MP4: Virtual Memory Management and Memory Allocation

Alex Kilgore
UIN: 928007100
CSCE410: Operating System

Assigned Tasks

Main: Completed.

System Design

This machine problem is an extension of the previous machine problems where we complete our memory manager. The first part of the MP is extending our page table management to support large address spaces. Previously, our page table was entirely stored in directly mapped memory, so we will move the page table pages into virtual memory via the process memory pool. When paging is turned on, the CPU issues logical addresses, which we need to convert into the correct physical address to find the page being referenced. We solve this with recursive page table lookup, where the last entry in the page table directory points back to the beginning of the table. This means that we can always find the directory itself by looking at the address stored in its last index. This can also be used to find and manipulate a page table page in virtual memory.

The next part of the MP is to modify the existing PageTable class to support the virtual memory pool we implement in part 3. Each PageTable object must keep track of all of its associated virtual memory pools so that the page fault handler can make sure that page faults are coming from legitimate addresses and handle them accordingly. We must also allow virtual memory pools to request the release of any associated frames for a given page. This requires that we flush the TLB so there are not any defunct entries left over that could cause problems later.

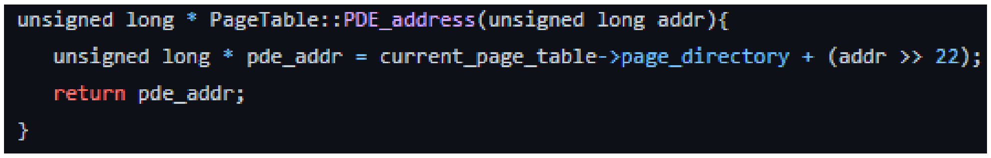
Part 3 of this MP is to actually implement the memory pools as the allocator for our virtual memory. This is done through the VMPool class. Address spaces can have multiple virtual memory pools, which are each split up into regions that can be allocated and released. We will be implementing a somewhat lazy allocator that doesn't automatically allocate frames for a new memory region, but instead keeps a list of which regions are allocated as well as their start address and size. Frames are only allocated once a reference to the memory region is made and the page fault handler is called into action.

Code Description

To compile and run the code, use the makefile, copy the kernel binary with ./copykernel.sh, and run in bochs.ff

PDE_address

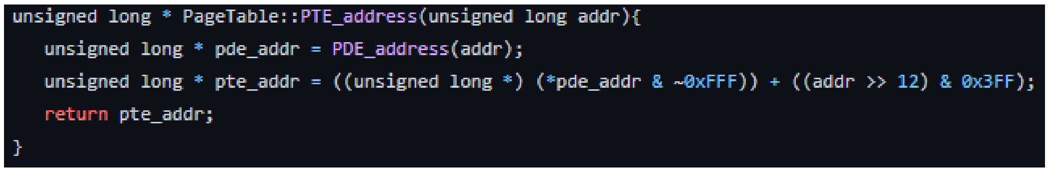
- Takes in an address as an unsigned long
- Returns the address of the PDE that matches the address in the current page table



```
unsigned long * PageTable::PDE_address(unsigned long addr){
   unsigned long * pde_addr = current_page_table->page_directory + (addr >> 22);
   return pde_addr;
}
```

PTE_address

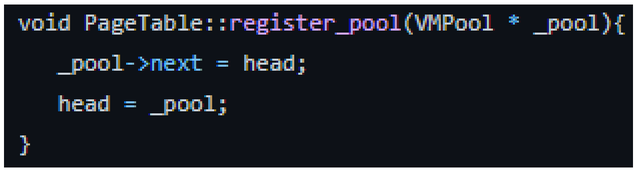
- Takes in an address as an unsigned long
- Finds the address of the PDE
- Returns the address of the PTE that matches the PDE and the address in the current page table



```
unsigned long * PageTable::PTE_address(unsigned long addr){
   unsigned long * pde_addr = PDE_address(addr);
   unsigned long * pte_addr = ((unsigned long *) (*pde_addr & ~0xFFF)) + ((addr >> 12) & 0x3FF);
   return pte_addr;
}
```

