Pintos project 3: Virtual Memory Management

Presented by

Xiaomo Liu

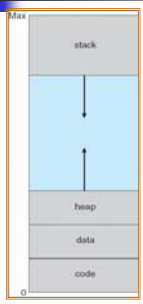
Acknowledgement:

The slides are based on Yi Ma's presentation

Outline

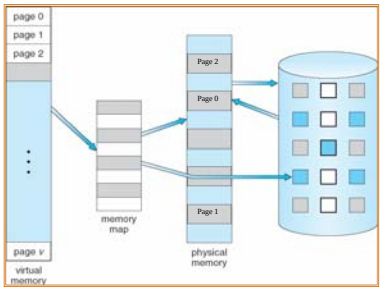
- Virtual Memory Concept
- Current Pintos memory management
- Requirement
  - Lazy load
  - Stack growth
  - File – memory mapping
  - Swapping
- Suggestions
  - How to start
  - Implementation order

Virtual Memory

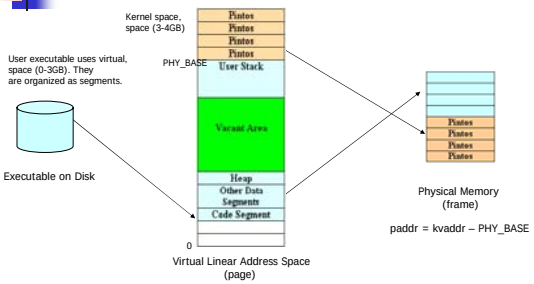


- Logical memory layout for every process
- Mapping to the real physical memory
- What to do it? Paging!
- Divide the process into small pieces (pages)– 4KB

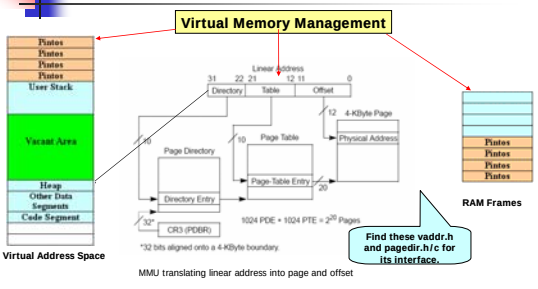
Virtual Memory



Pintos virtual memory



Pintos Virtual Memory



Current Status (before the project 3)

- Support multiprogramming
- Page directory including its page tables for each process
- Load the entire data and code segment and stack into the memory before the execution (see load function in process.c).
- Fixed stack (1 page) to each process

7

Requirement Overview

- Lazy load
  - Don't load any page initially. Load a page from executable when it is needed.
- Stack growth
  - Allocate additional pages for stack as necessary.
- File – memory mapping
  - Keep one copy of opened files in the memory. Keep track which memory maps to which file.
- Swapping
  - Run out of frames, select a used frame and swap it out to the swap disk. Return it as a free frame.

8

Step 1: Frame table management

- You need a frame table that keeps track all the frames of physical memory used by the user processes.
- Two approaches:
  - (1) Modify current allocator "palloc_get_page(PAL_USER)"
  - (2) Implement your own allocator on top of "palloc_get_page(PAL_USER)" without modifying it. (Recommended)
  - Have a look at "init.c, palloc.c" to get some ideas
- Frame table is necessary for swapping

9

Step 2: Lazy Load

- How to load executables?
  - Allocate a frame and load a page of executable into memory
- Before project 3: pintos will load all pages of executables into the physical memory.
- After project 3:
  - Load nothing except setup the stack at the beginning
  - When executing the process, a page fault occurs and the page fault handler checks where the expected page is: in executable? in swap disk?
  - If in executable, you need to load the corresponding page from executable
  - If in swap disk, you need to load the corresponding page from swap disk
  - Page fault handler need to resume the execution of the process after lazy load

10

Lazy load: supplemental page table

- Functionalities
  - Your "s - page table" must be able to decide where to load executable and which corresponding page of executable
  - Your "s - page table " must be able to decide how to get swap disk and which sectors of swap disk stores the corresponding page
- Used by page fault handler
- Populated in load_segment() and mmap() system call

11

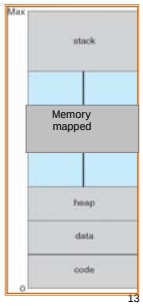
Step 3: Stack Growth

- Functionality
  - Before project 3: user stack is fixed size of 1 page, i.e. 4 KB
  - After project 3: user stack is allow to grow
  - Allocate additional pages for user stack as necessary
- Implementation
  - If the user program exceeds the stack size, a page fault will happen
  - In page fault handler, you need to distinguish stack accesses from other accesses (It is a stack access if the fault address is greater or equal to esp – 32)
  - Catch the stack pointer—esp of interrupt frame
  - You can impose a absolute limit on stack size, STACK_SIZE

