

# MP4: Virtual memory Management and Memory Allocation

Vinay Balamurali  
UIN: 936002032  
CSCE611: Operating System

## Assigned Tasks

**Main:** Completed.

## System Design

This machine problem aims to extend the Page Manager to support large address spaces and implement a virtual memory allocator. The problem statement is split into three parts as follows:

### Part 1: Support for Large Address Spaces

The size of the directly mapped memory is small and therefore, cannot handle large address spaces. In the previous MP, the Page directory and Page Table Pages resided in the directly mapped memory so the logical address was equivalent to the physical address. But now, these elements have been moved to the freely mapped memory area, that is, they are a part of the process memory pool. The Page directory is contained within a single page and is being retained as part of the kernel memory pool (direct-mapped).

In this MP, we use a trick called 'Recursive Page Table lookup' to map the logical addresses to their corresponding physical addresses. The crux of this trick is to assign the last entry of the Page Directory (index=1023) to point to the Page Directory itself. The following two situations could arise:

### Accessing a Page Directory Entry (PDE)

The CPU will issue logical addresses of the form

| 1023: 10 | 1023: 10 | offset : 12 |

- The MMU will use the first 10 bits to index into the Page Directory to look up the PDE. But PDE 1023 points to the directory itself but the MMU thinks that this entry points to a Page Table Page.
- The MMU uses the second 10 bits to look up the 1023 in the supposed Page Table Page and ends up pointing back to itself.
- Now it uses the offset to index into the physical frame.

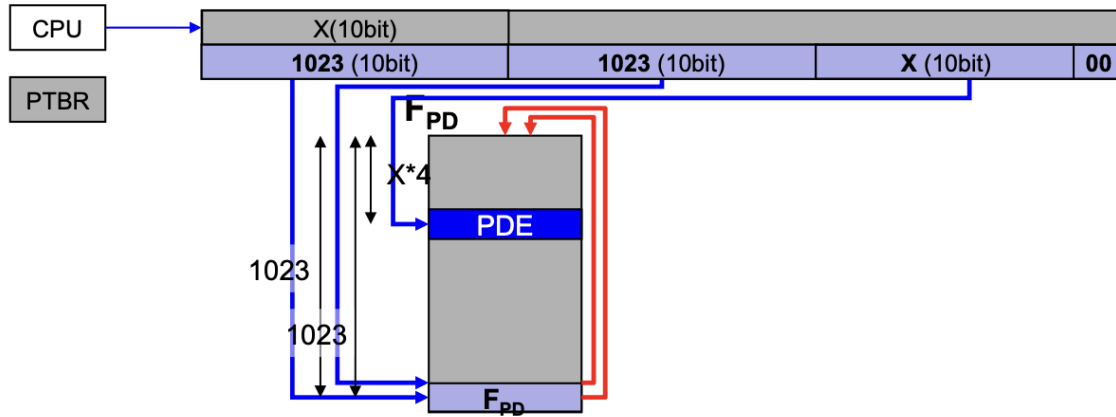


Figure 1: PDE access mechanism (credit: CSCSE 611 lectures)

## Accessing a Page Table Entry (PTE)

The CPU will issue logical addresses of the form

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

- The MMU will use the first 10 bits to index into the Page Directory to look up the PDE but this entry points back to itself.
- The MMU uses the second 10 bits to look up the Xth entry in the supposed Page Table page but this is the Xth entry of the PDE.
- The offset is used to index into the supposed physical frame which is in actuality, the page table page associated with the Xth PDE.
- The remaining 12 bits  $\rightarrow Y:00$  are used to enter into the Yth entry of the Page Table Page.