Register_pool

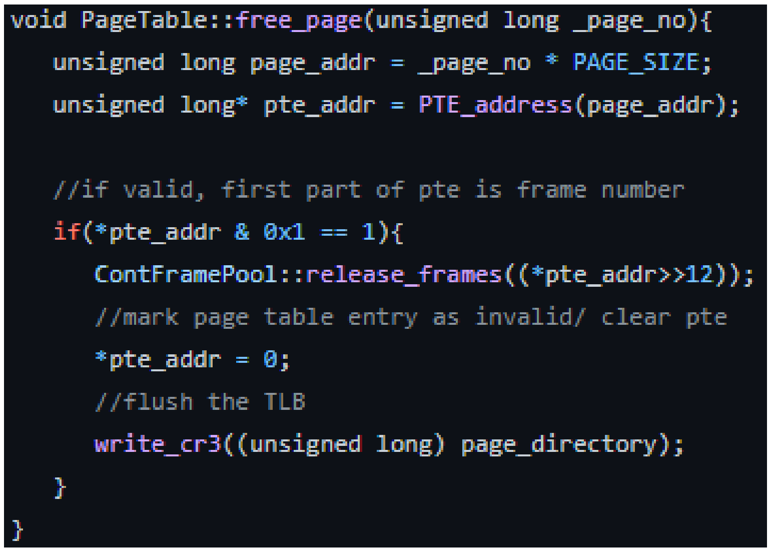
- Takes in a pointer to a VMPool
- Puts that VMPool object at the head of the linked list



```
void PageTable::register_pool(VMPool * _pool){
   _pool->next = head;
   head = _pool;
}
```

Free_page

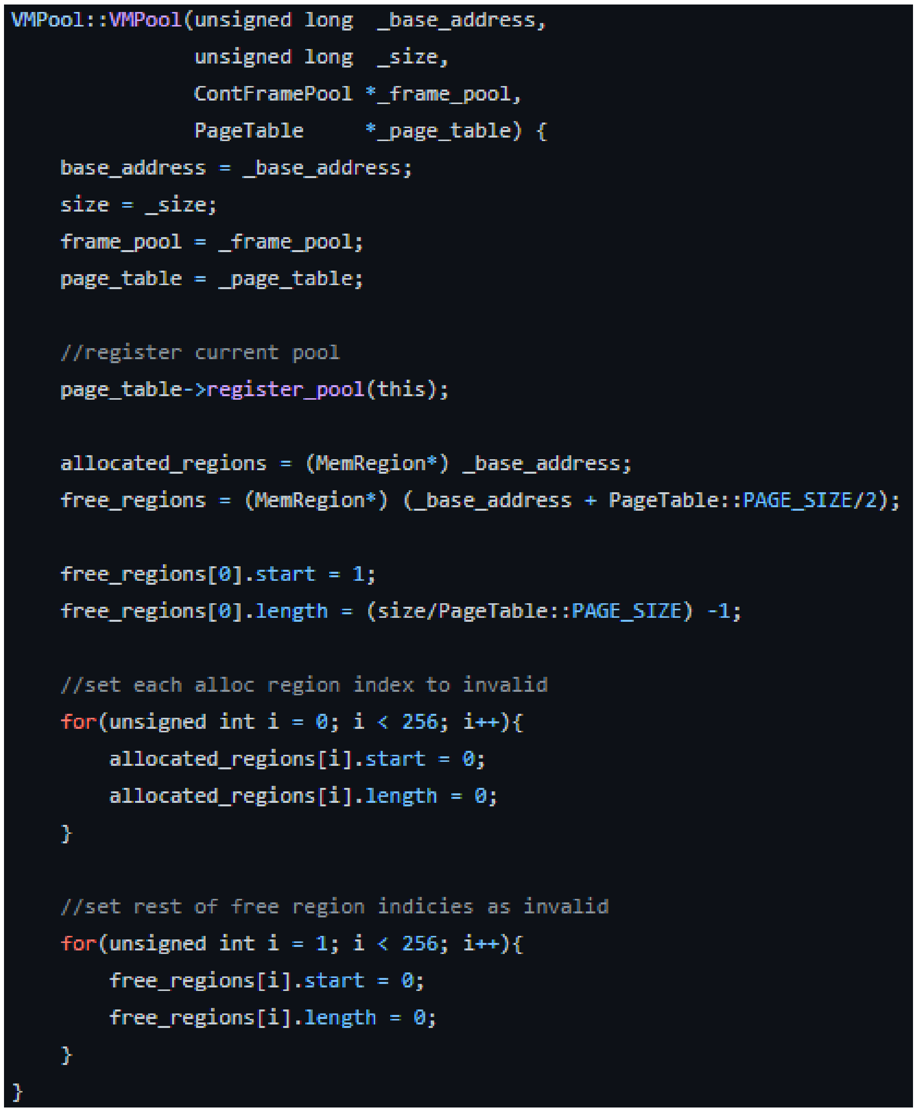
- Takes in page number as an unsigned long
- Computes page address by multiplying page no with page size
- Gets the PTE address of the page
- If the page is valid, release the frame at the address
- Flush the TLB by writing to CR3



```
void PageTable::free_page(unsigned long _page_no){
   unsigned long page_addr = _page_no * PAGE_SIZE;
   unsigned long* pte_addr = PTE_address(page_addr);

   //if valid, first part of pte is frame number
   if(*pte_addr & 0x1 == 1){
      ContFramePool::release_frames((*pte_addr>>12));
      //mark page table entry as invalid/ clear pte
      *pte_addr = 0;
      //flush the TLB
      write_cr3((unsigned long) page_directory);
   }
}
```

