MP4: Virtual Memory Management and Memory Allocation

Ajay Jagannath
UIN: 236002822
CSCE611: Operating System

Assigned Tasks

Part I: Support for Large Address Spaces (Recursive Paging) - Completed.
Part II: Preparing PageTable for VM Pools (Registration, Freeing) - Completed.
Part III: Virtual Memory Allocator (VMPool) - Completed.

Files modified: page_table.H, page_table.C, vm_pool.C, vm_pool.H, kernel.C (for testing).

System Design

The goal of this machine problem is to complete the memory manager by extending our MP3 solution in three main parts.

Part I addresses the problems in scalability of storing the page directory (PD) and page tables (PTs) directly in the small, direct-mapped kernel pool (4MB). The solution is to move these tables into the much larger "process memory pool" (which are not directly mapped memory). This creates a new challenge: once paging is enabled, the CPU can no longer access the physical addresses of the PD and PTs (as it always translates addresses into logical memory only). To solve this, we implement the "recursive page table lookup" scheme as thought in class. The last entry (1023) of the page directory is set to point to the physical address of the page directory itself. This allows the kernel to access any Page Directory Entry (PDE) or Page Table Entry (PTE) using a altered virtual address, even after paging is active. A virtual address of the form '(1023 — 1023 — X — 00)' maps back to the page directory, and an address of the form '(1023 — X — Y — 00)' maps to the page table managed by PDE 'X'. This is essential for the page fault handler to create new PTs and pages on demand.

Part II changes the 'PageTable' class to support virtual memory pools. The 'PageTable' now maintains a linked list of all active 'VMPool' objects. This allows us to verify if a memory access is "legitimate" (i.e., part of a region allocated by a 'VMPool') whenever we want to release it. This part also contains the 'free_page' function, which is called by a 'VMPool' during deallocation. This function is used for finding the physical frame associated with a virtual page, releasing it back to the frame pool, marking the corresponding PTE as "not present", and then flushing the TLB.

Part III has the implementation of a 'VMPool' class, which acts as a simple virtual memory allocator. This class manages a large, contiguous region of virtual memory. It uses the first two pages of its own pool to store its management data structures: an 'alloc_list' and a 'free_list'. When 'allocate' is called, it finds a suitable chunk from the 'free_list', moves it to the 'alloc_list', and then "page faults" every page in the new region. This is a lazy allocation done on demand that triggers the 'PageTable::handle_fault' mechanism. When 'release' is called, it iterates over all pages in the region, calling 'PageTable::free_page' for each, before returning the virtual memory chunk to the 'free_list'.

Code Description

Part I: Large Address Spaces (page_table.C)

PageTable - constructor : The constructor is updated for Part I to allocate all paging structures from the 'process_mem_pool'. First, it requests a frame for the Page Directory (PD). It then requests another frame for the first Page Table (PT), which will handle the 1-to-1 mapping for the first 4MB of kernel space. It populates this first PT with 1-to-1 mappings. The remaining 1023 entries in the PD are marked as "not present" (but writable). The most important change is the setup for recursive mapping: the very last entry of the page directory ('page_directory[1023]') is set to point to the physical address of the page directory frame itself. This enables the recursive lookup mechanism used in 'handle_fault'.

```
PageTable::PageTable()
{
   // Two modes: paging enabled and paging disabled
   // We handle only the Paging disabled mode here.
   // With paging Enabled, we wont have direct access to physical memory, so we handle that inside th
   // Paging Disabled: initalize a frame for page directory in kernel pool and assign
   // Also initalize a page for kernel memory and do one to one mapping
   head_pool = nullptr;
   // Page Directory will be stored in kernel space - need 4KB (2^10 entries of 4byte each) => 1 fram
   unsigned long pd_frame_no = process_mem_pool->get_frames(1);

   // Stop if no frame available
   assert(pd_frame_no != 0);
   page_directory = (unsigned long *)(pd_frame_no * PAGE_SIZE);

   // now page directory points to an address and starting from that address we can access any of the
   // directly like an array ( were each entry in array is 4 bytes and shares an address - thats why

   unsigned long pt_frame_no = process_mem_pool->get_frames(1);
   assert(pt_frame_no != 0);
   unsigned long *first_page_table = (unsigned long *)(pt_frame_no * PAGE_SIZE);
   // mapping first page table to first entry of page directory (so that it governs first 4MB of memo
   // also need to set valid and R&W bits.
   page_directory[0] = (unsigned long)first_page_table | 3;

   // one to one mapping for all kernel physical space to the pagetable rows (and hence the pages)
   for (unsigned long pno = 0; pno < ENTRIES_PER_PAGE; pno++)
   {
      first_page_table[pno] = (unsigned long)(pno * PAGE_SIZE) | 3;
   }

   // Map the remaining page tables to the page directory (keeping valid bit = 0)
   for (unsigned long pdno = 1; pdno < ENTRIES_PER_PAGE; pdno++)
   {
      page_directory[pdno] = 0 | 2;
      // Recursive Mapping - last entry points to page directory itself
      if (pdno == ENTRIES_PER_PAGE - 1)
      {
         page_directory[pdno] = (unsigned long)(pd_frame_no * PAGE_SIZE) | 3;
      }
   }

   Console::puts("Constructed Page Table object\n");
}
```