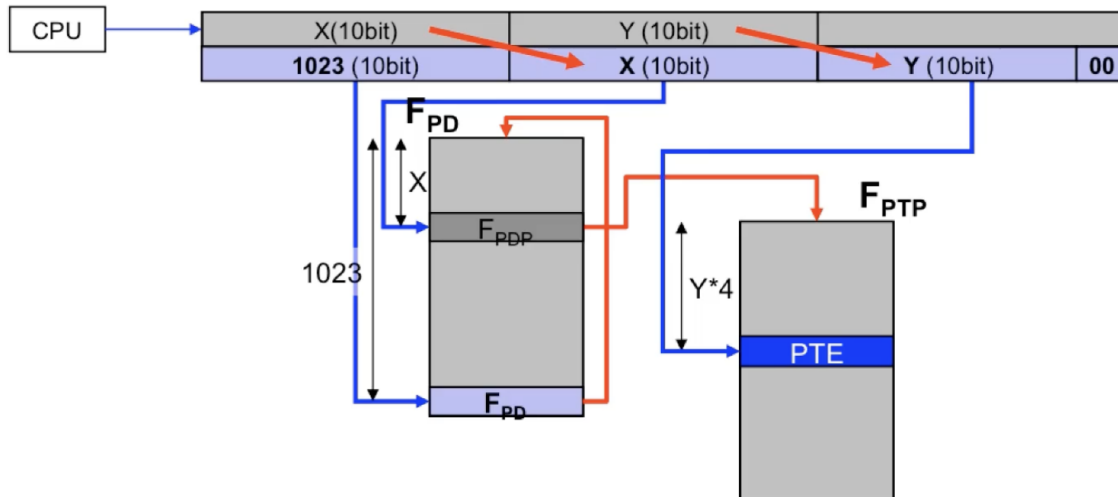


Figure 2: PTE access mechanism (credit: CSCSE 611 lectures)

## Part2: Modify class *PageTable* to handle Virtual Memory

The page table must know about all the virtual memory pools. This is because the page fault handler must be able to differentiate between legitimate accesses to memory that just so happened to be paged out or whether the application is trying to access memory that has not been allocated yet. This is implemented as follows:

- Each Virtual memory pool is registered with its *PageTable* object.
- The Virtual Memory pools are maintained as a linked list.

## Part 3: Virtual Memory Allocator

A simple virtual memory allocator has been implemented that can allocate virtual memory in pages only. There can be multiple virtual memory pools within the address space and each pool can have multiple regions. This would be a lazy allocator that only allocates physical memory through the Page Fault handler. Initially, it only remembers the base address and the length in bytes of the region within the pool. The memory can be de-allocated as well by first verifying which region a specific address belongs to and then removing the reference to that region.

## Code Description

### 0.1 Files Modified

- page\_table.H
- page\_table.C
- vm\_pool.H
- vm\_pool.C
- cont\_frame\_pool.H
- makefile

**page\_table.H** : The header file introduces a new data structure *static VMPool \*poolHead* which is a pointer to the head of the linked list containing the list of Virtual Memory Pools.

```
class PageTable {
private:
    /* THESE MEMBERS ARE COMMON TO ENTIRE PAGING SUBSYSTEM */
    static PageTable * current_page_table; /* pointer to currently loaded page table object */
    static unsigned int paging_enabled; /* is paging turned on (i.e. are addresses logical)? */
    static ContFramePool * kernel_mem_pool; /* Frame pool for the kernel memory */
    static ContFramePool * process_mem_pool; /* Frame pool for the process memory */
    static unsigned long shared_size; /* size of shared address space */

    static VMPool *poolHead; /* pointer to head of the Linked List to access Virtual Memory pools*/

    /* DATA FOR CURRENT PAGE TABLE */
    unsigned long * page_directory; /* where is page directory located? */

public:
    static const unsigned int PAGE_SIZE = Machine::PAGE_SIZE;
```

Figure 3: class *PageTable* header file

**page\_table.C: PageTable::init\_paging()** : This method is primarily responsible for setting the static members for this class. The static members are shared across instances of this class. This method is typically called before an instance of this class is declared. This method is largely unchanged from the previous MP.

```

// Method to initialize paging properties.
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");
} // PageTable::init_paging

