MP4: Virtual Memory Management and Memory Allocation

Vaibhav Pundir
UIN: 734004197
CSCE611: Operating System

Assigned Tasks

Main: Completed.

System Design

The objective of this machine problem is to upgrade page table management to support virtual memory and implement a virtual memory allocator. In this task, we allocate page table pages in mapped memory, specifically memory above 4 MB, which is managed by the process frame pool.

Due to the limited size of directly mapped memory, it cannot accommodate larger address spaces. Therefore, we utilize the process memory pool to allocate memory for page table pages. Additionally, when paging is activated, the CPU generates logical addresses. Thus, to map logical addresses to frames and adjust entries in the page directory and page table pages, we employ a method known as 'Recursive Page Table Lookup.'

In the 'Recursive Page Table Lookup' approach, the final entry of the page directory points to the beginning of the page directory itself. Both the page directory and page table pages contain physical addresses.

To access a page directory entry, we utilize the following logical address format: The MMU (Memory

1023: 10	1023: 10	offset: 12

Management Unit) utilizes the initial 10 bits (value 1023) to index into the page directory for the PDE (Page Directory Entry). The PDE number 1023 (the last one) directs to the page directory itself. Subsequently, the MMU uses the subsequent 10 bits to index into the supposed page table page for the PTE (Page Table Entry). Since both sets of 10 bits have a value of 1023, the resulting PTE also points to the page directory itself. Consequently, the MMU treats the page directory like any other page table page, utilizing the offset to index into the physical frame.

To access a page table page entry, the following logical address format is utilized: The MMU

1023: 10	X: 10	Y: 10	O: 2

employs the initial 10 bits (value 1023) to index into the page directory for the PDE. Similar to before, PDE number 1023 points to the page directory itself. Subsequently, the MMU uses the subsequent 10 bits (value X) to index into the supposed page table page for the PTE (which is essentially the Xth PDE). The offset is then utilized to index into the supposed physical frame, which in reality is the page table page associated with the Xth directory entry. Consequently, the remaining 12 bits are used to index into the Yth entry in the page table page.

This method allows manipulation of a page directory entry or page table page entry stored in virtual memory.

The page table needs to be aware of all created virtual memory pools to differentiate between legitimate and invalid memory accesses. To facilitate this, the page table requires a list of all virtual memory pools. Furthermore, upon the occurrence of a page fault, we verify the legitimacy of the address and subsequently release previously allocated pages.

Code Description

For this machine problem, I've made changes to the following code files:

1. cont_frame_pool.C
2. page_table.H
3. page_table.C
4. vm_pool.H
5. vm_pool.C

cont_frame_pool.C

ContFramePool::release_frames() : Updated the release_frames code to stop freeing the frames on encountering a frame start other than "Used". The function will find the frame pool to which the _first_frame_no belongs to and will check if it's in "HoS" state and will continue free the consecutive frames which are in "Used" state.

```
void ContFramePool::release_frames(unsigned long _first_frame_no)
{
    ContFramePool * node = ContFramePool::head;
    // iterate over the linked list to find the pool to which _first_frame_no belongs to
    for(; node != nullptr; node = node->next) {
        // if the _first_frame_no is in the range of current frame pool
        if(_first_frame_no >= node->base_frame_no && _first_frame_no < node->base_frame_no + node->n_frames) {
            unsigned long n = _first_frame_no - node->base_frame_no;
            if(node->get_state(n) == FrameState::HoS) {
                node->set_state(_first_frame_no, FrameState::Free);
                // if _first_frame_no is Head of Sequence, start releasing subsequent used frames
                for(unsigned long fno = _first_frame_no + 1; fno < _first_frame_no + node->n_frames; ++fno) {
                    if(node->get_state(fno - node->base_frame_no) == FrameState::Used) {
                        node->set_state(fno, FrameState::Free);
                        node->n_free_frames += 1;
                    } else {
                        break;
                    }
                }
            } else {
                Console::puts("ContFramePool::release_frames - {_first_frame_no} is not in HoS frame state!\n");
                // assert(false);
            }
            break;
        }
    }
```

Listing 1: C++ code snippet

page_table.H

```
private:
    ...
    static VMPool    * vm_pool_list;       /* A Linked List for Virtual Memory Pool */
    ...
```