handle_fault : This static method is the core of our demand paging system. When a page fault occurs (because the 'Present' bit is 0), this handler is invoked. It first reads the faulting virtual address from the 'CR2' register. We also add an extra check to assert that the page in question is legitimate (talked about in part 3).

1. **Find/Create Page Table:** It uses recursive mapping to find the virtual address of the Page Directory Entry (PDE) corresponding to the faulting address. The address (1023 — 1023 — 0) is used as a base. If this PDE is marked "not present", it means the page table for that 4MB region does not exist. The handler allocates a new frame for the PT from the 'process_mem_pool', updates the PDE to point to it, and marks it present. It then uses another recursive address (1023 — X — Y — 00) to get a virtual pointer to the new PT and initializes all its entries as "not present".
2. **Find/Create Page:** Once the page table is guaranteed to exist, the handler finds the virtual address of the Page Table Entry (PTE) using the recursive base address (1023 — X — Y — 00). It allocates a new frame from the 'process_mem_pool' for the user's data page, updates the PTE to point to this new frame, and marks it as "present," "writable," and "user-accessible" (flags = 7).

The handler then returns, and the CPU re-executes the instruction that caused the fault, which now succeeds.

```
void PageTable::handle_fault(REGS *_r)
{
   // for now, we cant handle a protection fault
   assert(!(_r->err_code & 1)); // P bit must be 0

   // get the virtual address in question from cr2 register
   unsigned long cr2 = read_cr2();

    // ensure the faulting address belongs to a registered VM pool
   // uncomment for part 2 and 3 when access only through VM pool.
   // assert(current_page_table != nullptr);
   // bool check = false;
   // VMPool *node = current_page_table->head_pool;
   // while (node != nullptr && !check)
   // {
   //    check = node->is_legitimate(cr2);
   //    node = node->next_pool;
   // }
   // assert(check);

   // Now since no one to one mapping exists for process memory
   // CPU cant access page directory properly, so we need to access it via recursive mapping
   // we do this by accessing the last 4MB of virtual memory which maps to page directory itself
   // needs to be (1023 | 1023 | X (PDE) | 00)
   unsigned long *pde = (unsigned long *)(((cr2 >> 22) << 2) | 0xFFFFF000);
   if (!(*pde & 1))
   {
      // The page table for this 4MB region does not exist. We need to create it.
      // Allocate a frame for the new page table from kernel space
      unsigned long new_pt_frame = process_mem_pool->get_frames(1);
      assert(new_pt_frame != 0);

      *pde = (unsigned long)(new_pt_frame * PAGE_SIZE) | 7;

      // now to traverse the PT, we get the virtual address of PT first with recursive mapping.
      // it should be (1023 | X (PDE) | 0 | 00)
      // a pointer to this new page table's virtual memory.
```

```
        unsigned long *page_table = (unsigned long *)(((cr2 >> 22) << 12) | 0xFFC00000);

        for (unsigned int i = 0; i < ENTRIES_PER_PAGE; i++)
        {
            // Marked as invalid, but writable and user-accessible
            *page_table = 2 | 4;
            // Recursive Mapping - last entry points to page table itself
            if (i == ENTRIES_PER_PAGE - 1)
            {
                *page_table = (unsigned long)(new_pt_frame * PAGE_SIZE) | 7;
            }
            page_table++;
        }
    }

    // Again for recursive mapping
    // it should be (1023 | X (PDE) | Y (PTE) | 00)
    unsigned long *pte = (unsigned long *)(((cr2 >> 12) << 2) | 0xFFC00000);

    // Allocate a frame
    unsigned long new_page_frame = process_mem_pool->get_frames(1);
    assert(new_page_frame != 0);
    *pte = (new_page_frame * PAGE_SIZE) | 7;

    // no need to do load() - that is only for context switching

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

Part II: VM Pool Support (page_table.H and page_table.C)

Data Members : Private member 'head_pool' to store the top of the linked list of VM_pool objects.

