Machine Problem 3: Page Table Management

Introduction

The objective of this machine problem is to get you started on a demand-paging based virtual memory system for our kernel. You will study the **paging mechanism on the x86 architecture** and set up and initialize the **paging system** and the **page table infrastructure** for a single address space, with an eye towards extending it to multiple processes, and therefore multiple address spaces, in the future. In this machine problem we limit ourselves to managing small amounts of memory, which in turn allows us to keep the page table in physical memory. (In a future MP we will store the page table itself in virtual memory!)
Make sure that you are familiar with the lesson on "Paging on the x86" before starting work on this Machine Problem.

Page Management in Our Kernel

The memory layout in our kernel looks as follows:

- The total amount of memory in the machine is 32MB.
- The memory layout is such that the first 4MB are reserved for the kernel (code and kernel data) and are shared by all processes.
- Memory within the first 4MB will be **direct-mapped** to physical memory. By this we mean that logical address say 0x01000 will be mapped to physical address 0x01000. Any portion of the address space that extends beyond 4MB will be **freely mapped**: every page in this address range will be mapped to whatever physical frame was available when the page was allocated.
- The first 1MB contains all global data, memory that is mapped to devices, and other stuff.
- The actual kernel code starts at address 0x100000, i.e. at 1MB.

In this machine problem we limit ourselves to a single process, and therefore a single address space. When we support multiple address spaces later, the first 4MB of each address space will map to the same first 4MB of physical memory, and the remaining portion of the address spaces will map to non-overlapping sets of memory frames.

The paging subsystem represents an address space by an object of class `PageTable` to the rest of the kernel. The class `PageTable` provides support for paging in general (through static variables and functions) and address spaces. The page table is defined as follows:

```
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 FramePool     * kernel_mem_pool;   /* Frame pool for the kernel
                                               memory */
```

```
  static FramePool      * process_mem_pool;  /* Frame pool for the process
                                                memory */
  static unsigned long    shared_size;       /* size of shared address space */

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

public:
  static const unsigned int PAGE_SIZE        = Machine::FRAME_SIZE;
                                               /* in bytes */
  static const unsigned int ENTRIES_PER_PAGE = Machine::PT_ENTRIES_PER_PAGE;
                                               /* in entries */

  static void init_paging(FramePool * _kernel_mem_pool,
                          FramePool * _process_mem_pool,
                          const unsigned long _shared_size);
  /* Set the global parameters for the paging subsystem. */

  PageTable();
  /* NOTE1: The PageTable *object* still must be stored somewhere! Probably it
      is best to have it on the stack, as there is no memory manager yet...
     NOTE2: It may also be simpler to create the first page table *before*
         paging has been enabled. */

  void load();
  /* Makes the given page table the current table. This must be done once
      during system startup and whenever the address space is switched (e.g.
      during process switching). Loading a page table also flushes the TLB. */

  static void enable_paging();
  /* Enable paging on the CPU. Typically, a CPU start with paging disabled, and
       memory is accessed by addressing physical memory directly. After paging
      is enabled, memory is addressed logically. */

  static void handle_fault(REGS * _r);
  /* The page fault handler. This method will be invoked by the hardware. */
};
```

We tacitly assume that the address space (and the base address of the direct-mapped portion of memory) starts at address 0x0. The `shared_size` defines the size of the shared, direct-mapped portion (i.e. 4MB in our case). The `page_directory` is the address of the page directory page.

The physical memory will be managed by two so-called *Frame Pools*, which you implemented in a previous MP. Frame pools support the **get** and **release** operations for frames. Each address space is managed by two such pools (see `kernel.C`):

- The `kernel_mem_pool`, between 2MB and 4MB, manages frames in the **shared** portion of memory, typically for use by the kernel for its data structures. Note that the kernel pool is located in direct-mapped memory; where physical and logical addresses are identical.
- The `process_mem_pool`, above 4MB, manages frames in the **non-shared** memory portion. (For details see below.) Note that this pool is located in freely-mapped memory, where logical addresses are not the same as physical addresses.

