Machine Problem 4

Jicheng Lu, 525004048

1. Introduction

In Machine Problem 4, we extend and complete our memory manager. First, we extend the page table management to support very large numbers and sizes of address spaces. In order to achieve this, we move the page tables into “virtual” memory (i.e., process memory pool) and prepare the page table to support virtual memory. Then, we implement a simple virtual-memory allocator and hook it up to the “new” and “delete” operators of C++.

As a result, we have a very flexible simple memory management system, which is able to dynamically allocate memory.

2. Page Table Implementation

In this machine problem, we use Recursive Page Table Look-up to support for large address space. By doing this, we need to move the page directory and page table pages to the mapped memory (i.e., process memory pool).

First, we need to get a frame for page directory and page table from the ‘process memory pool’, respectively. We then map the first 4MB memory like what we did in machine problem 3. The difference is we need to change last entry in the page directory so that it can point to the page directory itself.

```
//
PageTable::PageTable()
{
    page_directory = (unsigned long *)(process_mem_pool->get_frames(1)*FRAME_SIZE);

    page_table = (unsigned long *)(process_mem_pool->get_frames(1)*FRAME_SIZE);

```

Figure 2.1. Get frames for page directory and page table from process pool to map the first 4MB.

```
    //set the last PDE to point to itself
    page_directory[shared_size/PAGE_SIZE-1] = (unsigned long)page_directory|3;
```

Figure 2.2. Change last entry in the page directory.

We also modify the page fault handler by changing the page directory address and page table address.

```
//
void PageTable::handle_fault(REGS * _r)
{
    // this address recursively points to the page directory
    // by referencing itself twice
    unsigned long *page_Dir = (unsigned long *)0xFFFFF000;
    unsigned long *page_Table;

    unsigned long address;
    unsigned long error = _r->err_code;
```

```
    unsigned long dir_index;
    unsigned long pt_index;

    dir_index = address >> 22;
    pt_index = (address >> 12) & 0x3FF;

    if ((page_Dir[dir_index] & 1) == 0){
        Console::puts("Page table is not present.\n");

        // get a frame from process mem pool
        page_Dir[dir_index] = (unsigned long)((process_mem_pool->get_frames(1)<<12)|3);

        // get the address of the new page table
        page_Table = (unsigned long *)(0xFFC00000|(dir_index<<12)); // it points to multiple of 4KB

        // initialize page table
        for (int i=0; i<ENTRIES_PER_PAGE; ++i){
            page_Table[i] = 0;
        }

        // load the frame into page table
        page_Table[pt_index] = (process_mem_pool->get_frames(1)<<12) | 3;

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

    }else{
        Console::puts("Page table is present.\n");

        // get the address of the present page table
        page_Table = (unsigned long *)(0xFFC00000|(dir_index<<12));

        // load the frame into page table
        page_Table[pt_index] = ((process_mem_pool->get_frames(1))<<12) | 3;

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

Figure 2.3. Page fault handler in ‘Page Table’.

After these modification, we test the page table memory references using the default kernel.C. The result is given as follows:

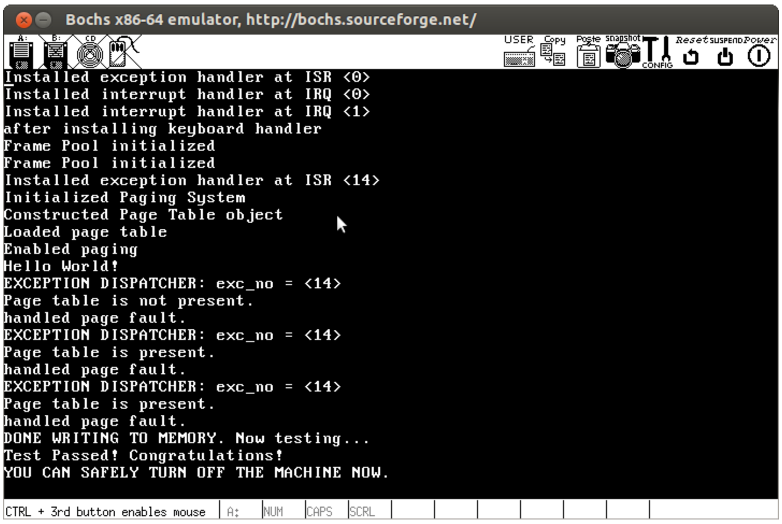

Figure 2.4. Page table memory reference test.

Next, we extend the page table manager to handle the registration of virtual memory pools. In order to do this, we first define a “vmpool_manager” array with a size of “vmpool_max_num”. Initially, we set each element in the “vmpool_manager” as NULL in the constructor. When we need to register a virtual memory pool, we simply put it into the NULL position in the “vmpool_manager”.

