

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.

```
27
28 //
29 PageTable::PageTable()
30 {
31     page_directory = (unsigned long *) (process_mem_pool->get_frames(1)*FRAME_SIZE);
32     page_table = (unsigned long *) (process_mem_pool->get_frames(1)*FRAME_SIZE);
33
34
```

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

```
58
59 //set the last PDE to point to itself
60 page_directory[shared_size/PAGE_SIZE-1] = (unsigned long)page_directory|3;
61
```

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.

```

89 //
90 void PageTable::handle_fault(REGS * _r)
91 {
92     // this address recursively points to the page directory
93     // by referencing itself twice
94     unsigned long *page_Dir = (unsigned long *)0xFFFFF000;
95     unsigned long *page_Table;
96
97     unsigned long address;
98     unsigned long error = _r->err_code;
99
100
101     unsigned long dir_index;
102     unsigned long pt_index;
103
104     dir_index = address >> 22;
105     pt_index = (address >> 12) & 0x3FF;
106
107     if ((page_Dir[dir_index] & 1) == 0){
108         Console::puts("Page table is not present.\n");
109
110         // get a frame from process mem pool
111         page_Dir[dir_index] = (unsigned long)((process_mem_pool->get_frames(1)<<12)|3);
112
113         // get the address of the new page table
114         page_Table = (unsigned long *) (0xFFC00000|(dir_index<<12)); // it points to multiple of 4KB
115
116         // initialize page table
117         for (int i=0; i<ENTRIES_PER_PAGE; ++i){
118             page_Table[i] = 0;
119         }
120
121         // load the frame into page table
122         page_Table[pt_index] = (process_mem_pool->get_frames(1)<<12) | 3;
123
124         Console::puts("handled page fault.\n");
125     }else{
126         Console::puts("Page table is present.\n");
127
128         // get the address of the present page table
129         page_Table = (unsigned long *) (0xFFC00000|(dir_index<<12));
130
131         // load the frame into page table
132         page_Table[pt_index] = ((process_mem_pool->get_frames(1))<<12) | 3;
133
134         Console::puts("handled page fault.\n");
135     }
136 }

```

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:

```

Bochs x86-64 emulator, http://bochs.sourceforge.net/
USER Copy Paste Snapshot Reset/Suspend/Power
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!
EXCEPTION DISPATCHER: exc_no = <14>
Page table is not present.
handled page fault.
EXCEPTION DISPATCHER: exc_no = <14>
Page table is present.
handled page fault.
EXCEPTION DISPATCHER: exc_no = <14>
Page table is present.
handled page fault.
DONE WRITING TO MEMORY. Now testing...
Test Passed! Congratulations!
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
CTRL + 3rd button enables mouse | A: | NUM | CAPS | SCRL | | | | | | | | | |

```

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”.

```

61
62     for (int i=0; i<vmpool_max_num; ++i){
63         vmpool_manager[i] = NULL;
64     }
65
171 //
172 void PageTable::register_pool(VMPool * _vm_pool)
173 {
174     int flag = -1;
175
176     for(int i=0; i<vmpool_max_num; ++i){
177         if (vmpool_manager[i] == NULL){
178             flag = i;
179             vmpool_manager[flag] = _vm_pool;
180             Console::puts("register VM pool successfull.\n");
181             break;
182         }
183     }
184
185     if (flag < 0){
186         Console::puts("register VM pool failed.\n");
187     }
188
189
190
191 }

```

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.

```

194 //
195 void PageTable::free_page(unsigned long _page_no)
196 {
197     unsigned long dir_index = _page_no >> 22;
198     unsigned long pt_index = (_page_no >> 12) & 0x3FF;
199     unsigned long *page_Table = (unsigned long *) (0xFFC00000 | (dir_index << 12));
200     unsigned long frame_num = page_Table[pt_index];
201     process_mem_pool->release_frames(frame_num);
202
203     Console::puts("freed page\n");
204 }
205

```

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

```

105     address = read_cr2();
106
107     // check if the logical address is legitimate
108     VMPool** vm_manager = current_page_table->vm_pool_manager;
109
110     int check_flag = -1;
111     for (int i=0; i<vm_pool_max_num; ++i){
112         if (vm_manager[i] != NULL){
113             if (vm_manager[i]->is_legitimate(address)){
114                 check_flag = i;
115                 Console::puts("Valid address...\n");
116                 break;
117             }
118         }
119     }
120
121     if (check_flag < 0){
122         Console::puts("Invalid address...\n");
123     }
124     //

```

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.

```

33
34 struct allocator_info{
35     unsigned long base_address;
36     unsigned long region_size;
37 };
38
48 VMPool::VMPool(unsigned long _base_address,
49                unsigned long _size,
50                ContFramePool *_frame_pool,
51                PageTable *_page_table) {
52
53     base_address = _base_address;
54     size = _size;
55     frame_pool = _frame_pool;
56     page_table = _page_table;
57
58     region_num = 0;
59
60     // how many region management info can be stored in a frame at most
61     region_max_num = PageTable::PAGE_SIZE / sizeof(allocator_info);
62
63     // get a frame to store the allocate_list
64     allocate_list = (allocator_info *) (frame_pool->get_frames(1)*PageTable::PAGE_SIZE);
65
66     // register current vm pool
67     page_table->register_pool(this);
68

