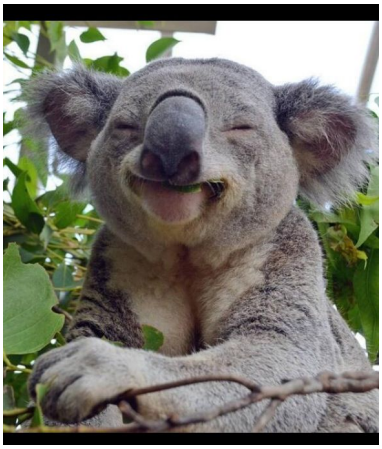
Lab 5 - Virtual Memory

Operating Systems, CS-UH 3010

Assigned:	Nov 15, 2022	Due:	Nov 30, 2022



It's the last lab of the course! We feel good :)
Do not share the start codes or virtual image provided in this lab, as they were kindly provided by Princeton without permission to redistribute.

1 Overview

In this project, you will add memory management and support for virtual memory to the kernel.

Your previous kernels used a single global memory layout, where the kernel and all processes shared access to the same memory, and granted kernel permissions to all user processes.

Now, you will extend the provided kernel with a demand-paged virtual memory manager and restrict user processes to user mode privileges (ring 3) instead of kernel mode privileges (ring 0). You will implement:

1. virtual address spaces for user processes
2. page allocation
3. paging to and from disk
4. page fault handler

Specifically, it will be your job to modify `memory.h` and `memory.c`.

As a result, each process will get its own address space and will not be allowed to use certain instructions that could be used to interfere with the rest of the system. Assuming the kernel and kernel threads have no bugs/vulnerabilities, this will prevent buggy or malicious processes from corrupting the kernel and other processes. Furthermore, paging memory will allow programs to use more data than is available in RAM.

We also provide six test programs (`process[1-4].c` and `th[1-2].c`) that automatically test your memory manager when built. See Section 5 for instructions on how to use these tests.

1.1 Finding Your Way Around this Project

The code for this project is a bit different from the other projects so far (e.g., PCB structure is different.) **Don't Panic!**

You will need to look at `kernel.[ch]` and `tlb.h` to find useful data structures and functions. The test processes and kernel show you how your memory manager will be used.

There are many constants in these files that you'll want to use:

```
grep "PROCESS_STACK" *.h
grep "PROCESS_START" Makefile
grep "PAGE" *.h
```

Note: this start code ships with its own bochs **VGAROM** and **ROM**. You may have to edit the provided bochsrc to get it to point to the right places... but it should be okay!

1.2 Steps

1. Review project specs and download the start code.
2. Prepare your design review
3. Complete memory.h, and memory.c
4. Compile and build the disk image.
5. Test Run the Bochs emulator and play with the small shell program.
6. Complete the general project requirements
7. Submit

2 Page Table Setup

You must set up a two-level page table to map virtual to physical addresses, as described in class and in your course textbook, for each process. The i386 architecture uses a two-level table structure: one page of memory is set aside as a directory that has 1024 entries pointing to the actual page tables. Each of these page tables also has 1024 entries describing pages of virtual memory. You need to understand this structure well.

That last statement needs to be reiterated. You will need to understand what page table entries must look like.

- How are flags set?
- How do you map a virtual address to a particular entry?
- How do you store the physical page address for the entry?

Make use of the provided functions `init_ptab_entry()`, `insert_ptab_dir()`, and `set_ptab_entry_flags()` when setting page table entries, page directory entries, and page entry flags.

Kernel threads all share the same kernel address space and screen memory, so they can use the same page table. Use `N_KERNEL_PTS` (the number of kernel page tables in the kernel page directory) to determine how many kernel pages tables to make. Each kernel page table should contain `PAGE_N_ENTRIES` (1024) entries, until the page base address of a page reaches `MAX_PHYSICAL_MEMORY`. Notice that our initial values for the constants given in memory.h can all fit in one table. It's OK to map more than required (for example, you can map the whole page table).