Listing 2: C++ code snippet

page_table.C

PageTable::init_paging() : Initializes the kernel & process memory pool along with the shared size for the paging subsystem.

```
void PageTable::init_paging(ContFramePool * _kernel_mem_pool,
                            ContFramePool * _process_mem_pool,
                            const unsigned long _shared_size) {

    kernel_mem_pool = _kernel_mem_pool;
    process_mem_pool = _process_mem_pool;
    shared_size = _shared_size;

    Console::puts("Initialized Paging System\n");
}
```

Listing 3: C++ code snippet

PageTable::PageTable() : This method is the constructor and configures the page table and page directory entries. It allocates one frame from the kernel frame pool to create the page directory. To enable recursive page table lookup, the last element in the page directory points back to the start of the page directory. Using bitwise operations, the first and last page-directory entries are marked as "valid" and the remaining entries as "invalid". Similarly, one frame is allocated from the process frame pool to initialize the page table. With bitwise operations, the page table entries for the directly mapped first 4MB of shared memory are marked as "valid."

```
PageTable::PageTable() {
    // get frames for page_directory from kernel_mem_pool
    page_directory = (unsigned long *)(kernel_mem_pool->get_frames(1) * PAGE_SIZE);
    // compute the number of shared frames
    unsigned long no_shared_frames = PageTable::shared_size / PAGE_SIZE;
    page_directory[no_shared_frames-1] = (unsigned long)page_directory | 0b11;

    // get frames for page_table from process_mem_pool
    unsigned long *page_table = (unsigned long *)(process_mem_pool->get_frames(1) *
    PAGE_SIZE);

    // initializing page directory entries
    // mark first pde as valid
    // setting supervisor level, read/write and present bits
    page_directory[0] = (unsigned long)page_table | 0b11;

    // mark the rest of the pde as invalid
    // i.e. do not set present field
    for(unsigned int i=1; i<no_shared_frames-1; ++i)
        page_directory[i] |= 0b10;

    // map initial 4MB for page table
    // mark as valid
    for(unsigned int i=0; i<no_shared_frames; ++i)
        page_table[i] = PAGE_SIZE * i | 0b11;

    // initially, disable paging
    paging_enabled = 0;

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

Listing 4: C++ code snippet

void PageTable::load() : Loads the page table by writing the page directory's address into register CR3 (page table base register) and the current_page_table variable points to this table.

```
void PageTable::load() {
    current_page_table = this;
    // store the address of page directory in register CR3
    write_cr3((unsigned long)(current_page_table->page_directory));

    Console::puts("Loaded page table\n");
}
```

Listing 5: C++ code snippet

void PageTable::enable_paging() : Enables paging on the CPU. To enable paging, we set the 32nd bit and store it in register CR0, and set the paging_enabled flag.

```
void PageTable::enable_paging() {
    // set the 32nd-bit (paging bit)
    write_cr0(read_cr0() | 0x80000000);
    // enabling paging
    paging_enabled = 1;

    Console::puts("Enabled paging\n");
}
```

Listing 6: C++ code snippet

PageTable::handle_fault() : Handles CPU's page-fault exception. To determine how to handle the page fault, this procedure will search through the page table for the relevant entry.

If there isn't a physical memory frame linked to the page:

- An available frame is brought in and,
- The page table entry is updated.

If a new page table needs to be initialized:

- A new frame is allocated and,
- The new page table's page and directory are updated.

The logic is as follows:

- Check if 0th-bit of err_code is not set (0th-bit should be unset for "page not present" exception)
- Read the address of page-fault from register CR2 and page directory address from register CR3
- Compute the page directory index by reading the first 10-bits of page-fault address
- Compute the page table index by reading the next 10-bits of page-fault address
- Check the 'Present Field' which is the 0th-bit in the page directory address
- If 0th-bit is unset (page-fault occurred in page directory)
  - allocate a new frame to the page table from process_mem_pool
  - mark the page directory entry as valid via bitwise operation,
  - and mark all the page table entries as invalid via bitwise operations
- Then, for both the cases, page-fault in page directory or page-fault in page-level, execute:
  - allocate a new frame to the page directory entry from process_mem_pool
  - mark the page table entry in the page table as valid via bitwise operation

```
void PageTable::handle_fault(REGS * _r) {
  // check if it is page not present exception
  if ((_r->err_code & 1) == 0) {
    // get page-fault address
        unsigned long address = read_cr2();

