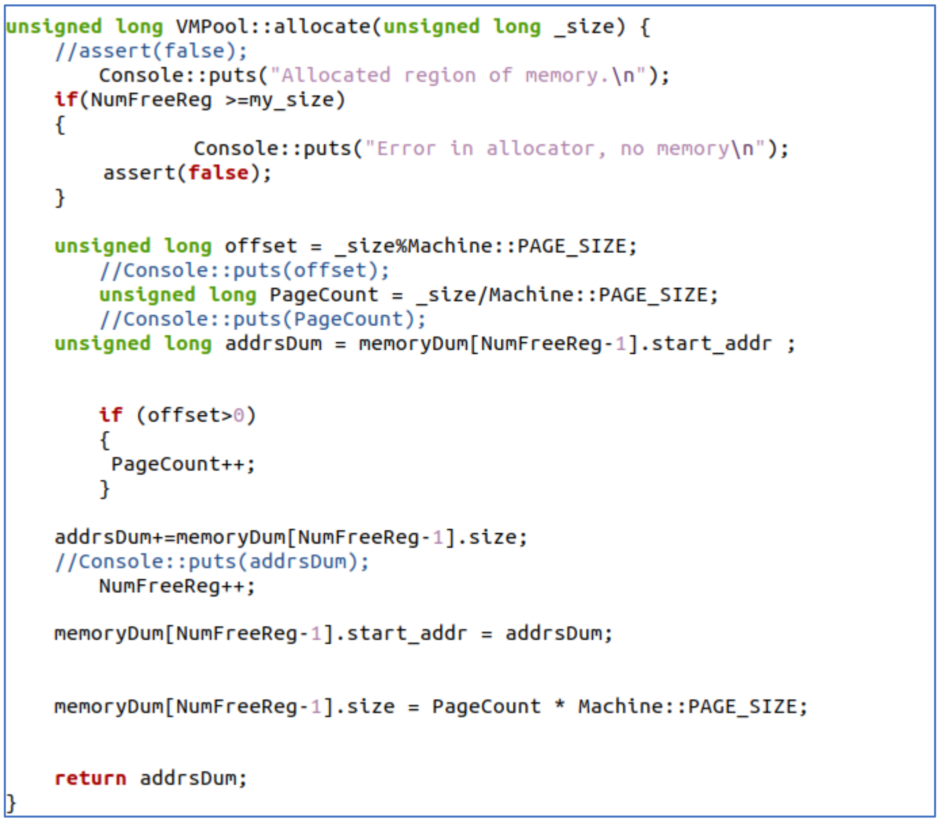


```
unsigned long PageTable::allocate_page(unsigned long page_count)
{
    unsigned long temp_addrs = (unsigned long) process_mem_pool->get_frames(page_count)*PAGE_SIZE ;
    temp_addrs |=3;
    return temp_addrs;

}
void PageTable::handle_fault(REGS * _r)
{
//    assert(false);
        Console::puts("handled page fault\n");
        unsigned long error_msg = _r->err_code;

        if ( (error_msg & 1) ==1 )
        {
                Console::puts("No page Fault\n");
                return;
        }
        else
        {
                unsigned long DirMask = 0xffc00000;
                unsigned long pageMask = 0x003ff000;
                unsigned long virtualAddrs = read_cr2();
                unsigned long *pte_addr = PTE_address(virtualAddrs);
                unsigned long *pde_addr = PDE_address(virtualAddrs);



                unsigned long test=0;
                bool validPageIndicator = false;
                for(int i=0;i<512;i++)
                {
                    if(vm[i]==0)
                    {
                        test++;
                    }
                }
                test=512-test;

                for(int i=0;i<test && !validPageIndicator;i++)
                {
                    if(vm[i]->is_legitimate(virtualAddrs))
                    {
```