After the pools are initialized, the kernel goes through the following steps:

1. It calls the function `init_paging` to set the parameters for the paging subsystem.
2. Once paging is configured, the kernel sets up the first address space by creating a first page table object[1].
3. The `PageTable` constructor sets up the entries in the page directory and the page table. The page table entries for the shared portion of the memory (i.e. the first 4MB) must be marked valid ("present" in x86 parlance). The remaining pages must be managed explicitly. (For more details see below.) NOTE: The constructor will need to grab a frame for the page directory; so make sure that you have access to a configured frame pool before you initialize the page table. Before returning, the constructor stores all the relevant information in the page table object.
4. After the page table is created, the kernel loads it into the processor context through the `load()` function. The page table is loaded by storing the address of the page directory into the `CR3` register. The hardware, when needing to translate a virtual address into its corresponding frame, will know to start "walking" the page tables by getting the address of the page directory from `CR3`. During a context switch, the system simply loads the page directory of the new process to switch the address space.
5. After everything is set up correctly, the kernel switches from physical addressing to logical addressing by **enabling** the paging through the `enable_paging()` function. The paging is easily enabled by setting a particular bit in the `CR0` register. Be careful that the page directory and page table is set up and loaded correctly **before** you turn on paging!

The essence of this MP is to first set up the page table correctly and then to implement the method `PageTable::handle_fault()`, which will handle the page-fault exception of the CPU. This method will look up the appropriate entry in the page table and decide how to handle the page fault. If there is no physical memory (frame) associated with the page, an available frame is brought in and the page-table entry is updated. If a new page-table page has to be initialized, a frame is brought in for that, and the new page table page and the directory are updated accordingly.

Frame Management above the 4MB Boundary

The memory below the 4MB mark will be **direct mapped**, and requires no additional management after the initial setup of the page tables. The memory **above** 4MB will have to be managed. The memory addressable by a single process in the x86 is 4GB. It is unlikely that a single process will need that much memory ever. Since we cannot predict which portions of the address space will be used, we will map the **used** portions of logical memory to physical memory frames. By default, memory pages above the 4MB mark have initially no physical memory associated. Whenever such a page is referenced, a page fault occurs (Exception 14), and a **page fault handler** takes over. The page fault handler performs the following steps:

1. It finds a free frame from a common **free-frame pool**.
2. It allocates this new frame to the process.
3. The page entry is updated accordingly.

[1] Remember that we don't have a memory manager yet, and the `new` operator does not work. Therefore, we create the first page table object on the stack (we did the same before with the frame pools).

4. The page fault handler returns.
5. The CPU then retries the instruction and this time finds the physical-to-virtual mapping ready to be used in the paging data structures.

A Note about the First 4MB

Don't get confused by the fact that the kernel frame pool does not extend across the entire initial 4MB, and ranges from 2MB to 4MB only. The first MB contains the GDT[2], the IDT[3], video memory, and other stuff. The second MB contains the kernel code and the stack space. We don't want to hand part of the memory to the kernel to store its own data. **Nevertheless, do not forget to initialize the page table to correctly map the entire first 4MB!**

Where to store Memory Management Data Structures

Given that we don't have a memory manager yet, we find ourselves in a bit of a dilemma when it comes to storing the data structures needed for the memory management. The stack space is limited, so it has to be used judiciously. A better solution is to request frames from the appropriate pool and store the data structures there. (The objects themselves, such as the page table object or the frame pool objects, can of course be stored on the stack. These objects are small, mostly pointers to data structures that are held elsewhere.)

The page directory and the page table pages can be stored in kernel pool frames; so can the management information for the process frame pool[4].

The management for the kernel frame pool is maintained inside the kernel frame pool itself (we took care of this in the implementation of the `FramePool` class). The question now is: Where to store the management information for the kernel frame pool? This works like a charm, because this portion of the memory is directly mapped. So nothing bad happens when you turn on paging.

Other Implementation Issues

A few hints that may come in handy for your implementation:

- You enable paging, load the page table, and have access to the faulting address by reading and writing to the registers CR0, CR2, and CR3. The functions to do this are given in file `paging_low.asm` and defined in file `paging_low.H` for inclusion in the rest of your C/C++ programs.