```
VMPool * head_pool;
```

register_pool : This function is called by the 'VMPool' constructor to register itself with the 'PageTable'. It implements a simple linked list by adding the new '_vm_pool' to the 'head_pool' variable. This list is later traversed by 'free_page' to check for address legitimacy.

```
void PageTable::register_pool(VMPool *_vm_pool)
{
    _vm_pool->next_pool = head_pool;
    head_pool = _vm_pool;
    //  assert(false);
    Console::puts("registered VM pool\n");
}
```

free_page : This function is called by 'VMPool::release' to deallocate a page. It first traverses the linked list of registered VM pools, calling 'is_legitimate()' on each one to ensure the page being freed is indeed part of an allocated region. If it is legitimate, it calculates the virtual address of the page's PTE using the same recursive mapping as 'handle_fault'. It reads the PTE to get the physical frame number, releases that frame back to the 'process_mem_pool', and marks the PTE as "not present" by clearing its 'Present' bit. Finally, it reloads the 'CR3' register to flush the TLB, which is essential to prevent the CPU from using a stale translation.

```
void PageTable::free_page(unsigned long _page_no)
{
```

```
   // check if _page_no is legitimate as it is coming from release
   bool check=false;
   unsigned long page_address = _page_no * PAGE_SIZE;
   VMPool * node = head_pool;
   while(node!=nullptr && !check){
      check = node->is_legitimate(page_address);
      node = node->next_pool;
   }
   // check if page not legit in any pools
   assert(check);
   // get frame number from page number
   unsigned long virtual_address = _page_no * PAGE_SIZE;
   // recursive mapping
   unsigned long *pte = (unsigned long *)(((virtual_address >> 12) << 2) | 0xFFC00000);
   unsigned long frame_number = (*pte)/PAGE_SIZE;
   // release frame;
   process_mem_pool->release_frames(frame_number);
   //mark pte invalid
   *pte &= ~0x00000001;

   //flush TLB by reloading CR3 with load
   this->load();

   // assert(false);
   Console::puts("freed page\n");
}
```

Part III: Virtual Memory Allocator (vm_pool.H and vm_pool.C)

Data Members : Members defined to handle the incoming arguments of constructor, as well as a linked pointers to link the VM_pools for the page table.

```
private:
   /* -- DEFINE YOUR VIRTUAL MEMORY POOL DATA STRUCTURE(s) HERE. */
   unsigned long          base_address;
   unsigned long          size;
   ContFramePool        * frame_pool;
   PageTable            * page_table;
   unsigned long        * alloc_list;
   unsigned long        * free_list;


public:
   VMPool *next_pool;
```

VMPool - constructor : The ‘VMPool‘ constructor initializes the pool’s base address and size (rounding the size up to a multiple of ‘PAGE_SIZE‘). It then calls ‘page_table->register_pool(this)‘ to register itself. It designates the first two pages of its own virtual address range to store the ‘alloc_list‘ and ‘free_list‘, as suggested in the handout. It then immediately accesses the base addresses of these two lists, which intentionally triggers two page faults. This forces the ‘PageTable::handle_fault‘ handler to allocate physical frames for these management structures. Finally, it initializes the ‘alloc_list‘ to contain these first two pages and the ‘free_list‘ to contain the rest of the pool’s memory as one large chunk.

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

```
{
    // Initialize the data structures
    base_address = _base_address;
    // Round up size to multiple of page size
    if (_size % PageTable::PAGE_SIZE != 0)
    {
        _size = (_size / PageTable::PAGE_SIZE + 1) * PageTable::PAGE_SIZE;
    }
    size = _size;
    frame_pool = _frame_pool;
    page_table = _page_table;
    next_pool = nullptr;