```
    for (int i=0; i<vmpool_max_num; ++i){
        vmpool_manager[i] = NULL;
    }
```

```
//
void PageTable::register_pool(VMPool * _vm_pool)
{
    int flag = -1;

    for(int i=0; i<vmpool_max_num; ++i){
        if (vmpool_manager[i] == NULL){
            flag = i;
            vmpool_manager[flag] = _vm_pool;
            Console::puts("register VM pool successfull.\n");
            break;
        }
    }

    if (flag < 0){
        Console::puts("register VM pool failed.\n");
    }



}
```

Figure 2.5. Initialization and register_pool in page table manager.

Then, we extend the page table manager to handle requests to free pages. In order to do this, we find the starting frame number and use “release_frames” function in the Contiguous Frame Pool.

```
//
void PageTable::free_page(unsigned long _page_no)
{
    unsigned long dir_index = _page_no >> 22;
    unsigned long pt_index = (_page_no >> 12) & 0x3FF;
    unsigned long *page_Table = (unsigned long *)(0xFFC00000|(dir_index<<12));
    unsigned long frame_num = page_Table[pt_index];
    process_mem_pool->release_frames(frame_num);

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

Figure 2.6. free_page in Page Table.

Moreover, we check if the logical address is legitimate during page faults by using the “is_legitimate” function in virtual memory pool. The detail of the “is_legitimate” function will be introduced in next section

```
    address = read_cr2();

    // check if the logical address is legitimate
    VMPool** vm_manager = current_page_table->vmpool_manager;

    int check_flag = -1;
    for (int i=0; i<vmpool_max_num; ++i){
        if (vm_manager[i] != NULL){
            if (vm_manager[i]->is_legitimate(address)){
                check_flag = i;
                Console::puts("Valid address...\n");
                break;
            }
        }
    }

    if (check_flag < 0){
        Console::puts("Invalid address...\n");
    }
    //
```

Figure 2.7. Check logical address during page faults.

3. Virtual Memory Pool Implementation

Here we implement a simple virtual memory pool manager. First, we define a structure, named “allocator_info”, which contains base_address and region_size. We also allocate a frame in memory to store the “allocate_list” in “vm_pool.H”, which contains the information of different regions. We also define a “region_num” to indicate the number of regions in the “allocate_list”, and a “region_max_num” to indicate the maximum number of regions that can be stored in “allocate_list”. We register this virtual memory pool after initialization.

```
struct allocator_info{
  unsigned long base_address;
  unsigned long region_size;
};
```

```
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;

    region_num = 0;

    // how many region management info can be stored in a frame at most
    region_max_num = PageTable::PAGE_SIZE / sizeof(allocator_info);

    // get a frame to store the allocate_list
    allocate_list = (allocator_info *)(frame_pool->get_frames(1)*PageTable::PAGE_SIZE);

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

```

Figure 3.1. VMPool initialization.

Then we define an allocator for virtual memory pool. We first sort the base address in each entry of the “allocate_list”. Then we get the starting address (i.e., “start_addr”) based on whether the newly-added region can be fit in the current configuration. After getting the starting address, we check the newly-added region is out of the vmpool.

```
//
unsigned long VMPool::allocate(unsigned long _size) {
    if (_size == 0){
        Console::puts("Unable to allocate because size is zero.\n");
        return 0;
    }

    unsigned long start_addr = 0;
    allocator_info temp;
    unsigned long hole = 0;

    for (int i=0; i<region_num; ++i){
        for (int j=i+1; j<region_num; ++j){
            if (allocate_list[j].base_address < allocate_list[i].base_address){
                temp = allocate_list[i];
                allocate_list[i] = allocate_list[j];
                allocate_list[j] = temp;
            }
        }
    }
```

