

CSCE611 Machine Problem 4 Virtual Memory Management and Memory Allocation

Gehao Yu 629008717

Mar 14 2021

1 Introduction

We have to design the virtual memory management and memory allocation function based on the requirement of the machine problem 4. In this machine problem, we have to consider that the number and the size of the table pages could be very large, so that 1 on 1 mapping physical address may not be enough to support such a system. To solve this problem, we need to extend our page tables to virtual memory.

What's more, we also need to implement the allocate function to support the region management in the current virtual memory pool.

2 Modification in page_table.H

Now we need to support not only the page tables but also the virtual memory pools, so we added the corresponding definitions into page_table.H

```
// -- NEW IN MP4
static const unsigned int vm_pool_cnt;
static VMPool * vm_pool_list[];
```

3 Modification in page_table.C

1: modification in the PageTable()

To implement the "recursive page table lookup" scheme in the part 1, we should let the last entry in the page directory pointed to itself. Then create the empty virtual memory list to store the virtual memory pools and set them as NULL.

```

page_directory[1023] = (unsigned long) page_directory | 3;
//Let the last page directory entry pointed to itself.

```

```

unsigned int j;
//empty the vm pool lists
for (j=0; j<vm_pool_cnt; j++){
    vm_pool_list[j] = NULL;
}

```

2: modification in the handle_fault()

Since we already implemented the "recursive page table lookup" scheme, we no longer need to read the page directory from CR3, we could directly go to the last entry of page directory to find the page directory start address.

```

    unsigned long *cur_page_dir = (unsigned long*) 0xFFFFF000;

```

Unlike what we have done in MP3, once we get the page fault address from CR2, we should firstly check if this logical address legitimate or not. To implement this function, we need to see if the virtual pool list in current page table empty or not.

```

//Check if the logical address is legitimate
Console::puts("Checking the logical address\n");

VMPool** vm_list = current_page_table->vm_pool_list;

bool logical_valid = false;
for(int i=0; i<vm_pool_cnt; i++){
    if(vm_list[i] != NULL){
        logical_valid = true;
        Console::puts("Current list is valid\n");
    }
}

if(logical_valid == false){
    Console::puts("No valid address..\n");
}

```

If the page directory entry of the page fault address is "not present", we need to pick a free frame from memory pool and initialize the page table all empty; But if the page is "present", we only need to get the start address of page table. After we locate the page table start address, we only need to put the frame into it and handle the page fault.

```

// If the current page directory's offset corresponding page table is "non present"
if((cur_page_dir[dir_offset] & 1) != 1){
    Console::puts("page not present\n");
    //get a frame from free frame pool and initialization process
    cur_page_dir[dir_offset] = (unsigned long)((process_mem_pool->get_frames(1)<<12) | 3);
    //get the new page table start address
    page_table = (unsigned long*)((dir_offset <<12) | 0xFFC00000);
    //empty all the entries in the new page table.
    for(int i = 0; i< 1024; i++){
        page_table[i] = 0;
    }
}else{
    Console::puts("page present\n");
    //only need to get the atrt address
    page_table = (unsigned long*)((dir_offset <<12) | 0xFFC00000);
}
//put the frame into the page table.
page_table[table_offset] = (process_mem_pool->get_frames(1) << 12) | 3;
Console::puts("handled page fault\n");

```

3: modification in the register_pool()

All that we need to do is traverse all the virtual memory pools in the list to see if there is empty pool. If so, register the given virtual memory pool and all settled.

```

//
void PageTable::register_pool(VMPool * _vm_pool)
{
    //for each pool in the list
    for(int i=0; i<vm_pool_cnt; ++i){
        //If we find out an empty one
        if (vm_pool_list[i] == NULL){
            vm_pool_list[i] = _vm_pool;
            Console::puts("Successfully register Virtual memory pool.\n");
            return;
        }
    }
    Console::puts("Failed to register virtual memory pool.\n");
}

