Project 3: Page Table Management
CSCE 613 Spring 2016
Due: 2/22/16 11:59pm CT

Introduction

The objective of this 3rd project is to get you started on a demand-paging virtual memory system for our kernel. You will continue to study the paging mechanism of 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. For P3, we will still assume that the memory we need to handle is not large: it is possible to fit all the frames that we need for page tables in the area directly-mapped memory area.

You will also design support for multiple page sizes.

Simple Paging in the x86 Architecture

The x86 architecture supports what can be called "paged segmentation." In the following we will ignore segmentation, and instead, limit ourselves to pure paging. Paging in the x86 uses a two-level scheme, with a page directory and page tables. This means that the 1st level in the paging contains a single table called *page directory*. Each of the entries in the page directory points to a single-page page table. The entries in the page tables then point to frames in memory.

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 is direct-mapped to physical memory. In other words, 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; that is, every page in this address range will be mapped to whatever physical frame is 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;

Stop now and do some back-of-the-envelope calculations: how many frames you have that are not directly-mapped? Does your P2 solution for FramePool management work for this number of frames, i.e., can the data structure you used for management accommodate the number of frames you need?

In this project, 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 you are implementing for your kernel should represent an address space by an object of class PageTable. 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 page directory
                                                  located? */

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();
    /* Initializes a page table with a given location for the directory and the
     * page table proper.
      * NOTE 1:
      * The PageTable object still needs to be stored somewhere! Probably it is best
      * to have it on the stack, as there is no memory manager yet...
      * NOTE 2: It may also be simpler to create the first page table *before* paging
      * has been enabled. Look at provided code for Kernel.C
    */