Figure 3.2. Sorting “allocate_list”.

```
// get start_addr based on different conditions.
if (region_num == 0)
{
  start_addr = base_address;
}
else if (region_num == 1)
{
  hole = allocate_list[0].base_address - base_address;
  if (_size <= hole){
    start_addr = base_address;
  }else{
    start_addr = allocate_list[region_num-1].base_address + allocate_list[region_num-1].region_
  }
}
else
{
  hole = allocate_list[0].base_address - base_address;
  if (_size <= hole){
    start_addr = base_address;
  }else{
    for (int i=0; i<region_num-1; ++i){
      hole = allocate_list[i+1].base_address - (allocate_list[i].base_address+allocate_list[i].
      if (_size <= hole){
        start_addr = allocate_list[i].base_address + allocate_list[i].region_size;
      }
    }
  }
  if (start_addr == 0 ){
    start_addr = allocate_list[region_num-1].base_address + allocate_list[region_num-1].region_
  }
}
```

Figure 3.3. Getting starting address.

```
// check whether the newly-allocated region is out of the vmpool
unsigned long end_addr;
end_addr = base_address + size;

if ((start_addr + _size) <= end_addr)
{
  allocate_list[region_num].base_address = start_addr;
  allocate_list[region_num].region_size = _size;
  region_num++;
  return start_addr;
}
else if (((start_addr + _size) > end_addr) || (region_num++ > region_max_num))
{
  Console::puts("Unable to allocate because it is out of space.\n");
  return 0;
}
```

Figure 3.4. Check the newly-added region.

After allocating, we need to release. In order to do this, we identify the “_start_address” in the “allocate_list” in the first place. If there is a match, we release the region by calling “free_page” in Page Table. Then, we delete the corresponding entry and reconstruct the allocate_list. More importantly, we flush TLB by reloading CR3 after we release the pages.

```
//
void VMPool::release(unsigned long _start_address) {

    unsigned long index;
    for (int i=0; i<region_num; ++i){
      if (allocate_list[i].base_address = _start_address){
        index = i;
        page_table->free_page(_start_address);  //release the pages for current region
        break;
      }
    }

    // delete entry and reonstruct allocate_list
    if (region_num > 1){
      for (int j=0; j<region_num; ++j){
        if (j>index){
          allocate_list[j-1] = allocate_list[j];
        }
      }
    }else{
      allocate_list[0].base_address = 0;
      allocate_list[0].region_size = 0;
    }

    region_num--;

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


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

Figure 3.5. “release” function in virtual memory pool.

Finally, we give the details of “is_legitimate” function. We can check if the address is valid by looking into each entry in the “allocate_list”.

```
//
bool VMPool::is_legitimate(unsigned long _address) {
    /*
    for (int k=0; k<region_num; ++k){
      Console::puti(allocate_list[k].base_address); Console::puts("\n");
    }

    Console::puti(_address);Console::puts("\n");
    */

    // check whether the address is legitimate
    if (allocate_list){
      for (int i=0; i<region_num; ++i){
        if((allocate_list[i].base_address <= _address)&&(_address < allocate_list[i].base_address
          Console::puts("The address is valid.\n");
          return true;
        }
      }
    }
    Console::puts("The address is invalid.\n");
    return false;

```

Figure 3.6. “is_legitimate” function in virtual memory pool.

We test the virtual memory references using the default kernel.C. The result is given as follows:

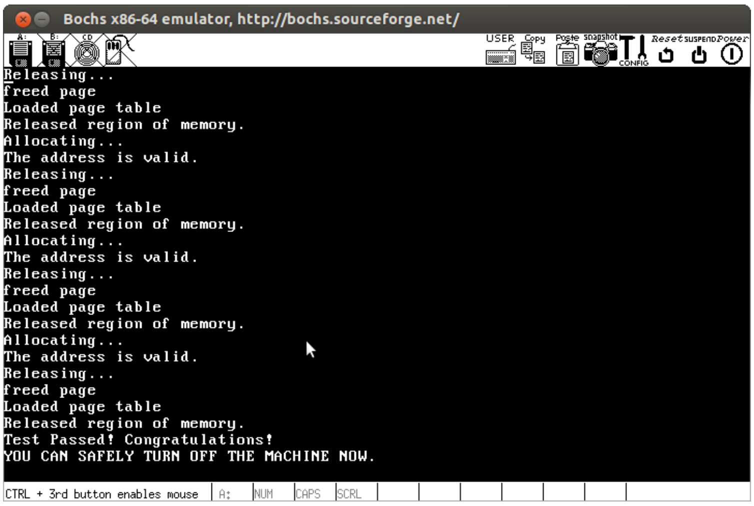

Figure 3.7. Virtual memory reference test.