Implement `init_memory()` to initially map your physical memory pages to run the kernel threads.

3 User Process Virtual Address Space

Map kernel and screen to their physical addresses (virtual = physical)

Map addresses for the process' code and data segments to its own pages. This means that each process has four types of pages: page directory, page table, stack page table, and stack pages. Look at `PROCESS_START` and `PROCESS_STACK` for an idea of where in virtual memory everything resides.

You should allocate `N_PROCESS_STACK_PAGES` pages for a process stack. Note

- Kernel memory: inaccessible
- Screen memory: read-write
- Other memory: read-write

Implement `setup_page_table()` to load the code and data of a user process into its virtual address space. You will need to allocate its pages and load the process. The page directory of a task is determined by `pcb->page_directory`. This needs to be set when you're setting up the page tables.

There are two more things that you will need to set up the address space and that will help you with keeping track of page mappings:

1. You will need a way to obtain the physical address of a page based on its virtual address. To do this, implement the `page_addr()` function.
2. To help you keep track of all pages and their metadata (e.g. whether or not it's pinned), you should define the `page_map_entry_t` data structure in memory.h

4 Page Allocator

Write a physical page frame allocator, `page_alloc()`, to prepare a free page of physical memory. If there are free pages available, the job is simple: pick one. Otherwise, you must choose one to page out to disk, using the policy implemented in `page_replacement_policy()`, and then reset the page to be used again. For this assignment, any simple replacement policy is acceptable (e.g. FIFO).

Some pages may be so important or frequently used (or it might just be more convenient for you) that they should never be paged out. For example, you should never evict page tables. Such pages are pinned. Implement a mechanism for pinning pages so that they are never evicted by your page allocator.

We can artificially limit the amount of physical RAM available to your OS by setting `PAGEABLE_PAGES` to some lower number (try the 20s) in memory.h. This way, the total available physical memory is smaller than the total memory needed to run all the processes, requiring the system to use your paging implementation. This is a helpful value to change for testing.

Note: You will probably want to clear the page at this point. Doing this later will be hard!

4.1 Paging to/from Disk

If there are no more free physical pages for your page allocator to allocate, you must choose a page to evict, writing it to disk using `scsi_write()` in `scsi.h: implement page_swap_out()`. You also must page in from disk in `page_swap_in()`.

Check out the PCB fields `swap_loc` and `swap_size`. They will be very helpful here.

You may assume that processes do not use a heap for dynamic memory allocation: a process does not change size. In this case, the user memory space of a process contains an interval of `pcb->swap_size` sectors starting from `PROCESS_START`, plus the memory for user stack.

Note: This is a good place to make sure that you are flushing the TLB appropriately. You will find the functions in scsi.h helpful for this part of the assignment.

4.2 Page Fault Handler

You will need to write a page fault handler in `page_fault_handler()` to handle the case where a process tries to access a page that is not currently in RAM. The handler should:

1. Find the faulting page in the page directory and page table.
2. Allocate a page of physical memory.
3. Load the contents of the page from the appropriate swap location on the USB disk. (How are you going to figure out the swap location?)
4. Update the page table of the process.

`page_fault_handler()` may be interrupted, which means there might be two concurrent `page_fault_handler()` calls. You will want to use some synchronization primitives to handle this.

5 Testing

Unlike project3, which could be broken down into several phases for testing, you will need to understand and implement project 5 in its entirety before you can test your work.

This will be frustrating, but Don't Panic! The test processes that are part of the start code and are built into the disk image upon compilation should demonstrate where your problems lie.

To test your project 5, simply make your code, and run with Bochs. *Note: Don't waste time trying to test your work on a real machine using a USB drive.*

6 Design Review

As always, please take a look at what we expect from you in general. This project will have, like the others, a design review worth 5 points. We'll ask about the following things:

1. **Page table + Page Faults (2 points):** Explain how virtual addresses are translated to physical addresses on i386. When are page faults triggered? How are you going to figure out what address a fault occurred on?
2. **Page Map (1 point):** You're going to need a data structure to track information about pages. What information should you track?
3. **Calling Relationships (2 point)** For the functions `page_alloc`, `page_swap_in`, `page_swap_out`, and `page_fault_handler`, please describe the caller-callee relationship graph.