        // get address of page directory
        unsigned long *_page_directory = (unsigned long *)read_cr3();
        // compute index of page directory (initial 10-bits)
        unsigned long page_directory_idx = (address >> 22);

        unsigned long *_page_table = nullptr;
    // compute index of page table (next 10-bits)
        unsigned long page_table_idx = ( (address & (0x03FF << 12) ) >> 12 );

    unsigned long *page_directory_entry;

    if ( (_page_directory[page_directory_idx] & 1 ) == 0 ) {  // the page-fault is at page directory level
      _page_table = (unsigned long *)(process_mem_pool->get_frames(1) * PAGE_SIZE);

      // get page directory entry
      page_directory_entry = (unsigned long *)(0xFFFFF << 12);
      page_directory_entry[page_directory_idx] = (unsigned long)(_page_table)| 0b11;

      // mark page table entry as invalid and set user level bit
            for(unsigned int i=0; i < 1024; ++i)
                _page_table[i] = 0b100;
    }
    // address the page-fault at page level
    // fetch page table entry
    page_directory_entry = (unsigned long *)(process_mem_pool->get_frames(1) * PAGE_SIZE);
    unsigned long *page_entry = (unsigned long *)((0x3FF << 22) | (page_directory_idx << 12));
    // mark page table entry as valid
    page_entry[page_table_idx] = ((unsigned long)(page_directory_entry) | 0b11);
  }

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

Listing 7: C++ code snippet

PageTable::register_pool() : Create a linked list of virtual memory pools in the page table.

```
void PageTable::register_pool(VMPool * _vm_pool) {
  if(PageTable::vm_pool_list) { // append _vm_pool to vm_pool_list
        VMPool *vmpool = PageTable::vm_pool_list;
        while(vmpool->next) {
            vmpool = vmpool->next;
        }
    vmpool->next = _vm_pool;
  } else {  // initialize vm_pool_list
    PageTable::vm_pool_list = _vm_pool;
  }
    Console::puts("registered VM pool\n");
}
```

Listing 8: C++ code snippet

PageTable::free_page() : To free the page, find the page directory index and page table index using the page number. Using the page table's entry at the page table index, calculate the frame number. At last, we release the frame and mark the page table entry as invalid along with reloading the page table to flush the TLB.

```cpp
void PageTable::free_page(unsigned long _page_no) {
  // compute index of page directory (initial 10-bits)
  unsigned long page_directory_idx = ( _page_no & 0xFFC00000) >> 22;
  // address of the page table entry
  unsigned long *page_table = (unsigned long *)((0x000003FF << 22) | (page_directory_idx << 12));

  // compute index of page table (next 10-bits)
  unsigned long page_table_idx = (_page_no & 0x003FF000 ) >> 12;
  unsigned long num_frame = (page_table[page_table_idx] & 0xFFFFF000) / PAGE_SIZE;

  // free the frames and set the page table entry as invalid
  process_mem_pool->release_frames(num_frame);
  page_table[page_table_idx] = page_table[page_table_idx] | 0b10;

  // reload the page table
  load();
    Console::puts("freed page\n");
}
```

Listing 9: C++ code snippet

vm_pool.H

```cpp
private:
   /* -- DEFINE YOUR VIRTUAL MEMORY POOL DATA STRUCTURE(s) HERE. */
   unsigned long              base_address;              // base address of virtual memory pool
   unsigned long              size;                      // size in bytes
   ContFramePool            * frame_pool;                // pointer to the frame pool
   PageTable                * page_table;                // pointer to the page table
   unsigned long              allocated_regions;      // allocated virtual memory regions
   unsigned long              free_memory;          // free memory regions
   struct RegionInfo {                                   // virtual memory region info
      unsigned long base_addr;                           // start address of the region
      unsigned long size;                                // size of the region
   };
   struct RegionInfo        * vmpool_regions;            // pointer to the virtual memory region list

public:
   VMPool                   * next;                 // points to the next virtual memory pool in the linked list
   ...
```

Listing 10: C++ code snippet

vm_pool.C

VMPool::VMPool() : This is the constructor for the VMPool class and configures the virtual memory pool. It registers the pool with the page table and initializes all the class variables that are necessary for the function of VMPool. Structure "RegionInfo" stores the start address and size of the VM pool region.

```
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;
    next = nullptr;

    // register the pool with the page table
    page_table->register_pool(this);

    // save the start address and page size
    vmpool_regions = (RegionInfo *)base_address;
    vmpool_regions[0] = {
        base_address,
        PageTable::PAGE_SIZE
    };

    allocated_regions = 1;

    // update free virtual memory available
    free_memory -= PageTable::PAGE_SIZE;

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

Listing 11: C++ code snippet

VMPool::allocate() : Allocates the size passed as argument. If the requested memory is not available, we throw an assert error. The function calculates the pages to be allocated and stores the start address and size in the RegionInfo data structure. Lastly, we update the free memory and number of allocated regions.

