601.418/618 (S24): Assignment 3A: Virtual Memory

jhuopsys.github.io/spring2024/assign/assign03a.html

601.418/618 (S24): Assignment 3A: Virtual Memory

Project 3: Virtual Memory - Part A

Due: Friday, April 12th by 11 pm

By now you should have some familiarity with the inner workings of Pintos. Your OS can properly handle multiple threads of execution with proper synchronization, and can load multiple user programs at once. However, the number and size of programs that can run is limited by the machine's main memory size. In this assignment, you will remove that limitation.

You will build this assi[illegible] project 2 should also work with project [illegible] oject 2 submission before you start work [illegible] cause the same problems in project 3.

You will continue to ha[illegible] you did in the previous assignment [illegible]

Background

B.1 Source Files

You will work in the "v[illegible] tains only "`Makefile`"s. The only [illegible] le" turns on the setting "-DVM". All coc[illegible] ed in earlier projects.

You will probably be encountering just a few files for the first time:

"`devices/block.h`"

"`devices/block.c`"
Provides sector-based read and write access to block device. You will use this interface to access the swap partition as a block device.

B.2 Memory Terminology

Careful definitions are needed to keep discussion of virtual memory from being confusing. Thus, we begin by presenting some terminology for memory and storage. Some of these terms should be familiar from project 2 (see section 4.1.4 Virtual Memory Layout), but much of it is new.

B.2.1 Pages

A *page*, sometimes called a *virtual page*, is a continuous region of virtual memory 4,096 bytes (the *page size*) in length. A page must be *page-aligned*, that is, start on a virtual address evenly divisible by the page size. Thus, a 32-bit virtual address can be divided into a 20-bit *page number* and a 12-bit *page offset* (or just *offset*), like this:

```
 31               12 11        0
+-------------------+-----------+
|    Page Number    |   Offset  |
+-------------------+-----------+
         Virtual Address
```

Each process has an independent set of *user (virtual) pages*, which are those pages below virtual address `PHYS_BASE`, typically `0xc0000000` (3 GB). The set of *kernel (virtual) pages*, on the other hand, is global, remaining the same regardless of what thread or process is active. The kernel may access both user and kernel pages, but a user process may access only its o Layout, for more information.

Pintos provides sever es. See section A.6 Virtual Addresses

B.2.2 Frames

A *frame*, sometimes c uous region of physical memory. Like ned. Thus, a 32-bit physical address can t *frame offset* (or just *offset*), like this:

```
 31               12
+-------------------
|    Frame Number
+-------------------
         Physical Address
```

The 80x86 doesn't provide any way to directly access memory at a physical address. Pintos works around this by mapping kernel virtual memory directly to physical memory: the first page of kernel virtual memory is mapped to the first frame of physical memory, the second page to the second frame, and so on. Thus, frames can be accessed through kernel virtual memory.

Pintos provides functions for translating between physical addresses and kernel virtual addresses. See section A.6 Virtual Addresses, for details.

B.2.3 Page Tables

In Pintos, a *page table* is a data structure that the CPU uses to translate a virtual address to a physical address, that is, from a page to a frame. The page table format is dictated by the 80x86 architecture. Pintos provides page table management code in "`pagedir.c`" (see section A.7 Page Table).

The diagram below illustrates the relationship between pages and frames. The virtual address, on the left, consists of a page number and an offset. The page table translates the page number into a frame number, which is combined with the unmodified offset to obtain the physical address, on the right.

```
                          +----------+
          .--------------->|Page Table|---------.
         /                +----------+          |
   31    |   12 11    0                    31    V   12 11    0
  +-----------+-------+                    +-----------+-------+
  |  Page Nr  |  Ofs  |                    |  Frame Nr |  Ofs  |
  +-----------+-------+                    +-----------+-------+
   Virt Addr      |                         Phys Addr      ^
                   \                                      /
```

B.2.4 Swap Slots

A *swap slot* is a contin … partition. Although hardware lim … r than for pages and frames, swap slot … wnside in doing so.

B.3 Resource Man

You will need to desig

Supplemental page t

Enables page fault ha … . See section B.4 Managing the Suppler

Frame table

Allows efficient implementation of eviction policy. See section B.5 Managing the Frame Table.

Swap table

Tracks usage of swap slots. See section B.6 Managing the Swap Table.

You do not necessarily need to implement three completely distinct data structures: it may be convenient to wholly or partially merge related resources into a unified data structure.

For each data structure, you need to determine what information each element should contain. You also need to decide on the data structure's scope, either local (per-process) or global (applying to the whole system), and how many instances are required within its scope.

To simplify your design, you may store these data structures in non-pageable memory. That means that you can be sure that pointers among them will remain valid.

Possible choices of data structures include arrays, lists, bitmaps, and hash tables. An array is often the simplest approach, but a sparsely populated array wastes memory. Lists are also simple, but traversing a long list to find a particular position wastes time. Both arrays and lists can be resized, but lists more efficiently support insertion and deletion in the middle.