Figure 4: Allocate_page and handle_fault

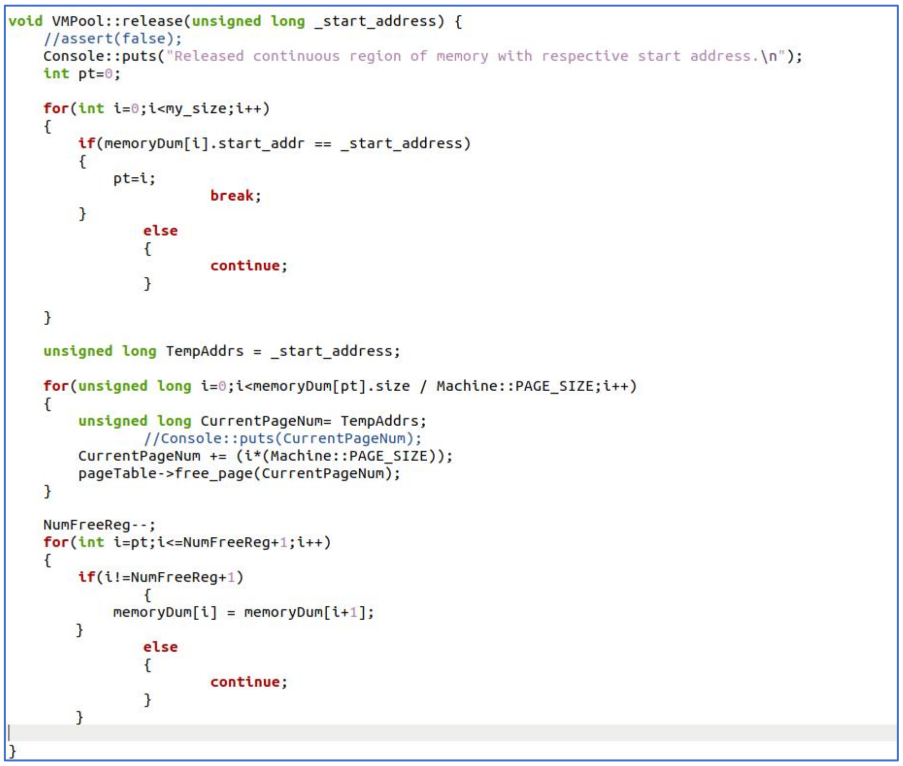
Part III: An Allocator for Virtual Memory:

- The page table class underwent modifications to include several additional APIs necessary for virtual memory management, such as the "register pool" function, which keeps track of all memory pools registered at any given time.
- Subsequently, the address was verified against the page fault handler. Any erroneous addresses should be located within the area designated for the virtual memory pool.
- If this verification process fails, an error message is issued, and the operation is halted. Furthermore, the page table file now incorporates a "free page" API, which calls the "release frame" API using the provided frame number as an input.



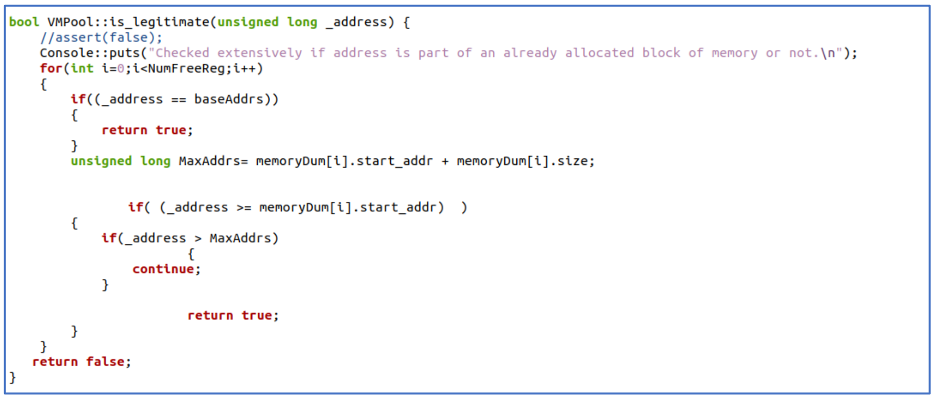
```
unsigned long VMPool::allocate(unsigned long _size) {
    //assert(false);
        Console::puts("Allocated region of memory.\n");
    if(NumFreeReg >=my_size)
    {
                Console::puts("Error in allocator, no memory\n");
        assert(false);
    }

    unsigned long offset = _size%Machine::PAGE_SIZE;
        //Console::puts(offset);
        unsigned long PageCount = _size/Machine::PAGE_SIZE;
        //Console::puts(PageCount);
    unsigned long addrsDum = memoryDum[NumFreeReg-1].start_addr ;


        if (offset>0)
        {
         PageCount++;
        }

    addrsDum+=memoryDum[NumFreeReg-1].size;
    //Console::puts(addrsDum);
        NumFreeReg++;

    memoryDum[NumFreeReg-1].start_addr = addrsDum;


    memoryDum[NumFreeReg-1].size = PageCount * Machine::PAGE_SIZE;


    return addrsDum;
}
```

