Lab 3: Virtual Memory

CS140

February 13, 2015

Logistics

- Due Friday February 27 at noon
- Probably the hardest assignment so far. Start early (now!)

Overview

- Motivation
- Terminology
- Requirements
- Tips
- Questions

Motivation: What Does VM Solve?

Up until now, a page fault meant game over. Assignment 3 changes that. Now, when you reach `page_fault`, a number of things could happen:

- Did we access an executable? *Bring in the page from disk*
- Did we access a stack page? *Bring in a new, empty page*
- Did we access a memory mapped file? *Read the file contents from disk*

Once you bring in enough pages, you have to evict pages:

- Which frame do we evict? *Use LRU/clock hand*
- How do we evict a frame? *Write its contents to disk if needed*

... and this all needs to happen with synchronization

Terminology

- **Pages**: A contiguous 4,096 byte region of virtual memory.
- **Frames**: A similar region of *physical* memory.
- **Swap Slots**: An *on-disk* slot containing an evicted frame.
- **Page Table**: A data structure for mapping virtual to physical addresses.
- **Supplemental Page Table**: Tracks additional information for each page for page faults.
- **Frame Table**: A data structure tracking frame usage to help with eviction.
- **Swap Table**: Tracks free and used slots in the swap partition.

Requirements

Your assignment should implement the following (most likely in the same order):

1. Implement the Frame Table
2. Implement the Supplemental Page Table
3. Implement Stack Growth
4. Implement Memory Mapped Files
5. Free Resources on Exit
6. Manage Swap
7. Implement Eviction
8. Manage User Memory
9. Manage Page Aliasing
10. Rejoice!

1. Implementing the Frame Table

Goal: Provide programs with free frames when needed

Physical memory is limited, so the size of your frame table is naturally limited. Until you run out of physical pages, you can simply call `palloc_get_page (PAL_USER)`. Once that returns `NULL`, you need to evict a frame using your LRU algorithm. Don't worry about handling that for now.

Tip: The number of frames is fixed because the size of physical memory is fixed. This should simplify the design of your data structure.

2. Implementing the Supplemental Page Table

Goal #1: Load missing data in `page_fault`

1. Find the entry in the supplemental page table
2. Make sure that the user program owns this address. If it doesn't terminate the process.
3. Obtain a free frame from your allocator
4. Fill page with data from swap/filesystem/all `0`s
5. Use `pagedir_set_page` to install the page in the page directory

Goal #2: Decide what to free on `exit`

The kernel should free all memory associated with a process after it terminates. You should close any open files and unmap `mmap`'ed files.

A Note on Synchronization

Synchronization & Parallelism:

There is only one requirement for parallelism, and the frame table and supplemental page table do the heavy lifting. From the writeup (emphasis mine):

> *If one page fault requires I/O, in the meantime processes that do not fault should continue executing and other page faults that do not require I/O should be able to complete.* ***This will require some synchronization effort.***

In other words, you cannot hold a lock on the frame table/supplemental page table/any shared data while reading a file from disk. There are less obvious interactions like evicting frames, freeing resources etc. that all need synchronization.

3. Implement Stack Growth

The user program stack can grow if needed. Stack pages should be loaded lazily (ie. only on a page fault) and written to swap when evicted.

Problem: How do we know what is a stack access?

```
int main(const int argc, const char* argv[]) {
  int foo[2048];
  printf("Check out this int: %d!", foo[2047]);
  printf("And this one isn't even mine: %d!", foo[2049]);
  return 0;
}
```

Solution: We don't. Use simple heuristics instead to validate stack accesses. For example, most GNU/Linux systems impose a 8 MB limit.

4. Implement Memory Mapped Files

Goal: Let the user map and umap files into memory using:

```
mapid_t mmap (int fd, void *addr) &
void munmap (mapid_t mapping)
```

Recall that these are useful for allocating larger chunks of memory that will be freed eventually. This turns out to be fairly straightforward, although there are some complicating requirements:

- You must load mapped pages lazily.
- The file size may not be a multiple of `PGSIZE`. In that case, you should ignore the remaining bytes (ie. don't write them to the file)
- If the file gets evicted, write all modified pages to disk.
- When a process exits, you must unmap all mapped files.

Tip: This is fairly similar to what you have already implemented for loading executables.