Pintos includes a bitmap data structure in "`lib/kernel/bitmap.c`" and "`lib/kernel/bitmap.h`". A bitmap is an array of bits, each of which can be true or false. Bitmaps are typically used to track usage in a set of (identical) resources: if resource n is in use, then bit n of th [illegible] although you could extend their implemen [illegible]

Pintos also includes a [illegible] Table). Pintos hash tables efficiently [illegible] e of table sizes.

Although more comple [illegible] benefits, they may also needlessly c [illegible] recommend implementing any adv [illegible] as part of your design.

B.4 Managing the [illegible]

The *supplemental pag* [illegible] l data about each page. It is needed bec [illegible] 's format. Such a data structure is often [illegible] plemental" to reduce confusion.

The supplemental page table is used for at least two purposes. Most importantly, on a page fault, the kernel looks up the virtual page that faulted in the supplemental page table to find out what data should be there. Second, the kernel consults the supplemental page table when a process terminates, to decide what resources to free.

You may organize the supplemental page table as you wish. There are at least two basic approaches to its organization: in terms of segments or in terms of pages. Optionally, you may use the page table itself as an index to track the members of the supplemental page table. You will have to modify the Pintos page table implementation in "`pagedir.c`" to do so. We recommend this approach for advanced students only. See section A.7.4.2 Page Table Entry Format, for more information.

The most important user of the supplemental page table is the page fault handler. In project 2, a page fault always indicated a bug in the kernel or a user program. In project 3, this is no longer true. Now, a page fault might only indicate that the page must be brought in from a file or swap. You will have to implement a more sophisticated page fault handler to handle these cases. Your page fault handler, which you should implement by modifying `page_fault()` in “`userprog/exception.c`”, needs to do roughly the following:

1. Locate the page that faulted in the supplemental page table. If the memory reference is valid, use the supplemental page table entry to locate the data that goes in the page, which might be in the file system, or in a swap slot, or it might simply be an all-zero page. If you implement sharing, the page's data might even already be in a page frame, but not in the page table.

   If the supplemental page table indicates that the user process should not expect any data at the address it was trying to access, or if the page lies within kernel virtual memory, or if the access is an attempt to write to a read-only page, then the access is invalid. Any invalid access terminates the process and thereby frees all of its resources.

2. Obtain a frame t … rame Table, for details.

   If you implemen … rame, in which case you must b …

3. Fetch the data ir … swap, zeroing it, etc.

   If you implemen … frame, in which case no action is …

4. Point the page t … ysical page. You can use the func …

B.5 Managing the Frame Table

The *frame table* contains one entry for each frame that contains a user page. Each entry in the frame table contains a pointer to the page, if any, that currently occupies it, and other data of your choice. The frame table allows Pintos to efficiently implement an eviction policy, by choosing a page to evict when no frames are free.

The frames used for user pages should be obtained from the "user pool," by calling `palloc_get_page(PAL_USER)`. You must use `PAL_USER` to avoid allocating from the "kernel pool," which could cause some test cases to fail unexpectedly (see Why PAL_USER?). If you modify “`palloc.c`” as part of your frame table implementation, be sure to retain the distinction between the two pools.

The most important operation on the frame table is obtaining an unused frame. This is easy when a frame is free. When none is free, a frame must be made free by evicting some page from its frame.

If no frame can be evicted without allocating a swap slot, but swap is full, panic the kernel. Real OSes apply a wide range of policies to recover from or prevent such situations, but these policies are beyond the scope of this project.

The process of eviction comprises roughly the following steps:

1. Choose a frame to evict, using your page replacement algorithm. The "accessed" and "dirty" bits in the page table, described below, will come in handy.

2. Remove references to the frame from any page table that refers to it.

   Unless you have implemented sharing, only a single page should refer to a frame at any given time.

3. If necessary, wri [illegible]

The evicted frame ma [illegible]

B.5.1 Accessed and [illegible]

80x86 hardware provi [illegible] icement algorithms, through a [illegible] page. On any read or write to a page [illegible] s PTE, and on any write, the CPU sets th [illegible] o 0, but the OS may do so.

You need to be aware [illegible] to the same frame. When an aliase [illegible] are updated in only one page table e [illegible] ccessed and dirty bits for the other alias [illegible]

In Pintos, every user virtual page is aliased to its kernel virtual page. You must manage these aliases somehow. For example, your code could check and update the accessed and dirty bits for both addresses. Alternatively, the kernel could avoid the problem by only accessing user data through the user virtual address.

Other aliases should only arise if you implement sharing for extra credit (see VM Extra Credit), or if there is a bug in your code.

See section A.7.3 Accessed and Dirty Bits, for details of the functions to work with accessed and dirty bits.

B.6 Managing the Swap Table