```

Figure 4: Init paging method

**page\_table.C: PageTable::PageTable()** : The constructor is responsible for mapping the first 4 MB of memory. The PDE is allocated in the kernel frame pool but the Page Table Page for the first 4MB is gotten from the Process Memory pool. Additionally, the last entry in the PDE is mapped back to the Page Directory itself to enable the recursive lookup mechanism. Aside from this, the constructor remains largely unchanged from the previous MP.

```

// Constructor.
PageTable::PageTable()
{
    // 1 Page Table Directory for the entire process.
    unsigned long pdeFrame = kernel_mem_pool->get_frames(1);
    this->page_directory = (unsigned long *) (pdeFrame * PAGE_SIZE);

    // 1 Page table page for mapping to the first entry in the PDE.
    // This is gotten from the Process Memory pool.
    unsigned long ptpFrame = process_mem_pool->get_frames(1);
    unsigned long * page_table_page;
    page_table_page = (unsigned long *) (ptpFrame * PAGE_SIZE);

    unsigned long addressRange = 0;
    // Set attributes of the page table page entries to 011 (last 3 bits)
    // 0 --> kernel mode
    // 1 --> read/write
    // 1 --> protection fault
    // Each page_table_page entry references 4096 bytes of memory
    for (unsigned int i = 0; i < ENTRIES_PER_PAGE; i++)
    {
        page_table_page[i] = addressRange | 0x3;
        addressRange += PAGE_SIZE;
    }

    // Set the first entry of the PDE to be the page_table_page.
    page_directory[0] = (unsigned long) page_table_page;
    // Map the first 4MB of memory.
    page_directory[0] = page_directory[0] | 0x3;

    // Set remaining entries to be not present.
    // 0 --> kernel mode
    // 1 --> read/write
    // 0 --> not present
    for (unsigned int i = 1; i < (ENTRIES_PER_PAGE - 1); i++)
    {
        page_directory[i] = 0x0 | 0x2;
    }
    // Point last entry back to itself.
    page_directory[1023] = (unsigned long) page_directory | 0x3;

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

```

Figure 5: *class PageTable* Constructor

**page\_table.C: PageTable::load()** : The appropriate page table object is pointed to in this method. It can be used to flush the TLB since re-loading the Page Directory forces the CPU to flush the TLB.

```

// Method to set 'this' object as the current page table.
void PageTable::load()
{
    current_page_table = this;

    // Write the current page directory to the CR3 register.
    write_cr3((unsigned long) current_page_table->page_directory);
    Console::puts("Loaded page table\n");
} // PageTable::load

```

Figure 6: Load method

**page\_table.C: PageTable::enable\_paging()** : Paging on the x86 is enabled within this method. This is done by setting the MSB of register CR0.

```
// Method to enable paging by writing a high bit into
// the cr0 register.
void PageTable::enable_paging()
{
    // Enable paging on x86 by setting the MSB of
    // CR0 to 'high'.
    write_cr0(read_cr0() | 0x80000000);
    paging_enabled = 1;

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

Figure 7: Enable paging method

**page\_table.C: PageTable::handle\_fault()** : This method is the most essential part of the paging mechanism. it slightly varies from MP3. The algorithm is described as follows:

- Fetch the faulting address from register CR2.
- Fetch the current page directory from register CR3.
- Check the *err\_code* pushed on to the stack. If the LSB is set, then a protection violation occurred and it can't be handled at present. This is out of scope. If the LSB is 0, then the page is not present in memory, and further processing is required.
- Iterate through the list of Virtual Memory pools and verify if the memory access is legitimate, else crash the kernel.
- Use bit masks to parse the *Page Directory Index* and the *Page Table Page Index*.
- If bit 0 (LSB) is set when indexed to the *Current Page Directory* using the *Page Directory Index*, then the PDE reference is valid but a corresponding Page Table entry is not present. This must be done by first allocating a frame from the *process memory pool* and setting the table entry.
- If the bit 0 is not set, then the PDE entry is invalid. First, a Page Table Page must be allocated in the process frame pool. Post that a mapping must be created to the new PDE entry. We must also allocate a frame from the *process memory pool* and map it to the new page table page to ensure that a further fault does not occur.
- Return from page fault handler and the address is re-issued by the CPU.

