Design for MP2

1. Implementation of frame pool

1.1. Constructor

— FramePool(unsigned long _base_frame_no, unsigned long _nframes, unsigned long _info_frame_no)

In the constructor function, we first decide this is a kernel frame pool or process frame pool by checking on _info_frame_no. Then we record the values like base frame number and the maximum capacity for this object. Also it is necessary to record these information into static parameters for the whole class (see 1.4).

Then we set up the bitmap. For kernel memory pool, the bitmap starts at 0x200000; for process memory pool, the bitmap starts at where it is told by _info_frame_no. If the corresponding bit of a frame in bitmap is 1, indicates that frame is available; otherwise, it is not. First we initialize all the valid bit to 1, and then set the corresponding bit of the frame storing the bitmap to 0.

1.2. unsigned long get_frame()

To get a free frame, we need to check the available bit. We initialize an unsigned long int type variable called searcher and set it to 0x80000000. Every time, the bit that equals to 1 shift 1 bit to the right. By using bitwise and operation on this searcher and bitmap, we can get the first available bit that is not equal to 0, then we can return the corresponding frame number of that frame.

1.3. void mark_inaccessible(unsigned long _base_frame_no, unsigned long _nframes)

In this function, we set all the corresponding bit on the bitmap to 0, according to the arguments passed to this function.

1.4. static void release_frame(unsigned long _frame_no)

The released frame can belong to either in kernel frame pool or process frame pool. First we need to decide which pool the frame goes. Because we already record the base frame number and the capacity of each pool as static parameters, we can easily check the released frame falls into which range. After that, we fetch the corresponding bitmap and change the available bit of that frame to 1.

2. Implementation of page table

2.1. void init_paging(FramePool* kernel_mem_pool, FramePool* _process_mem_pool const unsigned long _shared_size)

In this function, we set some global parameters for the paging system. Because paging should not be enabled now, we need to set paging_enable to 0. Then we set kernel memory pool and process memory pool.

2.2. Constructor — PageTable()

In this function, we set the page directory and the first page table. The way to do this is to get two frames from the kernel frame pool, the first page table is to directly map the first 4MB address, which can exactly fit into one page table. After that, we put the address of this page table into directory.

All the entries mentioned above are set to supervisor, read and write, present mode. As for the other entries on the page directory, we can simply fill 0x0 with supervisor, read and write, not present mode.

2.3. void load()

This function is used to load page directory for a process. Every time the process is switched, the address space will change, we need to load the new page directory for that process.

The approach is to write the page directory to register CR3, and let the page table pointer current_page_table points to the current page table object.

2.4. void enable_paging()

When this function is called, paging will be enabled and protected mode is turned on. To complement that, we need to set MSB of CR0 to 1, and then set the paing_enabled bit to 1.

3. Implementation of page fault handler

void handle_fault(REGS* _r)

This function is triggered by page fault. The reason can be either reading an invalid page entry or touching a protected memory. To know that, we read the last bit of the error code — if it is 1, means that it is a page-protection violation, we do nothing; otherwise, it is caused by non-present page, the page fault address is stored in CR2, and we do the following:

First, check if the page fault happens on page directory table or page table. To know that, we use the page fault address to find the corresponding entry on the page directory table. If the present bit of that entry is 1, page fault happens on page table; otherwise, it happens on page directory table.

If the page fault happens on page directory table, we first need to fetch a free frame for the new page table, and fill its address onto directory. Then, we need to fetch another free frame for the new page, and fill this address onto the corresponding entry of the new page table.

If the page fault happens on page table, we just need to fetch one free frame and fill its address onto the corresponding entry of its page table, and we are done.