Project 2: Page Table Management
CSCE 613 Spring 2016
Due: 2/5/16 11:59pm CT

Introduction

The objective of this 2nd project is to get you started on a demand-paging virtual memory system for our kernel. You begin by implementing a frame manager, which manages the allocation of frames (physical pages) to address spaces.

Assumptions about our kernel

The memory layout in our kernel looks as follows:

- the total amount of memory in the machine is 32MB;
- the memory layout will be so 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. In other words, logical address 0x01000 will be always 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 was available when the page was allocated;
    - the first 1MB contains all global data, the memory that is mapped to devices, and other stuff;
    - the actual kernel code starts at address 0x100000, i.e. at 1MB;

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 physical memory will be managed by two so-called Frame Pools, which support *get* and *release* operations for frames:

- The *kernel memory pool* manages frames in the shared portion of memory, i.e., the portion of the 1st 4MB of memory that is used for the kernel's data structures. Notice that memory allocated here will be shared among all processes in the system (we only support one for now, but once we support more, this continues to be shared).
- The *process memory pool* manages frames in the rest of the memory (i.e., pages starting at 4MB until the end of the 32MB memory we have).

The kernel (see file *kernel.C*) would typically set up two frames. On P3 it will then go onto initiating the paging system. For now, get this two frame managers to work.

The project: Frame Management

You will notice that 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.

Frame Management above the 4MB Boundary

The memory below the 4MB mark will be directly-mapped, so logical-to-physical mapping is trivial (the same address). The memory above 4MB will have to be managed, as we allocate frames, we need to populate the data structures that will allow the hardware to find the logical-to-physical translations. You will be doing that on P3. At that time, when a virtual address appears and we conclude it has never been mapped yet (it has never been used, so we never allocated a frame for it), your P3 code will ask for a free physical page. The functionality of managing free pages is what you are doing now.

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.

The Frame Pool

Available physical memory frames are managed in *FramePool* objects, which provides the following interface:

```
class FramePool {
    private:
    /* -- DEFINE YOUR FRAME POOL DATA STRUCTURE(s) HERE. */

    public:
    FramePool(unsigned long _base_frame_no,
    unsigned long _nframes,
    unsigned long _info_frame_no);
    /* Initializes the data structures needed for the management of this
     * frame pool. This function must be called before the paging system
     * is initialized.
     * _base_frame_no is the frame number at the start of the physical memory
     * region that this frame pool manages.
     * _nframes is the number of frames in the physical memory region that this
     * frame pool manages.
     * e.g. If _base_frame_no is 16 and _nframes is 4, this frame pool manages
     * physical frames numbered 16, 17, 18 and 19
     * _info_frame_no is the frame number (within the directly mapped region) of
     * the frame that should be used to store the management information of the
     * frame pool. However, if _info_frame_no is 0, the frame pool is free to
     * choose any frame from the pool itself to store management information.
     */
```

[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.

```
    unsigned long get_frame();
    /* Allocates a frame from the frame pool. If successful, returns the frame
     * number of the frame. If fails, returns 0.
     */

    void mark_inaccessible(unsigned long _base_frame_no,
    unsigned long _nframes);
    /* Mark the area of physical memory as inaccessible. The arguments have the
     * same semantics as in the constructor.
     */

    static void release_frame(unsigned long _frame_no);
    /* Releases frame back to the given frame pool.
     * The frame is identified by the frame number.
     * NOTE: This function is static because there may be more than one frame
     * pool in the system.
     */
};
```

Such a pool can be implemented using a variety of ways, such as a free list of frames, or a bitmap describing the availability of frames. The bitmap approach is particularly interesting in this setting, given that the amount of physical memory (and, therefore, the number of frames) is very small (28MB if we discount the direct-mapped first 4MB); a bit map for 32MB would need 8k bits, which easily fit into a single page.

Note: Unfortunately, there are some sections in the physical memory that you should not be touching. In addition to the various locations within the first 1MB where the memory is used by the system and is therefore not available for the user (we safely ignore these portions because the kernel pool starts at 2MB), there are other portions of memory that may not be accessible. For example, there is a region between 15MB and 16MB that may not be available, depending on the configuration of your system. This region is in the middle of our process frame pool. Our frame pools support the capability to declare portions of the pool to be off-limits to the user. Such portions are defined through the function *mark_inaccessible*. Once a portion of memory is marked inaccessible, the pool will not allocate frames that belong to the given portion.

The Assignment

- Implement a frame pool manager as defined in file *frame_pool.H*. Note that the pool allocates a single frame at a time, making the implementation very easy! Test your code. Add functionality to kernel.C that invokes the methods you implemented. You will not turn this code in; we will use our own testing when grading your code;
- You are done with the coding part of this assignment, feel free to submit your code as explained in the next section ("What to Hand in");
- Start preparing for the next project:
    - Read 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;

The zip file P2-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 about testing the code as you progress through this project. Notice that:

- Only Makefile.linux64 is provided, because this is the one I used when testing my own environment. If you need a different one for your environment, get the one from P1 and change it based on what you see on the linux64 version;
- When you first run make, it should fail. I did not provide a dummy frame_pool.C, so make tells you it can't proceed. Create your frame_pool.C and it should build to completion.

If the Makefile or any of the provided code causes you trouble, post on Piazza to get help and then later help other people.

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 frame pool;
- A pair of files, called *frame_pool.H* and *frame_pool.C*, which contain the definition and implementation of the frame pool. Any modifications to the provided file *frame_pool.H* must be well motivated and documented;

Grading of these MPs 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 P2

Item	**Points**	Comments
Kernel boots and our tests run to completion without crashing	20	This means that your FramePool implementation does not blow up when stressed with our tests.
Feature completeness	25	Every method was implemented, and it works on naïve tests
Feature correctness	20	Almost all of our test cases work
Feature correctness +	10	All of our test cases work
Quality of the design and of the design documentation	15	Your choice of data structures. The clarify of your description
Code documentation	10	No specific guidelines, just use good practices.
Grand Total	100	

Policy for Late Submissions

You start the semester with 4 free days that can be used towards any project this semester. 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%.