Pintos lab 3

Instructor: Youngjin Kwon

Big picture: Running a user program

- 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)

Things to do

- Supplemental page table
- Physical frame management
- Modifying the page fault handler for lazy loading (demand paging)
    - Stack growth, file-maped etc
- mmap, munmap
- Swap in/out

Terminology

- 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

Data structures you must 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.

Many choices for data structures

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

Concept of lazy loading

- When a virtual address is created (mmap), pintos associates a struct page to the virtual address

- Each virtual address is used for different purposes; anonymous memory, file-backed memory
  - A struct page reflects the information (going over struct page)
  - Allocating struct page does not mean a physical frame is allocated to the virtual address. The actual physical is allocated in ( )

```
static void
page_fault (struct intr_frame *f) {
  bool not_present;  /* True: not-present page, false: writing r/o page.
  bool write;        /* True: access was write, false: access was read.
  bool user;         /* True: access by user, false: access by kernel. *
  void *fault_addr;  /* Fault address. */

  /* Obtain faulting address, the virtual address that was
     accessed to cause the fault.  It may point to code or to
     data.  It is not necessarily the address of the instruction
     that caused the fault (that's f->rip). */

  fault_addr = (void *) rcr2();

  /* Turn interrupts back on (they were only off so that we could
     be assured of reading CR2 before it changed). */
  intr_enable ();


  /* Determine cause. */
  not_present = (f->error_code & PF_P) == 0;
  write = (f->error_code & PF_W) != 0;
  user = (f->error_code & PF_U) != 0;

#ifdef VM
  /* For project 3 and later. */
  if (vm_try_handle_fault (f, fault_addr, user, write, not_present))
    return;
#endif

  /* Count page faults. */
  page_fault_cnt++;

  /* If the fault is true fault, show info and exit. */
  printf ("Page fault at %p: %s error %s page in %s context.\n",
      fault_addr,
      not_present ? "not present" : "rights violation",
      write ? "writing" : "reading",
      user ? "user" : "kernel");
  kill (f);
}
```

Where to start?

```
/* Return true on success */
bool
vm_try_handle_fault (struct intr_frame *f UNUSED, void *addr UNUSED,
    bool user UNUSED, bool write UNUSED, bool not_present UNUSED) {
  struct supplemental_page_table *spt UNUSED = &thread_current ()->spt;
  struct page *page = NULL;
  /* TODO: Validate the fault */

  /* TODO: Your code goes here */

  // stack growth on demand (calling vm_stack_growth)

  return vm_do_claim_page (page);
}
```

```
/* Claim the PAGE and set up the mmu. */
static bool
vm_do_claim_page (struct page *page) {
  struct frame *frame = vm_get_frame ();

  /* Set links */
  frame->page = page;
  page->frame = frame;

  /* TODO: Insert page table entry to map page's VA to frame's PA. */
  // pml4_set_page

  return swap_in (page, frame->kva);
}
```

Struct page and page_operations

- How to fill the contents of a frame after it is allocated?
  - A page initially starts with uninit_page, then
  - A page for anonymous memory → anon_initializer
  - A page for file-backed memory → file_map_initializer
  - What function associate initializer according to each page's type?

```
struct page_operations {
  bool (*swap_in) (struct page *, void *);
  bool (*swap_out) (struct page *);
  void (*destroy) (struct page *);
  enum vm_type type;
};
```

Supplemental Page Table

- Supplements the page table with additional information about each page. Why not just change page table directly? Limitations on page table format.
  - spt_find_page: find struct page from spt and virtual address

- Two purposes:

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

Frame Table

- 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.
  - vm_get_victim
- Only manages frames for user pages (PAL_USER)

Frame Table

- 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.

Swap Table

- 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

Beware synchronization

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

Stack Growth

- Project 2: pre-allocate 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!

- Allocate new page in page fault handler if valid stack access.
- Get %rsp from struct intr_frame passed to page_fault()

Memory Mapped Files

- 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 mappings 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.

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!)