Machine Problem 2: 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. You will also implement a **frame manager**, which manages and allocates frames to address spaces.

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 the entries of a single-page **page_directory** pointing each to a single-page **page table**. The entries in the page tables then point to frames in memory.

Hands-on details about the implementation of page management and memory management for the x86 can be found in K.J.'s Tutorial on "Implementing Basic Paging" (http://www.osdever.net/tutorials/view/implementing-basic-paging) and more details in the two tutorials "Memory Management 1/2" by Tim Robinson (http://www.osdever.net/tutorials/view/memory-management-1).

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 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. 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 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 was available when the page was allocated. 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 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, duh! */
  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: 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...
     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). */
  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. */
};
```

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 support `get` and `release` operations for frames. Each address space is managed by two such pools: The `kernel_mem_pool` manages frames in the shared portion of memory, typically for use by the kernel for its data structures; the `process_mem_pool` manages frames in the shared memory portion. (For details see below.)

The kernel (see 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 4MP,) 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). 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). 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. 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. During a context switch, the system simply loads the page directory of the new process to switch the address space.

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!

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 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 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.

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 GDT's, IDT's, 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!

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.
      */
   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 bit-map describing the availability of frames. The bit-map 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 anyway) 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.

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: 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.)

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[1]. The question now is: Where to store the management information for the kernel frame pool? The interface for the kernel pool is set up so that you can store the management information inside the pool itself. (Here you can, because this portion of the memory 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()`.

The Assignment

1. 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.

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

2. Read at least the beginning of Tim Robinson's tutorial "Memory Management 1" (http://www.osdever.net/tutorials/view/memory-management-1) to understand some of the intricacies of setting up a memory manager.

3. Implement a simple frame pool manager as defined in file `frame_pool.H`. Note that the pool allocates a single frame at a time. This will make the implementation very easy!

4. 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.

5. 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. 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 will have to implement the functionality for the frame pool, i.e. implement the allocator and de-allocator, and realize the page-fault exception handler.

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 containing the following files:

  1. 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.
  2. A pair of files, called `page_table.H` and `page_table.C`, which contain 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.
  3. 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.**