    // register this pool with the page table
    page_table->register_pool(this);
    // get frames for the 2 list
    alloc_list = (unsigned long *)base_address;
    free_list = (unsigned long *)(base_address + PageTable::PAGE_SIZE);
    // Accessing these addresses triggers page faults that alots frames to these pages
    // start_address and length for alloc_list page
    alloc_list[0] = base_address;
    alloc_list[1] = PageTable::PAGE_SIZE;
    // start_address and length for free_list page
    alloc_list[2] = base_address + PageTable::PAGE_SIZE;
    alloc_list[3] = PageTable::PAGE_SIZE;
    // set other entries to 0
    for (unsigned long i = 4; i < PageTable::ENTRIES_PER_PAGE - 1; i += 2)
    {
        alloc_list[i] = 0;
        alloc_list[i + 1] = 0;
    }
    // Remove these from free_list
    free_list[0] = base_address + (2 * PageTable::PAGE_SIZE);
    free_list[1] = size - (2 * PageTable::PAGE_SIZE);
    // set other entries to 0
    for (unsigned long i = 2; i < PageTable::ENTRIES_PER_PAGE - 1; i += 2)
    {
        free_list[i] = 0;
        free_list[i + 1] = 0;
    }

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

allocate : This function implements an ”lazy” allocation strategy. First, it rounds the requested ‘_size‘ up to the nearest page size. It then performs a first-fit search on the ‘free_list‘ to find a chunk large enough. If a chunk is found, it is split (if necessary), and the allocated portion is added as a new entry to the ‘alloc_list‘. Inside the loop, it reads the value at each page-aligned address (‘*((unsigned long *)i)‘). This access faults every page in the new region, forcing the ‘PageTable‘ to allocate and map all required physical frames. It then returns the ‘allocated_start‘ address.

```
unsigned long VMPool::allocate(unsigned long _size)
{
    if (_size % PageTable::PAGE_SIZE != 0)
    {
        _size = (_size / PageTable::PAGE_SIZE + 1) * PageTable::PAGE_SIZE;
    }
```

```
    // Traverse throught the free list to get a chunk that fits it requirement;
    unsigned long iter = 0;
    unsigned long allocated_start;
    // find a chunk that fits
    while (free_list[iter + 1] < _size)
    {
        iter += 2;
        // Assert to ensure we dont exit the page
        assert(iter < PageTable::ENTRIES_PER_PAGE - 1);
    }
    // save the chunk that fits
    allocated_start = free_list[iter];
    // if less than size of free list entry, just edit the chunk out;
    if (free_list[iter + 1] > _size)
    {
        free_list[iter] += _size;
        free_list[iter + 1] -= _size;
    }
    // else is equal to free list entry, shift everything up;
    else
    {
        for (unsigned long i = iter + 2; i < PageTable::ENTRIES_PER_PAGE - 1; i += 2)
        {
            free_list[i - 2] = free_list[i];
            free_list[i - 1] = free_list[i + 1];
            // clear the last entry if we have shifted everything
            if (i + 2 >= PageTable::ENTRIES_PER_PAGE - 1)
            {
                free_list[i] = 0;
                free_list[i + 1] = 0;
            }
        }
    }
    // traverse to the end of alloc list and add this chunk there
    iter = 0;
    while (alloc_list[iter + 1] != 0)
    {
        iter += 2;
        // atleast one space must be free (to add this chunk)
        assert(iter < PageTable::ENTRIES_PER_PAGE - 3);
    }
    alloc_list[iter] = allocated_start;
    alloc_list[iter + 1] = _size;
    // access all pages in this region to trigger page faults and allocate frames
    for (unsigned long i = allocated_start; i < allocated_start + _size; i += PageTable::PAGE_SIZE)
    {
        unsigned long temp = *((unsigned long *)i);
    }
    // assert(false);
    Console::puts("Allocated region of memory.\n");
    return allocated_start;
}
```

release : This function frees a previously allocated region. It finds the region in the ‘alloc_list‘ matching the ‘_start_address‘. It then iterates from the ‘free_start‘ address to ‘free_start + free_length‘, calculating the page number for each page in the region and calling ‘page_table->free_page()‘ on it.