VMPool Constructor

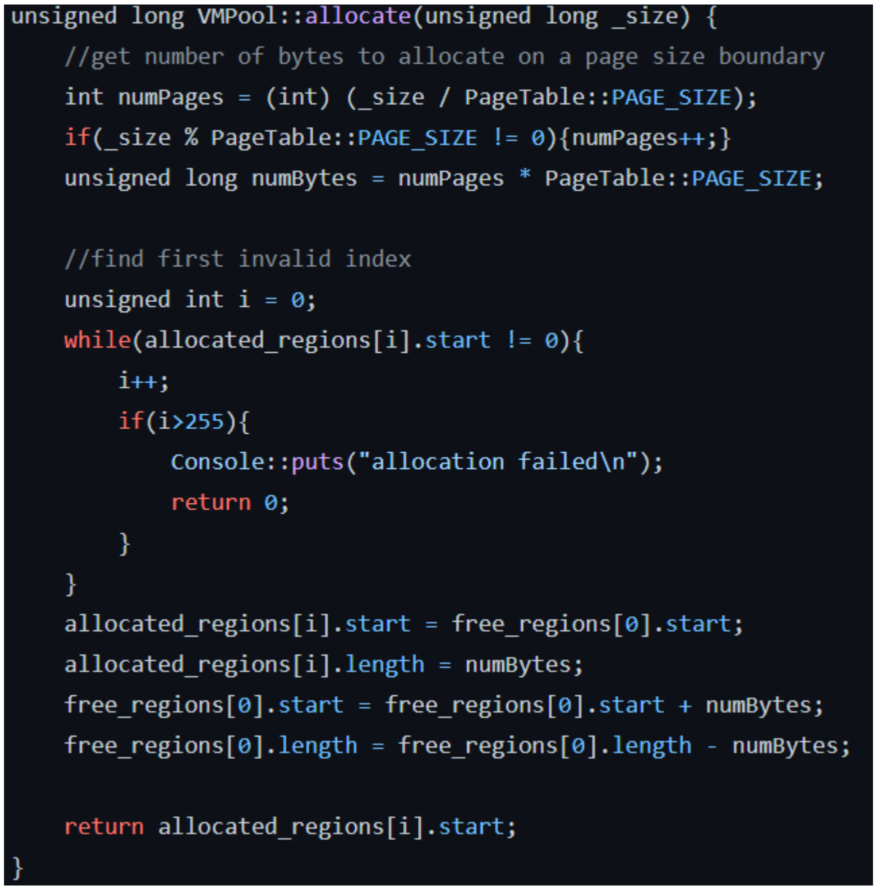
- Set private members
- Register the pool being created to the corresponding page table with register_pool()
- Create the arrays for the allocated and free regions
  - Pointer of custom Struct type MemRegion that holds start address and length as unsigned longs
  - Each array is half a page in size, so both fit in one frame
- Create first free region that takes up entire mem pool
- Set each index in the allocated regions array to invalid
  - A value of 0 for start and length is invalid because address 0 will never be used
- Set each index after the first in the free regions array to invalid



```
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;

    //register current pool
    page_table->register_pool(this);

    allocated_regions = (MemRegion*) _base_address;
    free_regions = (MemRegion*) (_base_address + PageTable::PAGE_SIZE/2);

    free_regions[0].start = 1;
    free_regions[0].length = (size/PageTable::PAGE_SIZE) -1;

    //set each alloc region index to invalid
    for(unsigned int i = 0; i < 256; i++){
        allocated_regions[i].start = 0;
        allocated_regions[i].length = 0;
    }

    //set rest of free region indicies as invalid
    for(unsigned int i = 1; i < 256; i++){
        free_regions[i].start = 0;
        free_regions[i].length = 0;
    }
}
```