- The page table can be represented as an array of **page table entries**. Each entry has the following structure:

$31 \cdots 12$	$11 \cdots 9$	$8 \cdots 7$	6	5	$4 \cdots 3$	2	1	0
Page frame	Avail	Reserved	D	A	Reserved	U/S	R/W	Present

  where

[2]GDT: Global Descriptor Table, the x86 data structure defining characteristics of various memory areas.
[3]IDT: Interrupt Descriptor Table, the x86 data structure that realizes the interrupt vector table.
[4]Don't put it in the process frame pool. Once you turn on paging, you may not be able to find it anymore!

Page frame =	number of physical frame storing this page
Avail =	feel free to use these bits
Reserved =	reserved by Intel; do not use!
D =	Dirty
A =	Accessed
U/S =	User or supervisor level
R/W =	Read or Read and Write
Present =	use bit

- A page fault triggers Exception 14. This exception pushes a word with the exception error code onto the stack, which can be accessed (field `err_code`) in the exception handler through the register context argument of type `REGS`. The lower 3 bits of the word are interpreted as follows:

value	bit 2	bit 1	bit 0
0	kernel	read	page not present
1	user	write	protection fault

  Also, note that the 32-bit address of the address that caused the page fault is stored in register CR2, and can be read using the function `read_cr2()`.

The Assignment

1. Study the lesson "Paging on the x86". You can find additional material in Read K.J.'s tutorial on "Implementing Basic Paging" (`http://www.osdever.net/tutorials/view/implementing-basic-paging`) to understand how to set up basic paging. Also, the beginning of Tim Robinson's tutorial "Memory Management 1" (`http://www.osdever.net/tutorials/view/memory-management-1`) helps you understand some of the intricacies of setting up a memory manager.

2. Implement the functionality defined in file `page_table.H` (and described above) to initialize and load the page table, and to enable paging. Additional details about how to implement these routines can be found in K.J.'s tutorial.

3. **Hint:** Test the routines with a page table for 4MB of memory and 4MB of the memory being direct-mapped. Because all the memory is direct-mapped, it should all be valid ("present"), and there is no need to bother with a page-fault handler. Make sure that you can address memory inside the 4MB boundary. If – after setting up your page table – the program crashes when you turn on paging, then something is wrong with your code to set up the page table.

4. Once you have convinced yourself that the page table is implemented correctly to handle the direct-mapped memory portion, extend the code to handle more than the 4MB shared memory. For this, you need to add the following:

   (a) The memory beyond the first 4MB will not be direct-mapped, and therefore must be marked as invalid ("not present").

(b) A page-fault hander must be implemented and installed, which is called whenever an invalid page is referenced. It locates a frame in the frame pool, maps the page to it, marks the page as "present", and returns from the exception.

(c) Keep in mind that we have a two-level page table. A page fault can therefore be caused by an invalid entry in the page directory as well as an invalid entry in a page table page. You need to handle both cases correctly for the page fault handler to work.

You should have access to a set of source files, BOCHS environment files, and a makefile that should make your implementation easier. In particular, the `kernel.C` file will contain documentation that describes where to add code an how to proceed about testing the code as you progress through the machine problem.

What to Hand In

You are to hand in the following items:

- A ZIP file, with name `mp3.zip`, containing the following files:
  1. A design document, called `design.pdf` (in PDF format) that describes your implementation of the page table and the page-fault handler.
  2. All the source files and the `makefile` needed to compile the code.
  3. The assignment should not require modifications to files other than `cont_frame_pool.H/C` and `page_table.H/C`. If you find yourself modifying and/or adding other files, add them to the `mp3.zip` file as well. You may have to modify and submit the `makefile` also. **Describe what you modified and why you are modifying and/or adding these additional files in your design document!**
- Any modification to the provided `.H` files must be well motivated and documented. **Do not modify the public interface provided by the `.H` files.** We are using our own test code to check the correctness of your implementations, and we have to ensure that your code is compatible with our tests.
- Grading of these MPs is a very tedious chore. These handin instructions are meant to mitigate the difficulty of grading, and to ensure that the grader does not overlook any of your efforts.
- **Failure to follow the handin instructions will result in lost points.**