This reclaims all physical frames and invalidates the PTEs. After all pages are freed, it removes the entry from the ‘alloc_list‘ and adds the entire region back to the ‘free_list‘ (without coalescing, for simplicity).

```
void VMPool::release(unsigned long _start_address)
{
    unsigned long iter = 0;
    while (alloc_list[iter] != _start_address)
    {
        iter += 2;
        // Assert to ensure we dont exit the page
        assert(iter < PageTable::ENTRIES_PER_PAGE - 1);
    }
    // save the chunk to be removed
    unsigned long free_start = alloc_list[iter];
    unsigned long free_length = alloc_list[iter + 1];
    assert(free_start % PageTable::PAGE_SIZE == 0);
    assert(free_length % PageTable::PAGE_SIZE == 0);

    // freeing the frames for these pages
    for (unsigned long i = free_start; i < free_start + free_length; i += PageTable::PAGE_SIZE)
    {
        unsigned long page_no = i / PageTable::PAGE_SIZE;
        page_table->free_page(page_no);
    }

    // remove chunk form alloc list and shift entries up
    alloc_list[iter] = 0;
    alloc_list[iter + 1] = 0;
    for (unsigned long i = iter + 2; i < PageTable::ENTRIES_PER_PAGE - 1; i += 2)
    {
        alloc_list[i - 2] = alloc_list[i];
        alloc_list[i - 1] = alloc_list[i + 1];
        // clear the last entry if we have shifted everything
        if (i + 2 >= PageTable::ENTRIES_PER_PAGE - 1)
        {
            alloc_list[i] = 0;
            alloc_list[i + 1] = 0;
        }
    }
    // add chunk to free list
    iter = 0;
    while (free_list[iter + 1] != 0)
    {
        iter += 2;
        // atleast one space must be free (to add this chunk)
        assert(iter < PageTable::ENTRIES_PER_PAGE - 3);
    }
    // assign chunk to first empty space in free list
    free_list[iter] = free_start;
    free_list[iter+1]= free_length;


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

is_legitimate : This is a helper function used by ‘PageTable::free_page‘ (and tested in ‘kernel.C‘). It performs a simple linear scan of the ‘alloc_list‘. For each entry, it checks if the provided ‘_address‘ falls within the allocated range (‘[alloc_list[i], alloc_list[i] + alloc_list[i+1] - 1]‘). It returns ‘true‘ if the address is found in any allocated region, and ‘false‘ otherwise.

```
bool VMPool::is_legitimate(unsigned long _address)
{
    // Metadata pages (alloc_list and free_list) must always be treated as valid
    unsigned long alloc_base = (unsigned long)alloc_list;
    if (_address >= alloc_base && _address < alloc_base + PageTable::PAGE_SIZE)
        return true;
    unsigned long free_base = (unsigned long)free_list;
    if (_address >= free_base && _address < free_base + PageTable::PAGE_SIZE)
        return true;