Allocate

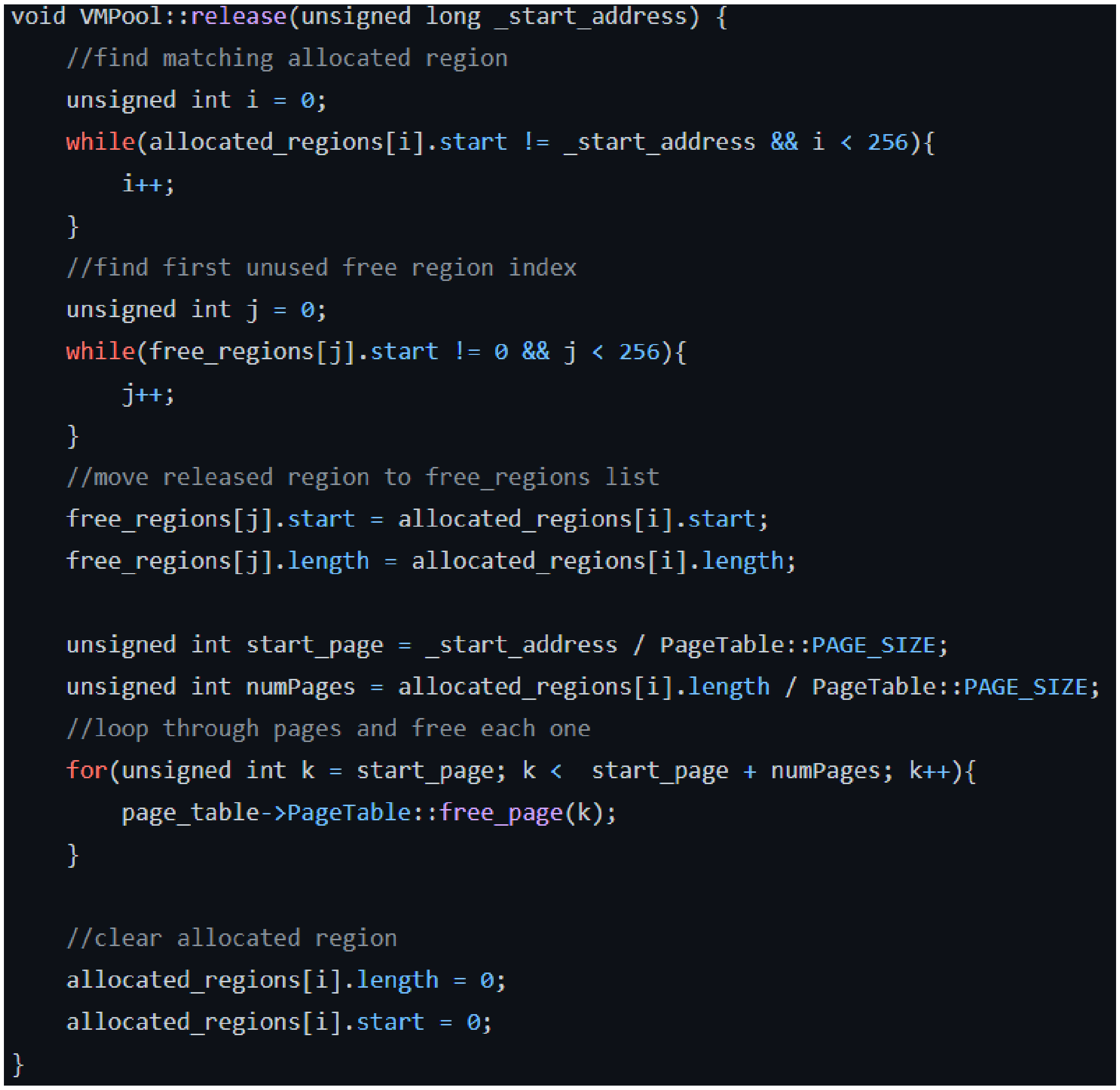
- Takes in size of region to allocate in bytes
- Calculate number of pages to allocate
    - Only allocate an amount of bytes that lies on a page size boundary
- Convert number of pages back into bytes
- Find the first available index in the allocated regions array
- Set the values of allocated region array at the index
    - Start address is the current start address of the front of the free list
    - Length is the calculated number of bytes
- Shift the first index of the free regions list by the number of bytes allocated
- Return the start address of the newly allocated region



```
unsigned long VMPool::allocate(unsigned long _size) {
    //get number of bytes to allocate on a page size boundary
    int numPages = (int) (_size / PageTable::PAGE_SIZE);
    if(_size % PageTable::PAGE_SIZE != 0){numPages++;}
    unsigned long numBytes = numPages * PageTable::PAGE_SIZE;

    //find first invalid index
    unsigned int i = 0;
    while(allocated_regions[i].start != 0){
        i++;
        if(i>255){
            Console::puts("allocation failed\n");
            return 0;
        }
    }
    allocated_regions[i].start = free_regions[0].start;
    allocated_regions[i].length = numBytes;
    free_regions[0].start = free_regions[0].start + numBytes;
    free_regions[0].length = free_regions[0].length - numBytes;

    return allocated_regions[i].start;
}
```

