CS140 Project 3: Virtual Memory

Project Requirements

Goal: Remove current limitation that the number and size of programs running in Pintos is limited by main memory size.

Components:

- Supplemental page table
- Frame table
- Swap table
- Modifying the page fault handler
- Stack growth
- mmap, munmap

The Big Picture

Limited physical memory, but many processes want to use physical memory.

Physical memory isn’t big enough to store every process’s pages all the time.

If a page isn’t needed in physical memory, it gets “paged out” (written to swap table or file system).

When a process needs a page and it’s not in physical memory, it has to get “paged in” (usually “paging out” another page).

Terminology [4.1.2]

- **Page:** contiguous region of virtual memory
- **Frame:** contiguous region of physical memory
- **Page Table:** data structure to translate a virtual address to physical address (page to a frame)
- **Eviction:** removing a page from its frame and potentially writing it to swap table or file system
- **Swap Table:** where evicted pages are written to in the swap partition

Pintos Physical Memory [A.6]

x86 doesn’t provide a way to directly access memory at a physical address.

Solution: Pintos maps kernel virtual memory directly to physical memory.

- Physical memory divided into two pools: user pool and kernel pool.

Virtual address = `PHYS_BASE` < -- > Physical address = `0`

Virtual address = `PHYS_BASE + 0x1234` < -- > Physical address = `0x1234`

Functions: `ptov, vtop, is_user_vaddr, is_kernel_vaddr`

Data structures you'll design

1. **Supplemental page table:** per-process data structure that tracks supplemental data for each page, such as location of data (frame/disk/swap), pointer to corresponding kernel virtual address, active vs. inactive, etc.
2. **Frame table:** global data structure that keeps track of physical frames that are allocated/free.
3. **Swap table:** keeps track of swap slots.
4. **File mapping table:** keeps track of which memory-mapped files are mapped to which pages.

Supplemental Page Table [4.1.4]

Supplements the page table with additional information about each page.

Why not just change page table directly? Limitations on page table format.

Two purposes:

1. On page fault, kernel looks up virtual page in supplemental page table to find what data should be there.
2. When a process terminates, kernel determines what resources to free.

Supplemental Page Table [4.1.4]

What does the page fault handler need to do?

1. Locate faulting page in supplemental page table and check if it's valid.
2. Get a frame to store the page (use the frame table!).
3. Fetch the data into the frame (swap table or file system).
4. Point the page table entry for the faulting virtual address to the physical page (`process_install_page()` or `pagedir_set_page()`)

Frame Table [4.1.5]

Easy to get a frame when not all the frames are full, but how do we get a frame to store a page if all the frames are full?

Solution: Eviction, managed by frame table.

Page replacement algorithm should approximate LRU and perform at least as well as the clock algorithm.

Only manages frames for user pages (`PAL_USER`)

Frame Table [4.1.5]

How to evict a page?

1. Choose a frame to evict, using your page replacement algorithm.
2. Remove page table reference(s) the frame (only multiple references if implementing extra credit).
3. If necessary, write the page to the file system or swap.

Leverage accessed and dirty bits, set by CPU.

Frame Table [4.1.5]

Implement page replacement algorithm using accessed and dirty bits set by CPU.

On any write, hardware sets the dirty bit to 1.

OS can set these back to 0.

Two virtual addresses can refer to the same frame → CPU only updates the accessed/dirty bits of the page used to access

- if you access a page in physical memory via the user virtual address, it won't update the bits for the corresponding kernel virtual address and vice versa.

Swap Table [4.1.6]

Track in-use and free swap slots.

Allow picking an unused swap slot for evicting a page from its frame to the swap partition.

Allow freeing a swap slot when its page read back or process terminated.

File-mapping Table

Track what memory is used by memory mapped files.

Need this to handle page faults in mapped regions and ensure that mapped files don’t overlap any other segments.

Many choices for data structures

1. **Arrays:** simplest approach, sparsely populated array wastes memory
2. **Lists:** pretty simple, traversing a list can take lots of time
3. **Bitmaps:** array of bits each of which can be true or false, track usage in a set of identical resources (`lib/kernel/bitmap.[ch]`)
4. **Hash Tables:** (`lib/kernel/hash.[ch]`)

You don’t need to necessarily implement four completely distinct data structures! Spec splits them into four to make the functionality requirements clear.

Synchronization

Multiple threads will be trying to page in/page out. Make sure your data structures are synchronized!

Stack Growth [4.3.3]

Project 2: pre-alllocate space for user process's stack.

Project 3: dynamically allocate more pages for a process stack as needed.

Valid stack accesses can now cause page faults!

- `PUSH`: 4 bytes below `%esp`
- `PUSHA`: 32 bytes below `%esp`

Allocate new page in page fault handler if valid stack access.

Get `%esp` from `struct intr_frame` passed to `page_fault()`

Memory Mapped Files [4.3.4]

`mmap()` and `munmap()`

Processes may map files into their address space

Memory-mapped pages must be loaded from disk lazily.

`mmap()` will return error status if:

- The size of the file is 0 bytes
- File will overlap with another already mapped page
- `addr` is not page aligned.

When you evict a mmap'd page, write changes back to original file.

All mappins are implicitly unmapped on process exit.

Where to page out/evict to?

Different “types” of pages that can be paged out.

User stack pages → page out to swap

File pages (mmap’d files) → page out to file system

- If it’s dirty, write changes out to the corresponding file.
- If it’s not dirty, simply deallocate because you can reload from the filesystem.

Getting Started

You need to build on top of Project 2. If your Project 2 is buggy, you should fix it before you start!

Getting Started: Suggested Order

1. **Frame table**. Don’t implement swapping yet. You should still pass all project 2 tests.
2. **Supplemental page table and page fault handler** (lazily load code and data segments via page fault handler). You should pass all project 2 functionality tests, but only some robustness tests.
3. **Stack growth, mapped files, page reclamation.**
4. **Eviction** (don’t forget synchronization!)

Getting Started: Tips

Make sure you understand the virtual memory layout and how paging and eviction works *before* you start coding. This assignment is hard to debug, especially if you're confused about what you're trying to implement.

Synchronization shouldn't be an afterthought!

In many people's opinion, this is the most-time consuming project of the class, so start early!