    // Traverse through the allocated list to check if _address is allocated
    for (unsigned long i = 0; i < PageTable::ENTRIES_PER_PAGE - 1; i += 2)
    {
        if ((_address >= alloc_list[i]) && (_address < alloc_list[i] + alloc_list[i + 1]))
            return true;
    }
    return false;
    // assert(false);
    Console::puts("Checked whether address is part of an allocated region.\n");
}
```

Testing

Default Testing for MP4 is divided by the into 2 parts in ‘kernel.C‘.

Part I Test (_TEST_PAGE_TABLE_ defined)

This function accesses memory starting at ‘FAULT_ADDR‘ (4MB). This directly tests the ”demand paging” capability of the ‘PageTable::handle_fault‘ function. The handler must correctly use recursive lookups to create new page tables and data pages (all from the ‘process_mem_pool‘) to satisfy these requests.

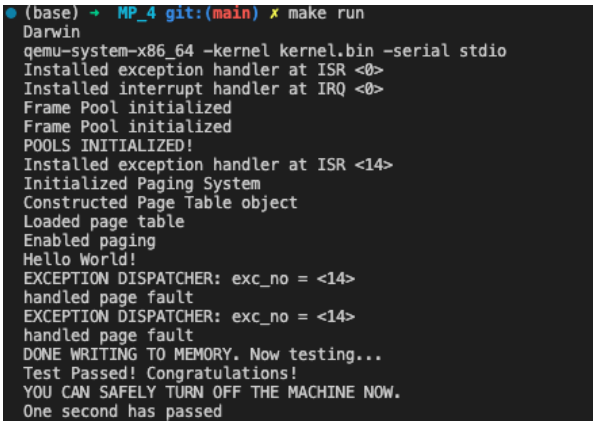

Figure 1: Default Testing Output 1

Part II & III Test (_TEST_PAGE_TABLE_ commented out)

It runs ‘GenerateVMPoolMemoryReferences‘ on two different ‘VMPool‘ objects (‘code_pool‘ and ‘heap_pool‘). This test does the following:

1. Calls ‘new[]‘, which triggers ‘VMPool::allocate‘.
2. ‘VMPool::allocate‘ pre-faults all pages, testing ‘PageTable::handle_fault‘.
3. The test explicitly calls ‘pool->is_legitimate()‘ to verify the ‘VMPool‘’s internal tracking.
4. The test writes to and reads from the allocated memory (‘arr[j] = j;‘), verifying that the pages are correctly mapped and writable.
5. The test calls ‘delete[]‘, which triggers ‘VMPool::release‘.
6. ‘VMPool::release‘ calls ‘PageTable::free_page‘ for every page, testing the frame release, PTE invalidation, and TLB flush mechanisms.

Because VMPool::allocate() rounds every request up to a full page, the number of pages touched grows in steps:

i = 1 … 10 → 400–4000B → 1 page
i = 11 … 20 → 4400–8000B → 2 pages
i = 21 … 30 → 8400–12000B → 3 pages
i = 31 … 40 → 12400–16000B → 4 pages
i = 41 … 49 → 16400–20000B → 5 pages

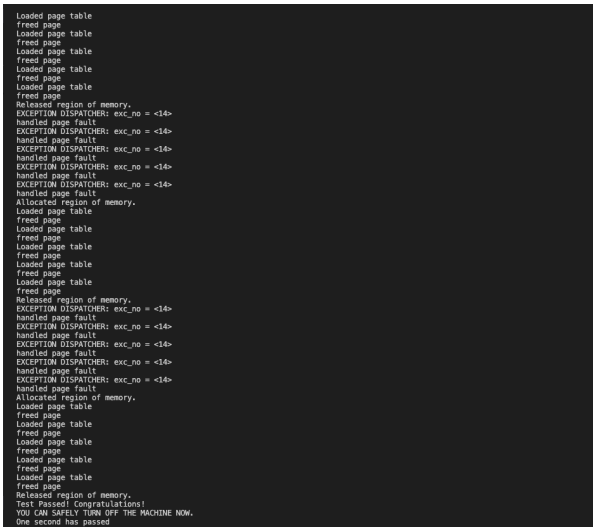

Figure 2: Default Testing Output 2

Custom Test:

Custom test in kernel.C allocates a tiny array from code_pool, checks that the pool flags the returned pointer as legitimate while a heap_pool does not, then frees the allocation to confirm cross-pool isolation stays intact. (Touching probe can be toggled on to force page faulting before delete[].)

```
void CustomTests(VMPool* pool_primary, VMPool* pool_other)
{
        current_pool = pool_primary;
        int* probe = new int[16]; // tiny alloc
        if (!pool_primary->is_legitimate((unsigned long)probe)) {
```

```
                Console::puts("Error: primary::is_legitimate returned false for its own allocation!\n
        }
        if (pool_other && pool_other->is_legitimate((unsigned long)probe)) {
                Console::puts("Error: Cross-pool isolation FAILED (other pool accepted primary addres
        }
        //since we are doing lazy allocation, ideally we need to access memory before deletion
        // for (int i = 0; i < 16; i++) {
        //         probe[i] = i;
        // }
        delete[] probe;
        Console::puts("Cross-pool legitimacy isolation passed.\n");
}
```

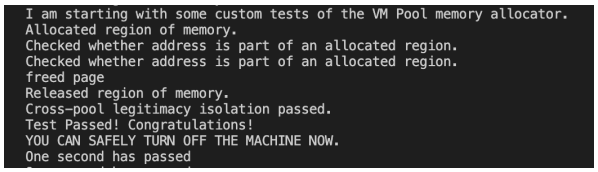

Figure 3: Custom Testing Output