Figure 5: Allocate function from vm_pool.c



```
void VMPool::release(unsigned long _start_address) {
    //assert(false);
    Console::puts("Released continuous region of memory with respective start address.\n");
    int pt=0;

    for(int i=0;i<my_size;i++)
    {
        if(memoryDum[i].start_addr == _start_address)
        {
            pt=i;
                        break;
        }
                else
                {
                        continue;
                }

    }

    unsigned long TempAddrs = _start_address;

    for(unsigned long i=0;i<memoryDum[pt].size / Machine::PAGE_SIZE;i++)
    {
        unsigned long CurrentPageNum= TempAddrs;
                //Console::puts(CurrentPageNum);
        CurrentPageNum += (i*(Machine::PAGE_SIZE));
        pageTable->free_page(CurrentPageNum);
    }

    NumFreeReg--;
    for(int i=pt;i<=NumFreeReg+1;i++)
    {
        if(i!=NumFreeReg+1)
                {
            memoryDum[i] = memoryDum[i+1];
        }
                else
                {
                        continue;
                }
        }

}
```

Figure 6: Release function



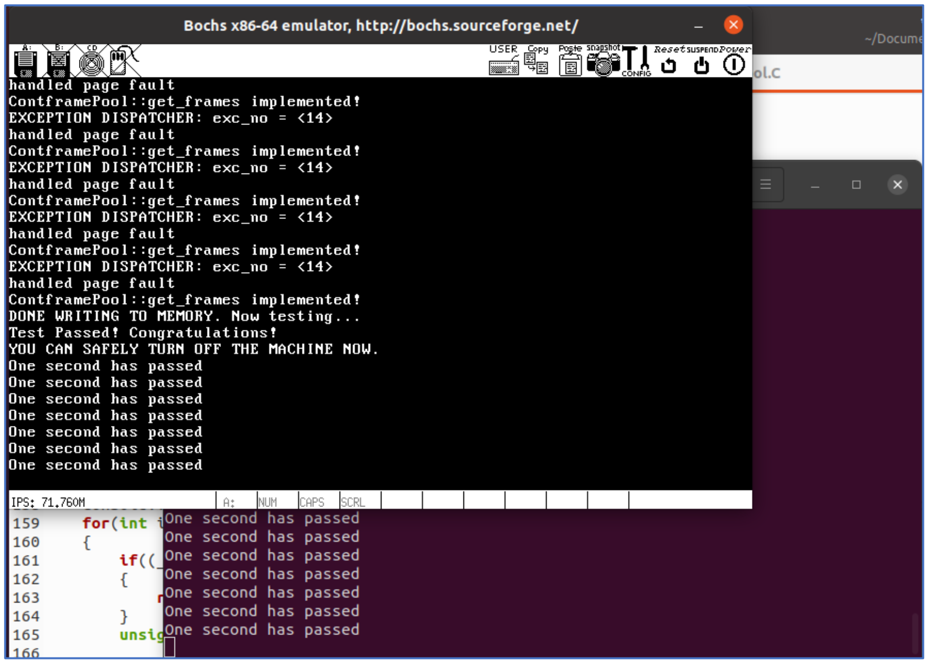
```
bool VMPool::is_legitimate(unsigned long _address) {
    //assert(false);
    Console::puts("Checked extensively if address is part of an already allocated block of memory or not.\n");
    for(int i=0;i<NumFreeReg;i++)
    {
        if((_address == baseAddrs))
        {
            return true;
        }
        unsigned long MaxAddrs= memoryDum[i].start_addr + memoryDum[i].size;


               if( (_address >= memoryDum[i].start_addr)  )
        {
            if(_address > MaxAddrs)
                           {
                continue;
            }

                           return true;
        }
    }
   return false;
}
```