5. Free Resources on `exit`

Goal: When a process exits, free its resources

You should already be closing all files, allowing writes to the executable etc. on process exit. A process could own arbitrary amounts of memory when it exits and your kernel responsible for freeing them. Keep in mind that you don't need to synchronize data that is owned exclusively by this thread, should make your life easier.

You probably want to:

1. Unmap all memory mapped files
2. Destroy internal data structures (e.g. supplemental page table)
3. Free any pages you own using `palloc_free_page`
4. Clean up your frame table.

6. Manage Swap

Goal: Efficiently store and retrieve dirty evicted pages

What is swap? A place where dirty pages go after being evicted. A small partition of the filesystem.

Swap helps us store dirty pages that can't live anywhere else:

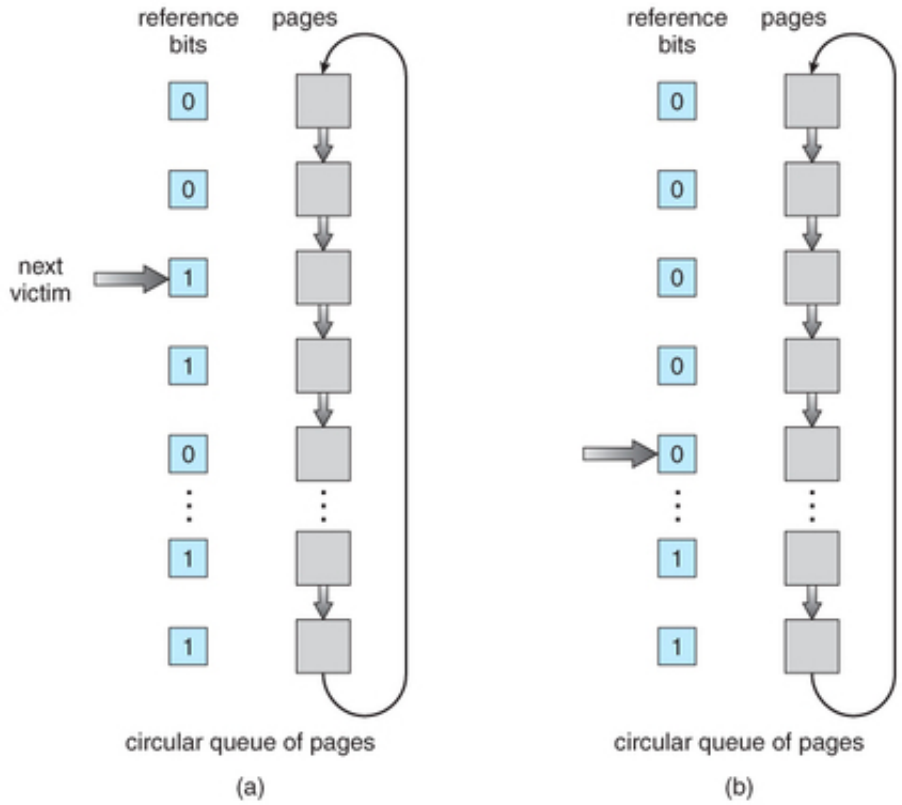
- Did we evict a clean page? *Just read it again when we need it.*
- Did we evict a dirty file system page? *Write it back to the filesystem.*
- Did we evict a dirty stack page? *Swap to the rescue!*

Note: Up until this point you never needed swap since you never evict pages.

Tip: Start simple. Evict a random page instead of doing something fancy to make sure swap works.

7. Implement Eviction

Goal: Implement a global page replacement algorithm that approximates LRU.



Intuitively: You skip frames that have an accessed bit set to `1` and zero their accessed bit. You evict the first frame that has a `0` accessed bit. This frame has not been used since your clock hand last visited it, so all other pages must have been used more recently.

7. Implement Eviction

Algorithm:

- Point your clock hand to some frame on initialization.
- When evicting, sweep through frames starting at clock hand:
  - *If frame is not in use, we are done.*
  - *If frame is pinned, we can't evict it.*
  - *If frame was accessed (since last sweep) give it a second chance.*
  - *Otherwise, evict the frame.*
  - *If we did not evict a page, advance the clock hand.*

Eviction should be **lazy**. You will only need one page at a time, and you should only evict when you know you will need a page.