```

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 vm_pool.

```

74 //
75 unsigned long VMPool::allocate(unsigned long _size) {
76     if (_size == 0){
77         Console::puts("Unable to allocate because size is zero.\n");
78         return 0;
79     }
80
81     unsigned long start_addr = 0;
82     allocator_info temp;
83     unsigned long hole = 0;
84
85     for (int i=0; i<region_num; ++i){
86         for (int j=i+1; j<region_num; ++j){
87             if (allocate_list[j].base_address < allocate_list[i].base_address){
88                 temp = allocate_list[i];
89                 allocate_list[i] = allocate_list[j];
90                 allocate_list[j] = temp;
91             }
92         }
93     }
94 }

```

Figure 3.2. Sorting “allocate_list”.

```

97 // get start_addr based on different conditions.
98 if (region_num == 0)
99 {
100     start_addr = base_address;
101 }
102 else if (region_num == 1)
103 {
104     hole = allocate_list[0].base_address - base_address;
105     if (_size <= hole){
106         start_addr = base_address;
107     }else{
108         start_addr = allocate_list[region_num-1].base_address + allocate_list[region_num-1].region_size;
109     }
110 }
111 else
112 {
113     hole = allocate_list[0].base_address - base_address;
114     if (_size <= hole){
115         start_addr = base_address;
116     }else{
117         for (int i=0; i<region_num-1; ++i){
118             hole = allocate_list[i+1].base_address - (allocate_list[i].base_address+allocate_list[i].region_size);
119             if (_size <= hole){
120                 start_addr = allocate_list[i].base_address + allocate_list[i].region_size;
121             }
122         }
123     }
124     if (start_addr == 0 ){
125         start_addr = allocate_list[region_num-1].base_address + allocate_list[region_num-1].region_size;
126     }
127 }
128 }
129

```

Figure 3.3. Getting starting address.

```

129 // check whether the newly-allocated region is out of the vm pool
130 unsigned long end_addr;
131 end_addr = base_address + size;
132
133 if ((start_addr + _size) <= end_addr)
134 {
135     allocate_list[region_num].base_address = start_addr;
136     allocate_list[region_num].region_size = _size;
137     region_num++;
138     return start_addr;
139 }
140 else if (((start_addr + _size) > end_addr) || (region_num++ > region_max_num))
141 {
142     Console::puts("Unable to allocate because it is out of space.\n");
143     return 0;
144 }
145 }
146
147

```

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.


```

153 void VMPool::release(unsigned long _start_address) {
154
155     unsigned long index;
156     for (int i=0; i<region_num; ++i){
157         if (allocate_list[i].base_address == _start_address){
158             index = i;
159             page_table->free_page(_start_address); //release the pages for current region
160             break;
161         }
162     }
163
164     // delete entry and reconstruct allocate_list
165     if (region_num > 1){
166         for (int j=0; j<region_num; ++j){
167             if (j>index){
168                 allocate_list[j-1] = allocate_list[j];
169             }
170         }
171     }else{
172         allocate_list[0].base_address = 0;
173         allocate_list[0].region_size = 0;
174     }
175
176     region_num--;
177
178     // flush TLB by reloading CR3
179     page_table->load();
180
181     Console::puts("Released region of memory.\n");
182 }
183

```

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”.

```

186 //
187 bool VMPool::is_legitimate(unsigned long _address) {
188     /*
189     for (int k=0; k<region_num; ++k){
190         Console::puti(allocate_list[k].base_address); Console::puts("\n");
191     }
192
193     Console::puti(_address); Console::puts("\n");
194     */
195
196     // check whether the address is legitimate
197     if (allocate_list){
198         for (int i=0; i<region_num; ++i){
199             if ((allocate_list[i].base_address <= _address) && (_address < allocate_list[i].base_address +
200                 Console::puts("The address is valid.\n");
201                 return true;
202             }
203         }
204     }
205     Console::puts("The address is invalid.\n");
206     return false;
207 }

```

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:

```

Bochs x86-64 emulator, http://bochs.sourceforge.net/
Releasing...
freed page
Loaded page table
Released region of memory.
Allocating...
The address is valid.
Releasing...
freed page
Loaded page table
Released region of memory.
Allocating...
The address is valid.
Releasing...
freed page
Loaded page table
Released region of memory.
Allocating...
The address is valid.
Releasing...
freed page
Loaded page table
Released region of memory.
Test Passed! Congratulations!
YOU CAN SAFELY TURN OFF THE MACHINE NOW.
CTRL + 3rd button enables mouse  A: NUM CAPS SCRL

```

Figure 3.7. Virtual memory reference test.