Also, you should read through the whole project description before showing up to design review. As always, come into the review prepared with your answers to all questions written down/typed up.

7 Hints

As mentioned in the section on the page fault handler, the page fault handler can be interrupted. Therefore, you may want to implement your memory manager using a synchronization primitive so that only one process can have its page fault handled at any given time. Keep in mind that there may also be other functions that can be interrupted, so you will want to handle all such cases in the same way.

Which pages are pinned and which aren't? The specs say that each process keeps track of four types of pages: page directories, page tables, stack page tables, and stack pages. These four types of pages can be pinned. But it should be implied that processes also do keep track of pages (corresponding to page table entries). These are the ones that are NOT pinned.

What does "It's OK to map more pages than required" (in page table setup) mean?. This means that you can initialize more page table entries/page directory entries. In part, you will probably want to do this to align a whole page directory/table to a multiple of `PAGE_SIZE`, and in part to simply fill in the whole page (think of this as initializing an array to all zeros). What this doesn't mean is that you're allocating more pages than required.

What's the role of screen memory? Screen memory is part of the kernel's memory space, user processes need to be given permission to read and write the screen when memory is initialized. This is only done once.

What does clearing the page mean? Clearing the page simply means that at the end of allocating a new page, they should clear it in memory (i.e. set the page to all zeros or something like that) and also reset the page table entry flags.

What is the page map? The page map is more of an internal data structure that keeps track of metadata (e.g. swap location, pinned?, virtual addr). This array is of size `PAGEABLE_PAGES` (initialized to 24, but you should eventually increase this value to 30) because it keeps track of all pages currently loaded in memory. Related to this is also what the parameter i represents in certain functions (`page_addr`, `swap_in`, `swap_out`). This i is an index in the page map.

You don't need to worry about code vs. data segments (that's why we have "normal" process pages vs. stack pages).

You don't need to implement the TLB, that's done for you. Think of the TLB as a black box where the interface is `flush_tlb_entry()`.

If you grep TODO, then you can ignore all TODOs that aren't in `memory.[ch]` These are remnants of the original developers' code.

Here is a checklist to help you complete this project. Implement the functions roughly in this order:

1. memory.h
2. `page_map_entry_t`
3. memory.c
4. `page_addr()`
5. `page_replacement_policy()`
6. `page_alloc()`
7. `init_mem()`
8. `setup_page_table()`
9. `page_swap_in()` and `page_swap_out()`
10. `page_fault_handler()`

7.1 Submission

Don't forget to complete the general requirements for each project!

In particular, you need to submit a README file, which concisely describes your design and implementation, and what parts work and what parts you didn't get to work. You don't have to repeat anything you presented in the design review.

8 Grading Rubric

The project is worth 25 points, with four possible bonus points. The breakdown is as follows:

Task	Deliverables	Points
Design review	Correct descriptions of the main concepts of the project and completion of tasks assigned by Nabil.	5
Threads run	Kernel threads running correctly.	6
Directory Setup	Correct process directory set up in setup page table.	3
Swap in	Processes run correctly with a larger number of pageable pages. This requires the proper implementation of swap in.	3
Swap out	Swap out correct – processes run correctly when pageable pages is low.	6
Kernel Protected	Processes unable to write to kernel memory (except for screen addresses).	2
Bonus FIFO (2nd chance)	FIFO with second chance works correctly.	+2
Bonus (NRU)	An implementation of not recently used page replacement algorithm works correctly.	+2

9 Acknowledgments

Once again, we gratefully acknowledge the kindness of the staff of Princeton's COS 318 course for providing us with the material for (and most of the text of) this assignment.

Like always, best wishes and good luck!