CS 5600
Computer Systems

Project 3: Virtual Memory in Pintos

Virtual Memory in Pintos

- Pintos already implements a basic virtual memory system
  - Can create and manage x86 page tables
  - Functions for translating virtual addresses into physical addresses
- But this system has limitations
  - No support for swapping pages to disk
  - No support for stack growth
  - No support for memory mapping files

Your Goals

1. Implement page swapping
   - If memory is full, take a page from physical memory and write it to disk
   - Keep track of which pages have been moved to disk
   - Reload pages from disk as necessary
2. Implement a frame table
   - Once memory becomes full, which pages should be evicted?
3. Implement a swap table
   - Maps pages evicted from memory to blocks on disk

Your Goals (cont.)

4. Implement stack growth
   - In project 2, the stack was limited to one page
   - Allow the stack to grow dynamically
5. Implement mmap() and munmap()
   - i.e. the ability to memory map files
   - Create a table that keeps track of which files are mapped to which pages in each process

What Pintos Does For You

- Basic virtual memory management
  - User processes live in virtual memory, cannot access the kernel directly
  - Kernel may access all memory
  - Functions to create and query x68 page tables
- Trivial filesystem implementation
  - You can read and write data to disk
  - Thus, you can read and write memory pages

Utilities

- threads/pte.h
  - Functions and macros for working with 32-bit x86 Page Table Entries (PTE)
- threads/vaddr.h
  - Functions and macros for working with virtualized addresses
  - Higher-level functionality than pte.h
  - Useful for converting user space pointers into kernel space
- userprog/pagedir.c
  - Implementation of x86 page tables

- Page fault handler: userprog/exception.c

```
static void page_fault (struct intr_frame *f) {
   bool not_present,  write, user;
   void *fault_addr;  /* Fault address. */

   asm ("movl %%cr2, %0" : "=r" (fault_addr));  /* Obtain faulting address*/
   intr_enable ();
   page_fault_cnt++; /* Count page faults. */

   /* Determine cause. */
   not_present = (f->error_code & PF_P) == 0; /* True: not-present page,
                                                 false: writing r/o page. */
   write = (f->error_code & PF_W) != 0;        /* True: access was write,
                                                 false: access was read. */
   user = (f->error_code & PF_U) != 0;         /* True: access by user,
                                                 false: access by kernel. */

   /* Code for handling swapped pages goes here! */

   printf ("Page fault at %p: %s error %s page in %s context.\n", ...);
   kill (f);
}
```

Supplementary Page Tables

- The format of the page table is defined by the x86 standard
  - You can’t modify or add to it
- Thus, you will need to define additional data structures
  - Supplementary page tables
  - Keep track of info for eviction policy, mapping from swapped memory pages to disk, locations of memory mapped files, etc.

Project 3 Is Open Ended

- The previous projects were about you extending the functionality of Pintos
- In this, you are free to implement things however you wish
  - pintos/src/vm/ is basically empty

Key Challenges

- Choosing the right data structures
  - Time and memory efficiency are critical
  - Hash tables? Lists? Bitmaps?
  - You don’t need to implement more exotic data structures (e.g. red-black trees)
- Handling page faults
  - All swapping is triggered by page faults
  - Handling them, and restarting the faulting instruction, are critical

More Key Challenges

- Implementing eviction
  - How do you choose which page to evict?
- Detecting stack growth
  - You will need to develop heuristics to determine when a process wants to grow the stack
- Managing concurrency
  - Pages can be evicted at any time
  - What happens if the kernel or a process is accessing them?

Extra Credit Challenge!

- Implementing Sharing
  - What happens if a program is run >1 time?
    - You could share the code pages
  - What happens if >1 process mmap()s the same file?
- Worth an additional two points
  - So 17 out of 15

Things Not To Worry About

- Your supplementary data structures may live in kernel memory
  - i.e. they will never get swapped to disk
  - In a real OS, page tables may be swapped to disk

Modified Files

- Makefile.build 4
- threads/init.c 5
- threads/interrupt.c 2
- threads/thread.c 31
- threads/thread.h 37
- userprog/exception.c 12
- userprog/pagedir.c 10
- userprog/process.c 319
- userprog/syscall.c 545
- userprog/syscall.h 1
- vm/<new files> 628
- 11+ files changed, 1594 insertions, 104 deletions

Add new files

Initialize supplementary tables for the system and per thread

Modified page fault handler

Support for mmap() syscall

Swapping implementation

Grading

- 15 (+2) points total
- To receive full credit:
  - Turn in working, well documented code that compiles successfully and completes all tests (50%)
  - Turn in a complete, well thought our design document (50%)
- If your code doesn't compile or doesn't run, you get zero credit
  - Must run on the CCIS Linux machines!
- All code will be scanned by plagiarism detection software