    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).
     */

    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 when
     * it is not able to find the page tables the translation it needs
     */
};
```

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. Notice that the page directory is an object of the class PageTable; it just happens to be the "top" page table, i.e., the first in the hierarchy of page tables that provide paging functionality. We target the x86 architecture, so we have 2 levels in the hierarchy, with the page directory being the name we refer to as the 1st level in the hierarchy.

The physical memory is be managed by the so-called Frame Pools that you implemented in P2. You are required to use the FramePool implementation you submitted in P2 as a start point, and you can change it as you see necessary (e.g., bug fixes). Do not forget to document changes in your design document. Make sure that any fixes and changes in FramePool.[CH] are properly documented in the corresponding commits in your github repository. As I hope you still recall, FramePools supports *get* and *release* operations for frames. Each address space is managed by two such pools:

- The kernel memory pool manages frames in the shared portion of memory, typically for use by the kernel for its data structures;
- The process memory pool manages frames in the shared memory portion. (More details below.)

The kernel (see provided file *kernel.C*) would typically set up two frame pools (i.e., the first for kernel data between 2MB and 4MB and the second for process data above 4MB) and then call the function *init_paging*.

After that, the first address space is set up by creating a first page table object. 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 in Kernel.C; those objects are alive as long as function Kernel.C:main() is still running.) The page table 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), and properly set for a directly-mapped translation (physical address is the same as virtual.) The remaining pages must be managed explicitly. (For more details see below.) [NOTE: Make sure that you have access to the physical memory for the page directory and the page table before you initialize the page table. This means that when the CPU needs to access the page directory to do translation, it knows how to translate that particular address.] Before returning, the constructor stores all the relevant information in the page table object.

After the page table is created, we load 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. Later on (in P5), when you are supporting multiple processes, during a context switch from one process to another, your paging system simply loads the page directory of the new process into CR3 to switch the address space (i.e., to use the correct page tables.)

After everything is set up correctly, we switch 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 project is to implement the method PageTable::handle_fault(), which will look up for the page that the hardware has faulted on, creating a PageTable object if necessary and updating page entries (in the page directory and page table) as needed.

You already implemented the Frame Management in P2

We need an allocator for physical-memory frames, which allows us to get and release frames to be used by the kernel or by user processes. Whenever the kernel needs frames in direct-mapped memory (i.e. below the 4MB boundary,) it gets them from the kernel frame pool, which is located between 2MB and 4MB. Non-kernel data is then allocated from the process frame pool, which manages memory above 4MB. If you completed P2, you have the frame management code ready to be used. If you skipped P2, you will need to catch up to proceed with P3.

Frame Management above the 4MB Boundary

The memory below the 4MB mark will be directly-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 a 32-bit architecture). For many applications, it is unlikely that their execution results in a process needing all that much memory. 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 a page is referenced for the 1st time, a page fault will occur (*Exception 14*), and a *page fault handler* takes over (the one that you are to implement in P3.) The page fault handler finds a free frame from a common *free-frame pool* and allocates it to the process. The page entry is updated accordingly, and the page fault handler returns. (So that the CPU can retry the instruction, this time finding 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, only ranging from 2MB to 4MB. The first MB contains data structures such as GDT[1], IDT[2], video memory, etc. The second MB contains the kernel code and the stack space. You leave those first 2MB alone, and use the last 2MB for the memory needed by the kernel. In P2, you already used a few frames of this area to store the data structures you defined in FramePool.

[Once more, so you do not forget!] Nevertheless, do not forget to initialize the page table to map correctly the entire first 4MB!

---

[1] GDT: Global Descriptor Table, the x86 data structure defining characteristics of various memory areas.

[2] IDT: Interrupt Descriptor Table, the x86 data structure that realizes the interrupt table.

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 (i.e. page directory, page table pages, management information for frame pools, and so on). We have two alternatives:

1. We can store the data structures on the stack – by defining the variables to be local to the *main()* function. This is not a good idea primarily for two reasons: First, the stack is limited in size. Second, it will be cumbersome to make the (local) variables available globally if needed later. 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.)
2. 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[3]. The question now is: Where to store the management information for the kernel frame pool? You probably already understand this after coding P2: the interface for the kernel pool is set up so that you can store the management information inside the pool itself. (Remember the constructor arguments for class FramePool?) Here you can do this, because the portion of the memory you are using is directly mapped. So nothing bad happens when you turn on paging.) For example, simply reserve the first few frames (the number depends on how you manage the pool and on it size) for management purposes. Make sure that you mark them as "used"; otherwise, your allocator may hand them out to users.

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.
- 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() .)

[3] Don't put it in the process frame pool. Once you turn on paging, you may not be able to find it anymore!

The Assignment: Part I

- In P2, you were asked to read the following documents. If you have not done so, you have to do it as soon as possible. If you did read but all is still confusing, you may want to read again.:
    - K.J.'s tutorial on Implementing Basic Paging (at http://www.osdever.net/tutorials/view/implementing-basic-paging) to understand how to set up basic paging;
    - Read at least the beginning of Tim Robinson's tutorial Memory Management 1 (at http://www.osdever.net/tutorials/view/memory-management-1) to understand some of the intricacies of setting up a memory manager;
- Implement the functionality defined in file *page_table.H* (and described above) to initialize and load the page table, and to enable paging. Details about how to implement these routines can be found in K.J.'s tutorial. 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.
- Once you have convinced yourself that the page table is implemented correctly to handle the direct-mapped memory portion, make sure you push this code to your tamu github repository so that we can verify that you have been developing your code incrementally as requested here.) Then you extend the code to handle more than the 4MB shared memory. For this, you need to add the following:
    - The memory beyond the first 4MB will not be direct-mapped, and therefore must be marked as invalid ("not present");
    - A page-fault hander must be implemented and installed, which is called whenever an invalid page is referenced. The handler checks whether the page is within the limits of the memory managed by the page table. If so, it locates a frame in the frame pool, maps the page to it, marks the page as "present", and returns from the exception. You already have implemented in P2 the functionality for the frame pool, i.e. implement the allocator and de-allocator. Now you have to implement the page-fault exception handler.

The zip file P3-provided-code.zip, available on eCampus, provides a set of source files, BOCHS environment files, and a makefile that should make your implementation easier. In particular, the *kernel.C* file contains documentation that describes where to add code and how to proceed with testing the code as you progress through P3.

The Assignment: Part II

Besides implementing Part I of this assignment, you need to describe a design for supporting two page sizes. You are not required to implement your design, you only have to describe how it would work:

- write a report on how your design could be updated to support two different page sizes: 4K and 16M. (For information on page sizes, see Appendix A.) Describe in detail what changes are needed in your data structures and in the functions you implemented;

- don't forget to describe any changes needed for frame management.

What to Hand In

You are to hand in the following items:
A ZIP file containing the following files:

- A design document, called *design.pdf* (in PDF format) that describes your implementation of the page table, the frame pool, and the page-fault handler;
- A second design document, design2pagesizes.pdf (in PDF format) that describes how their design is updated to handle two different page sizes (e.g. 4K and 16M);
- A pair of files, called *page_table.H* and *page_table.C*, which contains the definition and implementation of the required functions to initialize and enable paging, to construct the page table, and to handle page faults. Any modification to the provided .H file must be well motivated and documented;
- A pair of files, called *frame_pool.H* and *frame_pool.C*, which contain your implementation of the frame pool. If these files are different from the ones you submitted in P2, clearly describe the changes. Any modifications to the provided file *frame_pool.H* must be well motivated and documented,

Grading of P3 is a very tedious chore. These hand-in 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 hand-in instructions will result in lost points.

Project Grading Criteria for P3

Item	**Points**	Comments
Kernel boots	20	This means that you get something if your project boots. Make sure it compiles cleanly.
Feature completeness and functional correctness	40	
Efficiency	10	The code will be reviewed to see how efficient (in space and time usage) your solution is
Design document for implemented	15	

part		
Design document for supporting 2 page sizes	15	
Grand Total	100	

Policy for Late Submissions

You start the semester with 4 free days that can be used towards any project this semester. (You should know how many you used for P1 and P2.) If you run out of free days, a late assignment will be accepted up to 24 hours after the deadline with a penalty of 30%.

Appendix A – Supporting multiple page sizes

Most architectures use 4KB (4096 bytes) as the page size. With hardware available for increasing memory sizes, most computer architectures also offer larger page sizes. System efficiency can be improved by deploying the page size that best fits application needs, for example with large data structures backed up using large pages and, therefore, minimizing the number of virtual to physical address translations that need to be managed. For example, x86 offers 4 MB, x86-64 2 MB, Power 64MB and 16 MB, and ARMv7 1 MB and 16 MB.

Your page size and frame size are the same. Notice that this project description uses "KB" and "MB" to refer to 1024 and 1024*1024 bytes. This is common use, but there is some ambiguity with "k" meaning 1000. Unambiguous, but less familiar, is the notation of KiB and MiB. If you care, you can find more information at https://en.wikipedia.org/wiki/Kibibyte

Appendix B – Another document that may help

Students in previous years have said that a powerpoint presentation (that I believe was prepared by a TA for the course a few years ago) has been useful. I find the material in the presentation imprecise and even incorrect in simplifications at some passages, so I do not totally endorse it. I edited the document to remove parts that were incorrect, and left the rest even though I do not agree with the way it is presented. But there is no real harm in the imprecisions, and it may help a student who needs guidance on where to start, so you will find this presentation in eCampus (file P3-presentation-that-may-help.pdf).