Project 3: Demand Paging

Due Date: 3/19/21 11:59pm CT

Item	Deadline	Early bonus info
Midterm	3/16	
P3 part A	3/19	10 points if submitted by 3/04
P3 part B	3/19	10 points if submitted by 3/16
P4	4/05	10 points if submitted by 4/2
P5	4/21	none
P6	5/03	10 points if submitted by 4/30
Final	5/5 8 - 10:30 am	

Project 3 has two parts, with a common deadline. We have early submission bonuses for each part, as specified above.

(Personal note from instructor — Feel free to skip this paragraph.) If you decide to start working on this project (or its final component in project 4) just before the due dates, chances are that you will fail to get it to do anything. A last-minute mad push will consume your time and energy without generating any results towards your grade (or your learning.) I suggest that you book in your calendar regular hours to work on the project, and you stick with it. You know how to honor your calendar, you have done that consistently for many classes that required attendance and for others that did not. Most of you choose to honor your calendar and come to the CSCE 410 class regularly. I suggest that you allocate additional hours in the next four weeks to make regular progress on the projects. I do not doubt your ability to work or your desire to learn and become the best professional you can be. But I do believe that without planning, most of us do not use our time well and the quality (and quantity) of our work suffers. I apologize if I sound patronizing. I am trying to sound like a career coach. Besides, I have selfish reasons: I believe that knowledge in operating systems matters, and I do not want students to have a bad experience in my class. Last-minute work in non-trivial programming assignments can be frustrating and even painful, regardless of how much we enjoy (or not) our work.

Contents

Introduction

The objective of this project 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. We start, in Parts A and B, limiting ourselves to managing small amounts of memory, which in turn allows us to keep the page table in physical memory. In Project 4, we will store the page table itself in virtual memory!

We provide you with a set of source files, BOCHS environment files, and a makefile that should make your implementation easier.

Part A - page table for 4MB of directly-mapped memory

Task A.0 - understand x86 paging

Your first task is to get familiar with the lesson on "Paging on the x86" from Ciro Santilli `https://cirosantilli.com/x86-paging`.

We are working with pure paging (not segmentation), so that section of the document is not relevant to us. Because in this course we discussed multi-level hierarchical paging, many parts of the document won't be new to you. You do not need to learn details on Intel's hardware support, but it is important to note that there are some registers (e.g., CR3) where the hardware expects to find the information about the page tables. Your kernel will need to update these registers to make sure it has the appropriate information.

Task A.1 - Understand the big picture of P3/P4

Read this section to get a sense of what is going on. To start, take a look at the provided kernel.C:

- The kernel needs to be able to receive interrupts from the hardware (indicating that there is a page fault); therefore, the kernel enables timers and interrupt management (lines 82 – 91, and 109). Later on, it associates your method for page fault handling with an interrupt (line 145.)
- The kernel creates the objects to manage the physical frames (using your implementation from P2.)
- The kernel now calls the initialization of the paging system by calling the static method PageTable::init_paging. The paging system now has the information on which frames it has available to use when physical memory is needed by the kernel or by a user process.

- Once paging is configured, the kernel sets up the first address space by creating a page table object (line 153)
  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 in P2 with the frame pools).
- The kernel enables the paging system. From now on, the CPU will treat any address it sees as a virtual address and will have to translate it into its corresponding physical address.
- The kernel starts to use memory. If your memory management is working, the program will finish indicating that the test finished.

Now let's look at what you need to know to implement P3 (i.e., implement the methods for class PageTable.)

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 is reserved for the kernel (code and kernel data) and is shared by all processes.
- Memory within the first 4MB will be **direct-mapped** to physical memory. By this, we mean that a 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.
- FYI: 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 adding support to 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.

You will implement a two-level hierarchical page table. This means that we will have a page directory (the first level) where each entry points to the corresponding page table (the second level.)

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 associated with each process.

Take a look at `page_table.H`. Notice that there are many data members declared as `static`. This means that, regardless of how many PageTable objects we have in the system, these data fields are global to the entire page system. Take a look now at these data fields. The fields keep track of the PageTable objects for the kernel and the process (we assume only one for now) address spaces.

There is one non-static data member within PageTable objects: a pointer to the page directory for the address space. But this is the pointer to the data (i.e., a virtual address), where is the actual page directory data (i.e., within which frame in memory?) The constructor for the PageTable will have to get memory for this page directory. More on that later.

Your job is to implement the PageTable methods. Below information that may be useful:

init_page()

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 P2. 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.

The constructor PageTable()

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, direct-mapped) must be marked valid ("present" in x86 parlance). The remaining pages must be managed explicitly. (For more details, see below.)

Note that 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. (The kernel.C that we provided first sets up the frame pools, then it calls the constructor, so if your P2 code is working, all is well.)
Before returning, the constructor stores all the relevant information in the page table object.

load()

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 (i.e., when moving from one address space to another because the processor is executing another process), the system simply loads the page directory of the new process to switch the address space.

enable_paging()

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! Look at the object data fields (eighter using a debugger or printing the values.)

handle_fault()

(This method is not needed in part A.)
The essence of this project 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, then 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.

Task A.2 - Understand the memory we are managing

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.
4. The page fault handler returns.
5. The CPU then retries the instruction. This time, it 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[1], the IDT[2], 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!**

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

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.

[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 vector table.

Task A.4: Read about implementation Hints/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 ··· 12	11 ··· 9	8 ··· 7	6	5	4 ··· 3	2	1	0
Page frame	Avail	Reserved	D	A	Reserved	U/S	R/W	Present

where

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()`.

Task A.5 - Completing Part A — Implement and test access to first 4 MB.

Implement the PageTable methods assuming that the machine has only 4MB, all of it direct-mapped (i.e., virtual and physical address are the same.) This means that you do not need to implement `handle_fault` yet.

I recommend that you determine which entries of the page directory need to set up when accessing an address that is in the first 4 MB. Think about what happens for such addresses: which values you can have as an index for a page directory entry?
For each page directory entry that you need to set up, you create and initialize the corresponding PageTable object with the direct mapping.

When writing the code initialize and load the page table, you may find it useful to read K.J.'s tutorial on "Implementing Basic Paging"
(`http://www.osdever.net/tutorials/view/implementing-basic-paging`)

If – after setting up your page table – your program crashes when you turn on paging, then something is wrong with your code to set up the page table.

Part B: Beyond the first 4 MB

1. Look over the "Paging on the x86" document (URL provided in Task 0) again to refresh your memory.
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.

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

A page-fault handler 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.

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.

What to hand in

You are to hand in the following items:

- A ZIP file, with name `p3.zip`, containing the following files:
    1. A pair of files, named `page_table.H` and `page_table.C`, that contain the definition and implementation of the required functions to initialize and enable paging, to construct the page table, and to handle page faults.

2. a pair of files, called `cont_frame_pool.H` and `cont_frame_pool.C`, which contain the definition and implementation of class `ContFramePool` that you implemented as part of P2 (with any updates that you made since then.)
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 `p3.zip` file as well. You may have to modify and submit the `makefile` also. **You also need to submit a design document, named design.pdf, where you describe what you modified and why you are modifying and/or adding these additional files.**

- 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 projects 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.**