Figure 7: is_legititmate function

Testing

I used the 'make' command to compile all the files in the "mp3" folder. Then, I ran a script called 'copy kernel' to copy the bin file to the disk. After that, I executed the code using the "bochs -f bochsrc.bxrc" command.

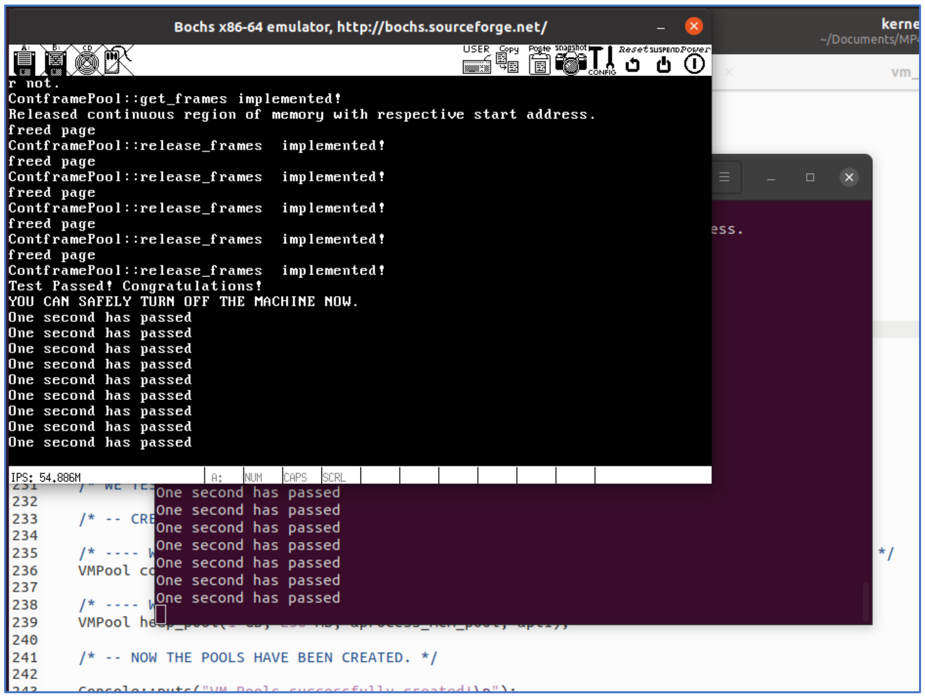
All the test cases worked as expected, and I've attached a screenshot for reference:



```
handled page fault
ContframePool::get_frames implemented!
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
ContframePool::get_frames implemented!
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
ContframePool::get_frames implemented!
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
ContframePool::get_frames implemented!
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
ContframePool::get_frames implemented!
DONE WRITING TO MEMORY. Now testing...
Test Passed! Congratulations!
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
```

```
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
```

Figure 8: Testing with TEST_PAGE_TABLE macro



```
r not.
ContframePool::get_frames implemented!
Released continuous region of memory with respective start address.
freed page
ContframePool::release_frames  implemented!
freed page
ContframePool::release_frames  implemented!
freed page
ContframePool::release_frames  implemented!
freed page
ContframePool::release_frames  implemented!
freed page
ContframePool::release_frames  implemented!
Test Passed! Congratulations!
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
```

```
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
One second has passed
```

```
    /* -- NOW THE POOLS HAVE BEEN CREATED. */
```

Figure 9: Testing without TEST_PAGE_TABLE macro