12

Step 4: Memory mapped files

- Functionality
  - Keep a copy of an open file in memory
  - Keep it in user space



13

Step 4: Memory mapped files

- If file size is not multiple of PGSIZE—stick-out, cause partial page
- Don't map when: zero address or length, overlap, or console
- Implementation
  - Use the "fd" to keep track of the open files of a process
  - Design two system calls: mapid_t mmap(fd, addr) and void munmap(mapid_t)
  - Design a data structure to keep track of these mappings
  - We don't require that two processes that map the same file see the same data

14

Step 5: Swapping

- Functionality
  - When no more free frame, evict a page from its frame and put a copy of into swap disk, if necessary, to get a free frame — swap out
  - When page fault handler finds a page is not memory but in swap disk, allocate a new frame and move it to memory —swap in
- Implementation
  - Need a method to keep track of which pages have been swapped and in which part of swap disk a page has been stored

15

Swapping: evict a frame

- Choose a suitable page replacement algorithms, such as second chance algorithm, additional reference bit algorithm. (See 9.4 of textbook)
- Select a frame to swap out from frame table
  - Use accessed/dirty bit in PTE
- Send it to swap
  - Prevent change to page during swapping
- Update PD & PT

16

Swapping: frame table

- The main purpose of maintaining frame table is to efficiently evict a frame for swapping
- Evict a frame usually need to access the "Accessed" & "Dirty" bits of the page table entry of this frame
- Remember this fact! It is very important to design your data structure of the frame table and its entry.
- Because you need to somehow refer frame table entry back to the page table entry (PTE) so as to get the "Accessed" & "Dirty" bits.

17

Swapping: swap space management

- You must be able to keep track of which swap slots have been used and which are not.
  - A page is 4KB
  - The Swap disk has sectors of 512B. (see disk.c/h)

18

Step 6: On process termination

- Destroy your supplemental page table
- Free your frames
- Free your swap slots
- Close all files. It means write back the dirty mmap pages

19

Important issues

- Access user data
  - In project 2, need only verify user address.
  - In project 3, need handle actual access to the content of user memory: (must prevent) process B from evicting a page belonging to process A if A accesses this page during a system call.
  - Need protections:
    - check address+ lock frame
    - read/write
    - unlock.

20

Important issues

- Synchronization
  - Allow parallelism of multiple processes.
  - Page fault handling from multiple processes must be possible in parallel.
  - E.g., A's page fault need I/O (swap, lazy load); B's page fault need not (stack growth, all '0' page), then B should go ahead.

21

Important issues

- Data structure
  - Proper data structure will affect your design
  - Bit map, hash, list, and array
  - How many copies
  - Make it simple

22

Design milestone

- Decide the data structures
  - Data structure for supplemental page table entry, frame table entry, swap table entry
  - Data structure for the tables, Hash table? Array? List? Bitmap?
  - Should your tables be global or per-process?
- Decide the operations for your data structures
  - How to populate the entries of your data structures
  - How to access the entries of your data structures
  - How many entries your data structure should have
  - When & how to free or destroy your data structure
- Deadline
  - Oct 26th 11:59pm
  - No extra days

23

Suggested Order

- Pre-study
  - Understand memory & virtual memory (Lecture slides and Ch 8 & 9 of your textbook)
  - Understand the project documentation (including Appendix A: virtual address & hash table)
  - Understand the important source codes (load() in process.c and pagedir.h)
  - Submit your design milestone
- Fix the bugs of project 2 and make it pass all the test cases
- Frame table management:
  - Implement your frame table allocator.

24

Suggested Order

- Supplemental page table management
  - Modifying load() function in process.c or designing a new one to populate your supplemental page table
  - Modify the page fault handler to implement the lazy load, i.e. load a page when page fault occurs
- Run the regression test cases from project2
  - Your kernel with lazy load should pass all the test cases at this point
- Implement stack growth, memory mapping in parallel
- Swapping
  - Construct your data structure for swap slots management
  - Implement the page replacement algorithm
  - Implement “swap out” functionality
  - Implement “swap in” functionality

25

Other suggestions

- Working in the VM directory
  - Create your page.h, frame.h, swap.h, as well as page.c, frame.c, swap.c in the /VM directory
  - Add your additional files to the makefile: Makefile.build
- Keep an eye on project forum

26

End

- Question?
- Good luck!

27