```
unsigned long VMPool::allocate(unsigned long _size) {
    // check if the required memory is available
    assert(_size <= free_memory)

    // number of pages
    unsigned long pages_count = ( _size / PageTable::PAGE_SIZE ) + ( (_size % PageTable::PAGE_SIZE) > 0 ? 1 : 0 );

    vmpool_regions[allocated_regions] = {
        vmpool_regions[allocated_regions-1].base_addr + vmpool_regions[allocated_regions-1].size, // start address of region
        pages_count * PageTable::PAGE_SIZE                              // size of the region
    };

    // update free memory
    free_memory -= pages_count * PageTable::PAGE_SIZE;
    // update number of allocated regions
    ++allocated_regions;

    Console::puts("Allocated region of memory.\n");

    return vmpool_regions[allocated_regions-1].base_addr;
}
```

Listing 12: C++ code snippet

VMPool::release() : Releases an allocated virtual memory region. Using the address from the argument, find out the region to be freed. Then, free all the pages in that region and overwrite that region with the next regions to delete the info it holds. Finally, we update the free memory and number of allocated regions.

```cpp
void VMPool::release(unsigned long _start_address) {
  // find which region does the _start_address belongs to
  int idx_region = -1;
  for(int i=1; i<allocated_regions; ++i)
    if(vmpool_regions[i].base_addr  == _start_address)
      idx_region = i;

  // number of pages to be released
  unsigned long page_count = vmpool_regions[idx_region].size / PageTable::PAGE_SIZE;
    for(int i=0; i<page_count; ++i) {
    page_table->free_page(_start_address);
    _start_address = _start_address + PageTable::PAGE_SIZE;
    }

  // overwrite virtual memory region to delete
  for(int i=idx_region; i<allocated_regions; ++i)
    vmpool_regions[i] = vmpool_regions[i+1];

  // update free memory
  free_memory += vmpool_regions[idx_region].size;

  // update the number of allocated regions
  --allocated_regions;

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

Listing 13: C++ code snippet

VMPool::is_legitimate() : Checks if the address lies between the currently allocated region i.e. in the range [base_address, base_address+size].

```cpp
bool VMPool::is_legitimate(unsigned long _address) {
    Console::puts("Checked whether address is part of an allocated region.\n");

    return _address >= base_address && _address <= base_address + size;
}
```

Listing 14: C++ code snippet

Testing

To test the implementation, I used the tests provided in kernel.C. Apart from that, I made a few changes in kernel.C for additional testing.

TEST 1

Default test provided in kernel.C. Fault Address = 4MB.
Tests the page table by generating page table memory references. The page-faults are handled and the test passed.

```cpp
#define FAULT_ADDR (4 MB)
#define NACCESS ((1 MB) / 4)
```

Listing 15: C++ code snippet

TEST 2

Fault Address = 8MB.
Similar to the previous test, the page table references are generated and all the page-faults are handled in the code and the test passed.

```
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
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
```

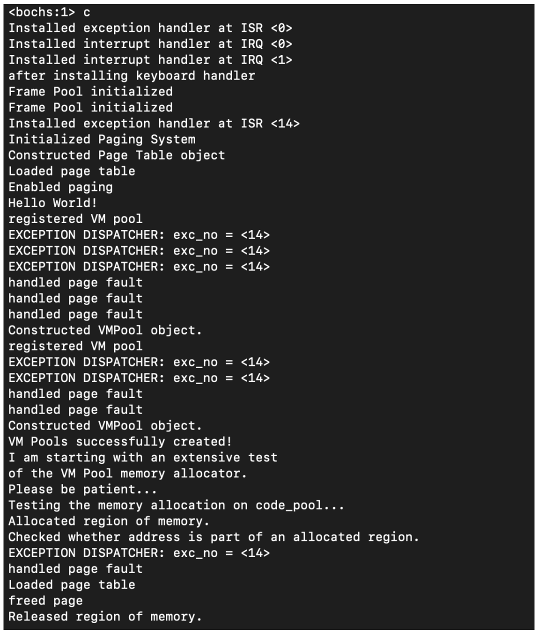
Figure 1: Test 1

```cpp
#define FAULT_ADDR (8 MB)
#define NACCESS ((1 MB) / 4)
```