```

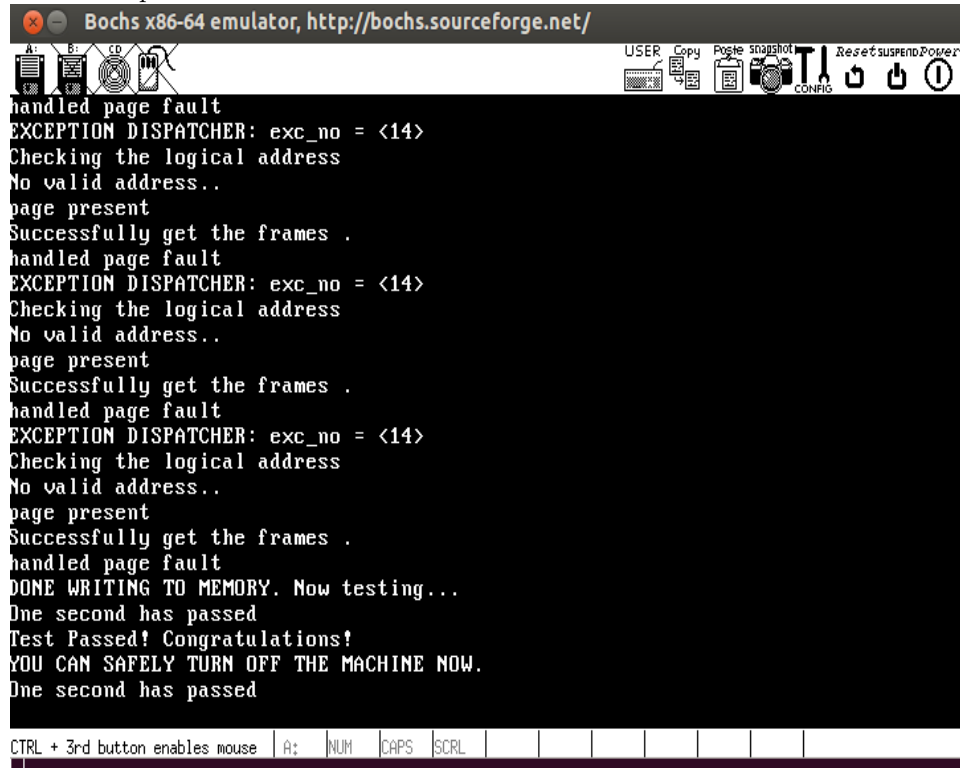
4: modification in the free_page()

Based on the given page number, we shall pick the page directory offset and the page table offset. Selected the entry in the page table and release the frames.

```
//
void PageTable::free_page(unsigned long _page_no)
{
    unsigned long dir_offset = _page_no >> 22;
    unsigned long table_offset = ((_page_no >> 12) & 0x3FF);
    unsigned long *page_table = (unsigned long *)((dir_offset<<12)|0xFFC00000);
    unsigned long frame_idx = page_table[table_offset];
    process_mem_pool->release_frames(frame_idx);
    Console::puts("Pages are freed\n");
}
```

4 Result of page_table test

The result performance of the code:



```
Bochs x86-64 emulator, http://bochs.sourceforge.net/
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
Checking the logical address
No valid address..
page present
Successfully get the frames .
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
Checking the logical address
No valid address..
page present
Successfully get the frames .
handled page fault
EXCEPTION DISPATCHER: exc_no = <14>
Checking the logical address
No valid address..
page present
Successfully get the frames .
handled page fault
DONE WRITING TO MEMORY. Now testing...
One second has passed
Test Passed! Congratulations!
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
One second has passed
```

5 Modification in vm_pool.H

New structure mem_region:

We need to define a new structure called mem_region, which included two variables base_addr and region_size. This structure describe the region size information of regions in a virtual memory pool.

```
struct mem_region{
    unsigned long base_addr;
    unsigned long region_size;
};
```

6 Modification in vm_pool.C

1: New variables :

We define 2 private variables in vm_pool.C. They are mem_region_list and last_mem_region. mem_region_list represented current virtual memory pool, while the last_mem_region represented the last used regions index, which is 0 at the beginning.

```
VMPool::VMPool(unsigned long _base_address,
               unsigned long _size,
               ContFramePool *_frame_pool,
               PageTable *_page_table) {
    //assert(false);
    base_address = _base_address;
    size = _size;
    frame_pool = _frame_pool;
    page_table = _page_table;

    last_mem_region = 0;
    //take free frame to store the region list
    mem_region_list = (mem_region *) (frame_pool->get_frames(1)*PageTable::PAGE_SIZE);
    page_table->register_pool(this);

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

2: modification in the allocate()

Firstly we have to sort the regions based on their base address to make the traverse more intuitively. Then, if we still have regions to allocate, we're finding out the start address of the regions.

```

//Bubble sort the memory regions sequence based on their base address
for (int i=0; i<last_mem_region; i++){
    for (int j=i+1; j<last_mem_region; j++){
        if (mem_region_list[j].base_addr < mem_region_list[i].base_addr){
            mem_region k;
            k = mem_region_list[i];
            mem_region_list[i] = mem_region_list[j];
            mem_region_list[j] = k;
        }
    }
}
}

```

If no region been allocated, we will assign the base address; otherwise, we have to traverse all the regions and to see if there exists holes(or empty regions) to allocate. If we could find one, we will allocate it and break; otherwise we will set the last region address as the start address.