```

// Method to handle the page fault exception.
void PageTable::handle_fault(REGS * _r)
{
    Console::puts("Page fault exception triggered!\n");
    // Fetch inaccessible address from register CR2.
    unsigned long faultAddress = read_cr2();

    // Load current page directory of the process from CR3.
    unsigned long *curr_page_dir = (unsigned long*)read_cr3();
    if (curr_page_dir == nullptr)
    {
        Console::puts("Failed to fetch Page Directory!\n");
        Console::puts("Exiting\n");
        assert(false);
    }

    // Only handle exception if page fault was triggered by
    // page not being present.
    unsigned long errorCode = _r->err_code;
    if ((errorCode & 0x1) == 1)
    {
        Console::puts("Handle Fault: Page is already present! Protection violation!\n");
        Console::puts("Exiting!\n");
        assert(false);
    }

    VMpool *node = PageTable::poolHead;

    bool isLegitimate = false;
    while (node != nullptr)
    {
        if (node->is_legitimate(faultAddress))
        {
            isLegitimate = true;
            Console::puts("Legitimate address. Breaking\n");
            break;
        }

        node = node->next;
    }
}

```

Figure 8: Page exception handler

**page\_table.C: PageTable::register\_pool()** : Add the virtual memory pool to the linked list by traversing through it. We follow insertion at the end.

```

// Method to register a new virtual memory pool
// to the linked list of pools.
void PageTable::register_pool(VMPool * _vm_pool)
{
    if (PageTable::poolHead == nullptr)
    {
        // first pool that is going to be registered.
        PageTable::poolHead = _vm_pool;
    }
    else
    {
        // Traverse to the end of linked list and insert.
        VMPool *currNode = PageTable::poolHead;

        while (currNode->next != nullptr)
        {
            currNode = currNode->next;
        }

        currNode->next = _vm_pool;
    }

    Console::puts("registered VM pool\n");
} // PageTable::register_pool

```

Figure 9: Register Pool method

**page\_table.C: PageTable::free\_page()** : This method constructs the logical address by accessing the Page Directory Index and Page Table Index to get the corresponding frame no. Post that, the process memory pool method: *release\_frames* is called with the frame no as the argument to release the frames. Finally, the TLB is flushed to ensure stale entries are not present.



```

// Method to free frame associated with a page, given the page no.s
void PageTable::free_page(unsigned long _page_no)
{
    // Fetch Page Directory Index.
    unsigned long pdeIndex = ((_page_no >> 22) & 0x3FF);

    // Fetch Page Table Page Index.
    unsigned long ptpIndex = ((_page_no >> 12) & 0x3FF);

    // Page Table Entry address = 1023 | PDE | Offset
    unsigned long *pageTablePage = (unsigned long *)((0x3FF << 22) | (pdeIndex << 12));

    // Fetch frame no and release it.
    unsigned long frame = (pageTablePage[ptpIndex] & 0xFFFFF000) / PAGE_SIZE;
    process_mem_pool->release_frames(frame);
    // Mark the entry as invalid.
    // 0 --> kernel mode
    // 1 --> read/write
    // 0 --> page not present.
    pageTablePage[ptpIndex] = pageTablePage[ptpIndex] | 0x2;

    // Flush the TLB.
    load();

    Console::puts("freed page\n");
} // PageTable::free_page

```

Figure 10: Free Page method

**vm\_pool.H** : The following private members were introduced.

- *unsigned long baseAddress* : Starting logical address of the virtual memory pool.
- *unsigned long size* : size in bytes of the pool.
- *ContFramePool\* framePool* : The physical frame pool where memory has been allocated.
- *PageTable \*pageTable* : Pointer to the Page Table object.
- *unsigned long totalRegions* : Total number of virtual memory regions within the virtual memory pool
- *unsigned long availableMemory* : Remaining memory of this virtual memory pool in bytes.
- *VMRegionInfo \*allRegions* : A pointer to all the virtual memory regions addressed as a struct.

The following public members were introduced.

- *VMPool \*next* : A pointer to the next Virtual Memory pool element in the linked list.

A new structure to map each virtual memory region was introduced, called ***vmRegionInfo***

- *unsigned long baseAddress* : Starting logical address of the virtual memory region.
- *unsigned long length* : Size in bytes of the virtual memory region.