Listing 16: C++ code snippet

```
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
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
```

Figure 2: Test 2



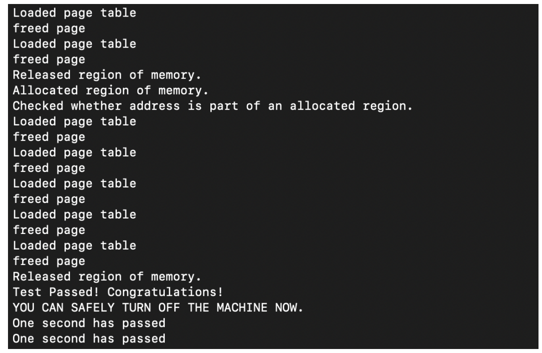


Figure 3: Test 3

TEST 3

Default test provided in kernel.C. I commented out #define _TEST_PAGE_TABLE_ so vmpools can be tested. The test generates virtual memory references.

```
// #define _TEST_PAGE_TABLE_
```

Listing 17: C++ code snippet

TEST 4

Modified the vmpool test. Tested allocation and de-allocation of multiple heap pools starting from the same memory location in sequential order.

```
    /* WE TEST JUST THE VM POOLS */

    /* -- CREATE THE VM POOLS. */

    /* ---- We define the code pool to be a 256MB segment starting at virtual address
    512MB -- */
    VMPool code_pool(512 MB, 256 MB, &process_mem_pool, &pt1);

    /* ---- We define a 256MB heap that starts at 1GB in virtual memory. -- */
    VMPool heap_pool_1(1 GB, 256 MB, &process_mem_pool, &pt1);
    VMPool heap_pool_2(1 GB, 256 MB, &process_mem_pool, &pt1);

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

    Console::puts("VM Pools successfully created!\n");

    /* -- GENERATE MEMORY REFERENCES TO THE VM POOLS */

    Console::puts("I am starting with an extensive test\n");
    Console::puts("of the VM Pool memory allocator.\n");
    Console::puts("Please be patient...\n");
    Console::puts("Testing the memory allocation on code_pool...\n");
    GenerateVMPoolMemoryReferences(&code_pool, 50, 100);
    Console::puts("Testing the memory allocation on heap_pool...\n");
    GenerateVMPoolMemoryReferences(&heap_pool_1, 50, 100);
    GenerateVMPoolMemoryReferences(&heap_pool_2, 50, 100);
```

Listing 18: C++ code snippet

```
<bochs:1> c
Installed exception handler at ISR <0>
Installed interrupt handler at IRQ <0>
Installed interrupt handler at IRQ <1>
after installing keyboard handler
Frame Pool initialized
Frame Pool initialized
Installed exception handler at ISR <14>
Initialized Paging System
Constructed Page Table object
Loaded page table
Enabled paging
Hello World!
registered VM pool
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
handled page fault
handled page fault
Constructed VMPool object.
registered VM pool
EXCEPTION DISPATCHER: exc_no = <14>
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
handled page fault
Constructed VMPool object.
registered VM pool
Constructed VMPool object.
VM Pools successfully created!
I am starting with an extensive test
of the VM Pool memory allocator.
Please be patient...
Testing the memory allocation on code_pool...
Allocated region of memory.
Checked whether address is part of an allocated region.
EXCEPTION DISPATCHER: exc_no = <14>
handled page fault
Loaded page table
freed page
Released region of memory.
Allocated region of memory.
Checked whether address is part of an allocated region.
Loaded page table
freed page
Released region of memory.
Allocated region of memory.
Checked whether address is part of an allocated region.
Loaded page table
freed page
```

```
freed page
Loaded page table
freed page
Released region of memory.
Allocated region of memory.
Checked whether address is part of an allocated region.
Loaded page table
freed page
Loaded page table
freed page
Loaded page table
freed page
Loaded page table
freed page
Loaded page table
freed page
Released region of memory.
Allocated region of memory.
Checked whether address is part of an allocated region.
Loaded page table
freed page
Loaded page table
freed page
Loaded page table
freed page
Loaded page table
freed page
Loaded page table
freed page
Released region of memory.
Allocated region of memory.
Checked whether address is part of an allocated region.
Loaded page table
freed page
Loaded page table
freed page
Loaded page table
freed page
Loaded page table
freed page
Loaded page table
freed page
Released region of memory.
Test Passed! Congratulations!
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
One second has passed
```

Figure 4: Test 4