```

// try to see if we could allocate the region
if (last_mem_region < (PageTable::PAGE_SIZE / sizeof(mem_region))){
    if (last_mem_region == 0){
        start_addr = base_address;
    }else{
        //To see if we can allocate the first gap space for space allocation
        unsigned long gap_region = 0;
        gap_region = mem_region_list[0].base_addr - base_address;
        if (_size <= gap_region){
            //If we could
            start_addr = base_address;
        }else{
            //If we couldn't
            bool flag = false;
            //If more than 2 regions already been allocated
            if (last_mem_region > 1){
                for (int i=0; i<last_mem_region-1; i++){
                    //updated the new gap region size
                    gap_region = mem_region_list[i+1].base_addr - (mem_region_list[i].base_addr+mem_region_list[i].region_size);
                    //If we could put it in & keep allocate untill the end
                    if (_size <= gap_region){
                        start_addr = mem_region_list[i].base_addr + mem_region_list[i].region_size;
                        flag = true;
                        break;
                    }
                }
            }
            //If we couldn't find a position, put it at the end
            if (flag == false){
                start_addr = mem_region_list[last_mem_region-1].base_addr + mem_region_list[last_mem_region-1].region_size;
            }
        }
    }
}
}
}

```

Once we got the start address, we have to make sure that this is a valid address(no overflow). If it is, we could return it.

```

//verify if allocated start address would lead to a space overflow
if ((start_addr + _size) <= base_address + size){
    mem_region_list[last_mem_region].base_addr = start_addr;
    mem_region_list[last_mem_region].region_size = _size;

    last_mem_region++;
    Console::puts("Successfully allocate space for memory.\n");
    return start_addr;

}else if ((start_addr + _size) > base_address + size){
    Console::puts("Unsuccessfully allocate space for memory because of virtual pool size overflow.\n");
    return 0;
}

}else{
    Console::puts("Unsuccessfully allocate space for memory because of virtual pool size overflow(no enough regions).\n");
    return 0;
}

```

3: modification in the release()

We need to traverse all the regions in the pool to find if there exist a region's start address is the same. Once we find it, we would free the corresponding page table and empty that region. If there is some regions located at the back of the released one, we have to move these regions one step ahead. After doing these, we reduce one region index and flush the TLB using read_CR3().

```

void VMpool::release(unsigned long _start_address) {
    //To use this function, we must make sure the provided address is valid

    unsigned long match_pos;
    //For each regions before the last memory region
    for (int i=0; i<last_mem_region; i++){
        //Once we find it
        if (mem_region_list[i].base_addr == _start_address){
            //record the position and then release the corresponding page
            match_pos = i;
            page_table->free_page(_start_address);
            //if no region left
            if (last_mem_region <= 1){
                mem_region_list[0].base_addr = 0;
                mem_region_list[0].region_size = 0;
            //if there are some regions left
            }else{
                //put the left regions into the new sequence
                for (int j=match_pos; j<last_mem_region-1; j++){
                    mem_region_list[j] = mem_region_list[j+1];
                }
            }
            break;
        }
    }

    last_mem_region--;

    //after released the page, flushed the TLB
    page_table->load();

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

```

4: modification in the is_legitimate()

This is simple. We traverse all the regions and see if there is a region included given address.


```

bool VMPool::is_legitimate(unsigned long _address) {
    // checking if the address is legitimate or not
    Console::puts("Checking given address is legitimate or not.\n");
    if (mem_region_list){
        //for each regions in the list
        for (int i=0; i<last_mem_region; i++){
            unsigned long head = mem_region_list[i].base_addr;
            unsigned long tail = mem_region_list[i].base_addr + mem_region_list[i].region_size;
            //we find a region that included the given address
            if((head <= _address)&&(_address < tail)){
                Console::puts("Valid address.\n");
                return true;
            }
        }
    }
    Console::puts("Invalid address.\n");
    return false;
}

```

7 Result of vm_pool test

The result performance of the code:

```

Bochs x86-64 emulator, http://bochs.sourceforge.net/
Loaded page table
Released region of memory.
Successfully allocate space for memory.
Checking given address is legitimate or not.
Valid address.
Pages are freed
Loaded page table
Released region of memory.
Successfully allocate space for memory.
Checking given address is legitimate or not.
Valid address.
Pages are freed
Loaded page table
Released region of memory.
Successfully allocate space for memory.
Checking given address is legitimate or not.
Valid address.
Pages are freed
Loaded page table
Released region of memory.
Test Passed! Congratulations!
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
One second has passed
One second has passed
CTRL + 3rd button enables mouse  A:  NUM  CAPS  SCRL

```