```

/* Forward declaration of class PageTable */
/* We need this to break a circular include sequence. */
class PageTable;

struct vmRegionInfo
{
    unsigned long baseAddress;
    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 baseAddress;
    unsigned long size;
    ContFramePool *framePool;
    PageTable *pageTable;
    unsigned long totalRegions;
    unsigned long availableMemory;

    struct vmRegionInfo *allRegions;
public:
    VMpool *next; /* Pointer to next node */

```

Figure 11: Header File

**vm\_pool.C: VMpool::VMpool()** : The constructor initializes all data structures. The first region of virtual memory is mapped which contains the starting address of the entire pool. This occupies one page.

```

// Constructor.
VMPool::VMPool(unsigned long _base_address,
               unsigned long _size,
               ContFramePool *_frame_pool,
               PageTable *_page_table)
{
    baseAddress = _base_address;
    size = _size;
    availableMemory = size;
    framePool = _frame_pool;
    pageTable = _page_table;

    next = nullptr;
    totalRegions = 0;

    // Register the pool with PageTable object.
    pageTable->register_pool(this);

    // Map the first entry with the baseAddress of this pool.
    vmRegionInfo *region = (vmRegionInfo *)baseAddress;
    region[0].baseAddress = baseAddress;
    region[0].length = PageTable::PAGE_SIZE;
    allRegions = region;

    totalRegions = totalRegions + 1;

    availableMemory = availableMemory - PageTable::PAGE_SIZE;

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

```

Figure 12: *class VMPool* constructor

**vm\_pool.C: VMPool::allocatel()** : This method allocates memory of *\_size* in terms of pages. It rounds up to a page larger than what is required. We don't take into account internal fragmentation using this logic.

```

// Method to allocate a virtual memory region given
// _size in bytes.
unsigned long VMPool::allocate(unsigned long _size)
{
    if (_size > availableMemory)
    {
        Console::puts("Requested Memory greater than available size!\n");
        return 0;
    }

    unsigned long pagesRequired = 0;

    // Divide to get total pages.
    pagesRequired = (_size / PageTable::PAGE_SIZE);

    // We also need to account for the last page. We allocate an entire page
    // and don't care about internal fragmentation.
    unsigned long extraPages = (_size % PageTable::PAGE_SIZE) > 0 ? 1 : 0;
    pagesRequired = pagesRequired + extraPages;

    unsigned long requiredMemory = pagesRequired * PageTable::PAGE_SIZE;
    unsigned long newBaseAddress = allRegions[totalRegions - 1].baseAddress + allRegions[totalRegions - 1].length;
    // totalRegions indexes the region that is being allocated.
    allRegions[totalRegions].baseAddress = newBaseAddress;
    allRegions[totalRegions].length = requiredMemory;

    availableMemory = availableMemory - requiredMemory;

    totalRegions = totalRegions + 1;

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

    return newBaseAddress;
} // VMPool::allocate

```

Figure 13: *class VMPool* allocate

**vm\_pool.C: VMPool::release()** : This method releases the region of virtual memory starting from the address passed into the argument. But first, the region must be found by iterating through the list of regions. Once the match is found, pages are freed iteratively. At the end, the current region.

```

// Method to release a region of virtual memory,
// starting from address _start_address.
void VMPool::release(unsigned long _start_address)
{
    int indexRegion = -1;

    for (int i = 1 ; i < totalRegions; i++)
    {
        if (allRegions[i].baseAddress == _start_address)
        {
            // Found a match for the region.
            indexRegion = i;
        }
    }

    // Crash the kernel if a match is not found.
    if (indexRegion == -1)
    {
        Console::puts("Given start address not found in the current VM pool!\n");
        assert(false);
    }

    unsigned long pagesToRelease = (allRegions[indexRegion].length / PageTable::PAGE_SIZE);
    unsigned long tempAddress = _start_address;

    // Release frames iteratively.
    while (pagesToRelease > 0)
    {
        pageTable->free_page(tempAddress);

        pagesToRelease = pagesToRelease - 1;

        tempAddress = tempAddress + PageTable::PAGE_SIZE;
    }

    availableMemory = availableMemory + allRegions[indexRegion].length;

    // Shift all regions greater than the one to be freed
    // to delete the current region.
    for (int i = indexRegion; i < totalRegions; i++)
    {

```

Figure 14: *class VMPool* release

**vm\_pool.C: VMPool::is\_legitimate()** : The address that is being handled at the Page Fault handler is verified to ensure that it is part of the Virtual Memory that has been allocated. This is to confirm that memory accesses do not cross over into unallocated regions.