Recall: Evicting a frame means potentially writing its contents to swap or filesystem. Other processes should not have to wait for this to complete.

7. Implement Eviction

What Happens to Evicted Pages?

When evicting a page:

- Make sure no page points to this frame
- Write the page to swap or the filesystem if needed

8. Accessing User Memory

Goal: Make the kernel resilient to page faults. Again.

In Lab 2, you made sure that your kernel's system calls either: 1. Never took a page fault **or** 2. Could recover gracefully from them.

Now, page faults are inevitable when accessing user memory. You should modify the kernel so you can bring in pages from files or swap in syscalls.

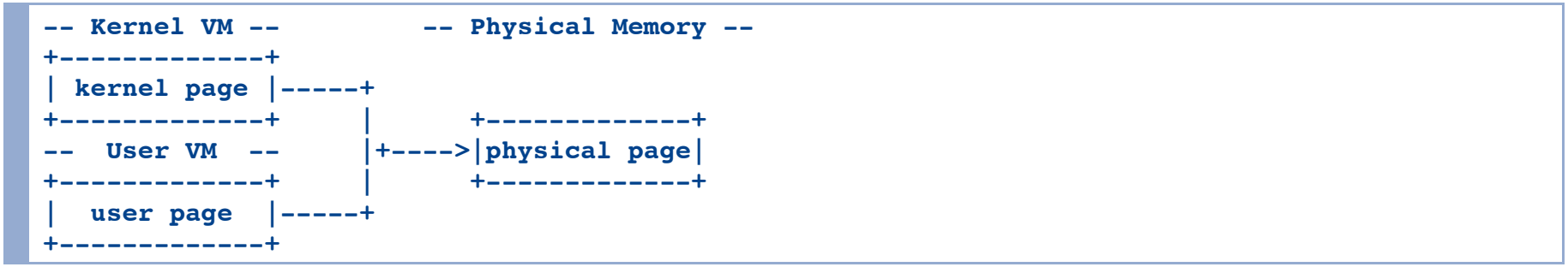
Note: There are some page faults you cannot recover from. For instance, Pintos device drivers acquire locks when reading from the file system. If you take a page fault in `file_read`, you will deadlock.

Solution: Pinning. "Pin" frames in your frame table by making sure they can't be evicted. This restricts the amount of memory available, so you should pin them only when needed.

9. Manage Page Aliasing

Goal: Prevent unfair evictions of aliased pages

Recall: `pagedir_get_page` gives you a kernel virtual memory address for a physical page given a user virtual memory address:



When you access these pages through a virtual address, the dirty/accessed bit will only be set in the page table that owns the address. When evicting pages, you need to consider the access bits on **both pages**. Otherwise, you could evict a frame that the user accessed recently solely because the kernel didn't access it.

Solution: Either check both locations or make sure you only read/write one of them.

10. Rejoice!

Tips: Data Structures

Previous projects were mostly about implementing algorithms and implicit data structures (e.g. the waiting-for graph in Lab 1). This project requires managing and/or implementing several data structures. Pintos gives you implementations of lists, hash tables, and bitmaps and you can easily implement an array.

Important: Use these and don't waste your time implementing anything more advanced. It's not worth it.

Keep the basic tradeoffs in mind:

- **Size**: fixed vs dynamic. Bitmaps and arrays are easy when size is fixed.
- **Access**: random vs linear. Do you need iteration or fast lookups?
- **Ownership**: private vs global. Is the data owned by a single process or global?
- **Synchronization**: coarse vs fine. Should you lock everything or only each element/bucket/bit?

Tips: General

- **Start Early**: This is the hardest project yet, possibly all quarter.
- **Leverage Data Structures**: Don't implement fancy data structures, use provided `list`, `hash`, or `bitmap`
- **Leverage Existing Code**: For example, `vaddr.h` contains methods like `pg_ofs` and `pg_no`. The quarter is too short to get those wrong.
- **Careful Synchronization**: `exit`/`wait` was harder to grasp. Here, the difficult part is deciding on granularity and finding all entry points.
- **Be Lazy**: *All* allocations for this assignment are done lazily. You only ever evict one page, read one page, etc.
- **Start Early**: Seriously, it's a long weekend.

Questions