Release

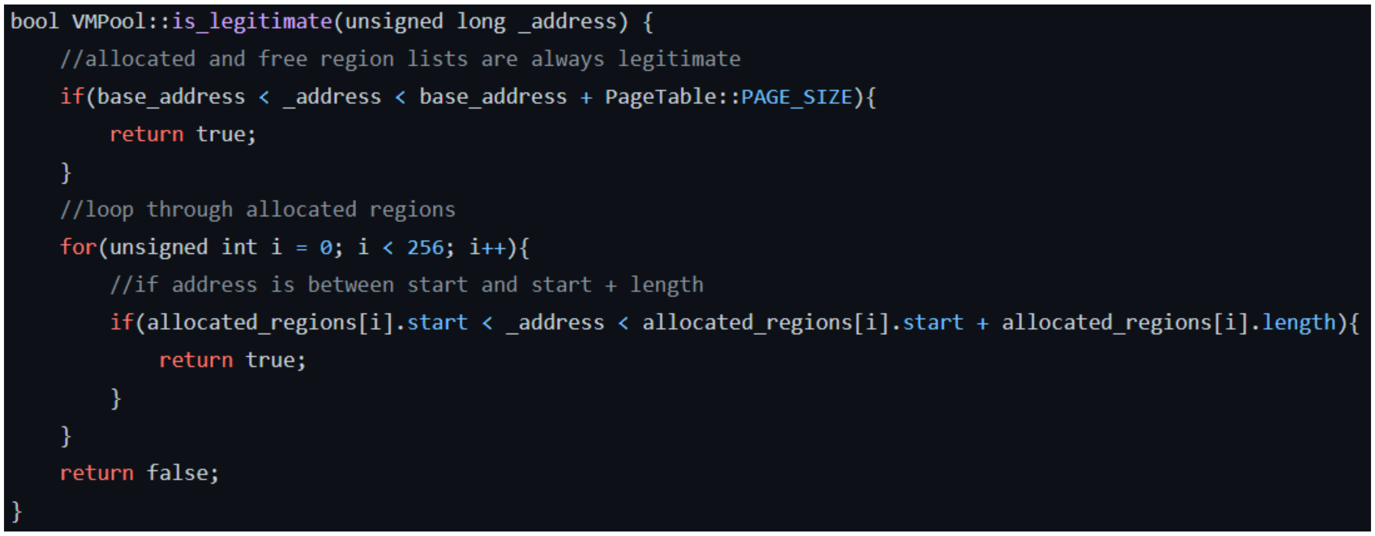
- Takes in the start address of the region to be released as an unsigned long
- Find the matching allocated region
    - Loop through regions to find the region with the same start address
- Find first unused free region index
- Move released region to the free region list
- Get the starting page number and the number of pages in the freed region
- Call free_page for each page in the released region
- Clear the allocated region from the allocated region list



```
void VMPool::release(unsigned long _start_address) {
    //find matching allocated region
    unsigned int i = 0;
    while(allocated_regions[i].start != _start_address && i < 256){
        i++;
    }
    //find first unused free region index
    unsigned int j = 0;
    while(free_regions[j].start != 0 && j < 256){
        j++;
    }
    //move released region to free_regions list
    free_regions[j].start = allocated_regions[i].start;
    free_regions[j].length = allocated_regions[i].length;

    unsigned int start_page = _start_address / PageTable::PAGE_SIZE;
    unsigned int numPages = allocated_regions[i].length / PageTable::PAGE_SIZE;
    //loop through pages and free each one
    for(unsigned int k = start_page; k <  start_page + numPages; k++){
        page_table->PageTable::free_page(k);
    }

    //clear allocated region
    allocated_regions[i].length = 0;
    allocated_regions[i].start = 0;
}
```

Is_legitimate

- Takes in an address as an unsigned long
- Return true if the address is used for accessing the free and allocated regions lists
- Loop through the regions and return true if the address lies in the range of any one memory region
- Else return false
-



```
bool VMPool::is_legitimate(unsigned long _address) {
    //allocated and free region lists are always legitimate
    if(base_address < _address < base_address + PageTable::PAGE_SIZE){
        return true;
    }
    //loop through allocated regions
    for(unsigned int i = 0; i < 256; i++){
        //if address is between start and start + length
        if(allocated_regions[i].start < _address < allocated_regions[i].start + allocated_regions[i].length){
            return true;
        }
    }
    return false;
}
```

Testing

My testing for this machine problem consisted of mainly the provided tests in kernel.C. I tested the page table with _TEST_PAGE_TABLE_ defined as well as the VMPools with it undefined. I was not worried about my page table implementation, because without creating virtual memory pools, my page table implementation should stay the same as it was in MP3 for the most part. Testing the VMPools was the more important part to me, and I read through the given test cases carefully to fully understand their scope. I did some print statement testing as well, where I added console print statements at key points in the code to make sure certain values were being set correctly. I am fairly confident that my testing was sufficient for this machine problem. I am ignoring cases where the user purposefully allocated more memory than the system can handle as that would require significant changes to the scope of the project and we can assume the user would understand the limitations of the system.