```

// Method to verify if an address belonging to virtual memory pool
// is valid by performing a bounds check.
bool VMPool::is_legitimate(unsigned long _address)
{
    Console::puts("Checked whether address is part of an allocated region.\n");

    if ((_address < baseAddress) || (_address > (baseAddress + size)))
    {
        return false;
    }

    return true;
} // VMPool::is_legitimate

```

Figure 15: *class VMPool* release

**cont\_frame\_pool.H** : This file has been taken from the official solutions as is.

```

/*-----*/
/* ContFramePool */
/*-----*/

class ContFramePool {
private:
    /* -- DEFINE YOUR CONT FRAME POOL DATA STRUCTURE(s) HERE. */

    /* ---- Bag of created frame pools */

    static ContFramePool * head;
    static ContFramePool * tail;

    ContFramePool * next;
    ContFramePool * prev;

    static void add_to_bag(ContFramePool * _new_pool);
    static void remove_from_bag(ContFramePool * _pool);

    /* ---- Info about this frame pool */
    unsigned char * bitmap;          // We implement the simple frame pool with a bitmap
    unsigned long base_frame_no;     // Where does the frame pool start in phys mem?
    unsigned long nframes;           // Size of the frame pool
    unsigned long info_frame_no;     // First of the frames with meta data about pool.

    /* ---- State management */

    enum class FrameState {Free, Used, HoS, INVALID};

    FrameState get_state(unsigned long _frame_no);
    void set_state(unsigned long _frame_no, FrameState _state);

    /* ---- Release function of the pool */

```

Figure 16: cont.frame.pool.H

**makefile** : *cont.frame.pool.o* has been used from the solution. We need to ensure this particular file is not removed when we run *make clean*. For this we slightly, modify the rule for *make clean* to filter out the particular file. Additionally, the rule to build *cont.frame.pool.C* has been removed as well.

```

GCC_OPTIONS = -m32 -nostdlib -fno-builtin -nostartfiles -nodefaultlibs -fno-exceptions

all: kernel.bin

clean:
    rm -f $(filter-out cont_frame_pool.o, $(wildcard *.o *.bin))

run:
    qemu-system-x86_64 -kernel kernel.bin -serial stdio

debug:
    qemu-system-x86_64 -s -S -kernel kernel.bin

# ==== KERNEL ENTRY POINT ====

```

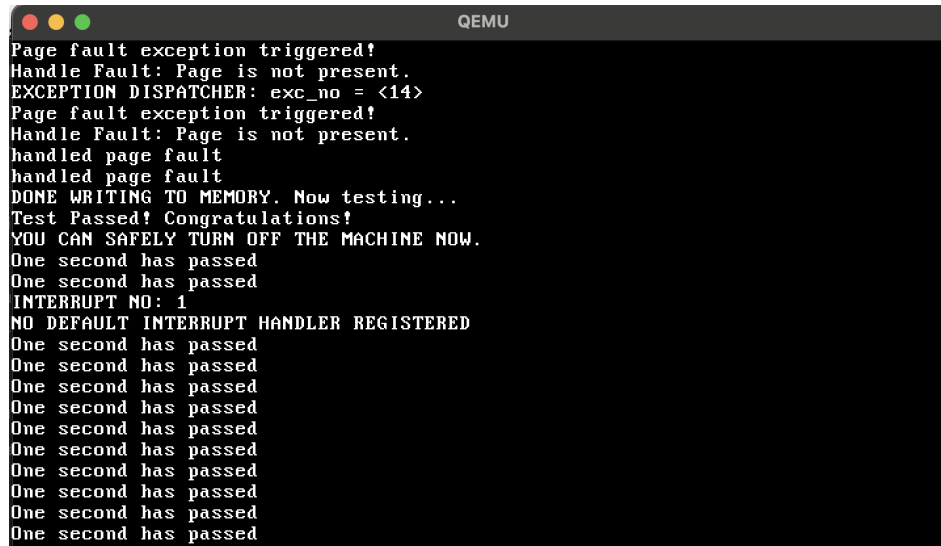
Figure 17: Makefile

## Testing

Testing is covered in two parts. One to test the Page Table and the other to test the Virtual Memory Allocation.

## Test Page table

Testing was performed using the functionality provided in *kernel.C* wherein the fault address was at 4MB and requesting 2 KB of memory.

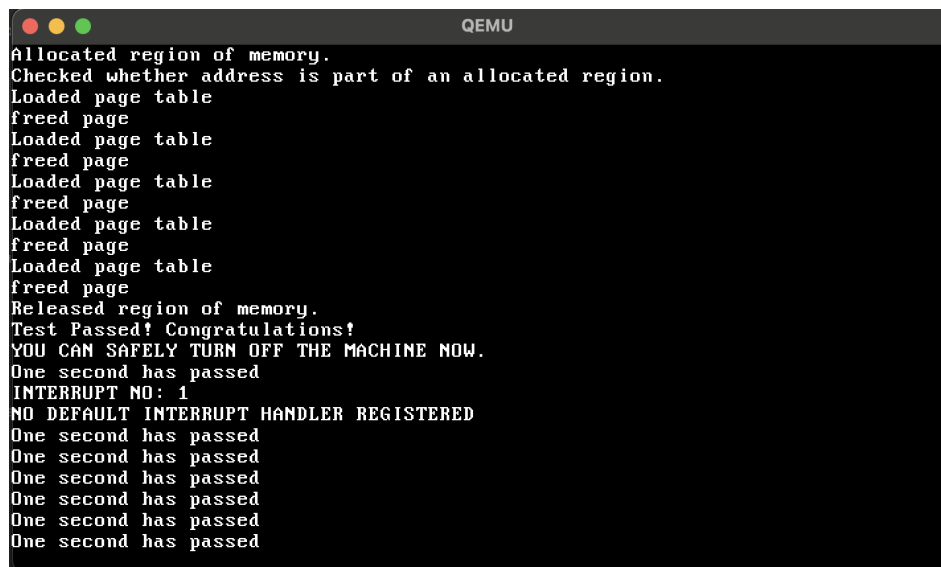
A screenshot of a QEMU terminal window with a black background and white text. The window title is 'QEMU'. The output shows a sequence of messages: 'Page fault exception triggered!', 'Handle Fault: Page is not present.', 'EXCEPTION DISPATCHER: exc\_no = <14>', 'Page fault exception triggered!', 'Handle Fault: Page is not present.', 'handled page fault', 'handled page fault', 'DONE WRITING TO MEMORY. Now testing...', 'Test Passed! Congratulations!', 'YOU CAN SAFELY TURN OFF THE MACHINE NOW.', followed by 'One second has passed' repeated five times, 'INTERRUPT NO: 1', 'NO DEFAULT INTERRUPT HANDLER REGISTERED', and then 'One second has passed' repeated five more times.

```
QEMU
Page fault exception triggered!
Handle Fault: Page is not present.
EXCEPTION DISPATCHER: exc_no = <14>
Page fault exception triggered!
Handle Fault: Page is not present.
handled page fault
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
INTERRUPT NO: 1
NO DEFAULT INTERRUPT HANDLER REGISTERED
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 18: QEMU output - Page Table Test

## Test Virtual Memory Pool Allocation

Two virtual memory pools are allocated namely: *code.pool* and *heap.pool*. Memory is randomly allocated and de-allocated in a loop to test the pools.

A screenshot of a QEMU terminal window with a black background and white text. The window title is 'QEMU'. The output shows: '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.', followed by 'One second has passed', 'INTERRUPT NO: 1', 'NO DEFAULT INTERRUPT HANDLER REGISTERED', and then 'One second has passed' repeated five more times.

```
QEMU
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
INTERRUPT NO: 1
NO DEFAULT INTERRUPT HANDLER REGISTERED
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 19: QEMU output - Virtual Memory Pool Test

Detailed logs can be found under *../outputs* as part of the submission.