Project 3: Virtual Memory

Winter 2024

Plan

- Review of virtual memory
- Project requirements and components
  - Supplemental page table
  - Swap table
  - Frame table
  - File mapping table
- Getting started

Virtual Memory

Virtual Memory Review: Terminology

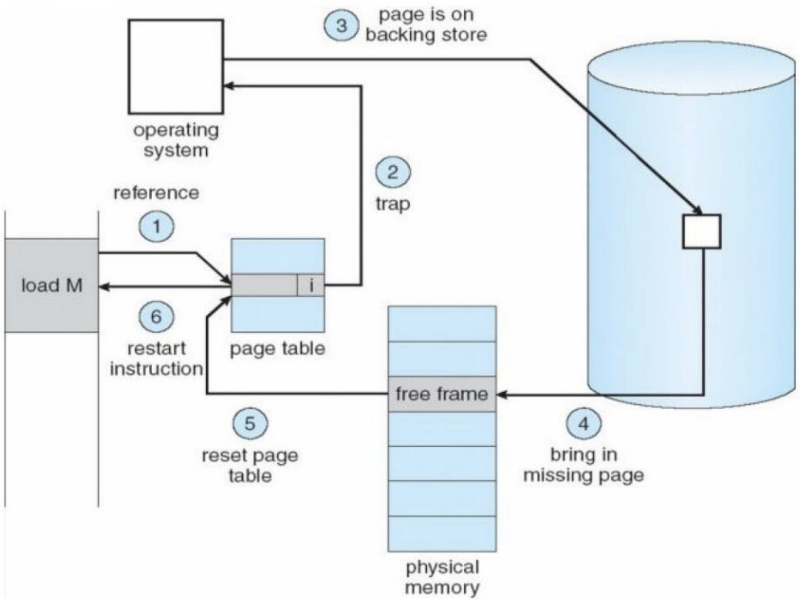
- **Page (virtual page):** a contiguous, fixed-size region of virtual memory
  - Each process has its own set of **user pages**
  - The kernel has **global pages** that are active no matter which process is running
- **Frame (physical page):** a contiguous , fixed-size region of physical memory
  - In Pintos, frames are mapped directly to kernel pages (so to access frames, one would use kernel pages - see A.6)
- **Page Table:** a mapping to convert virtual addresses to physical ones (translate page number to frame number)
- **Swap Slot:** A page sized region of disk space

Virtual Memory Review: Purpose

- Pretend you have access to much more physical memory than you actually do
- **Goal: achieve the speed of memory with the size of disk**
    - Done by **paging in** from disk and **evicting** pages from memory

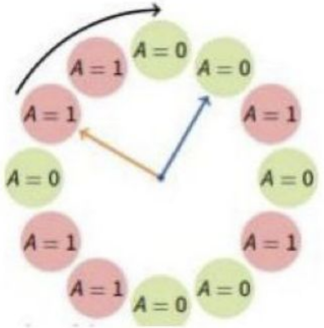
Virtual Memory Review: Paging

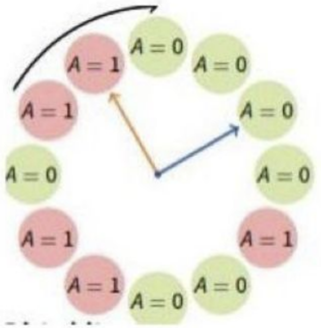
- Used to bring in a page from disk into memory when necessary

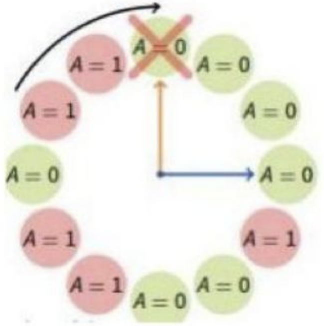


Virtual Memory Review: Eviction

- Paging requires a free frame; how do we choose which page to evict in order to free a frame?
- We want to evict the Least Recently Used (LRU) page, which we approximate with the clock algorithm
  - Have two hands moving in lockstep, one of which marks a page as unaccessed and another evicting the first unaccessed page it finds
  - For smaller memory, you can use a single clock hand







Project Requirements

Project Requirements: Summary

- In project 2, user programs were limited by the size of main memory
- We plan to remove this limitation by implementing paging and virtual memory

Project Requirements: Memory

- Physical memory is divided into 2 pools:
  - User pool - palloc_get_page(PAL_USER)
  - Kernel pool - palloc_get_page(0)
- Access a physical memory address by:
  - PHYS_BASE + phys_address
- CPU sets accessed bit = 1 on page read, and dirty bit = 1 on page write
  - OS can set bits back to 0, CPU cannot

Project Requirements: Data Structures

- Need to implement four data structures in project 3:
    - Supplemental page table
    - Frame table
    - Swap table
    - File mappings table
- Can wholly/partially merge these data structures as you see fit
- Make sure that these data structures cannot be evicted

Project Requirements: Supplemental page table

- Used to provide the page table with additional information about every page
  - Used when handling page faults
  - Used to decide which resources to free when a process terminates

Project Requirements: Frame Table

- Stores mapping between frames and the user page occupying the frame + any other information that might be needed
- Used for obtaining new frames
  - If the frame table is not full, use palloc_get_page to get a free frame; otherwise, use the frame table to evict a page

Project Requirements: Swap Table

- Tracks in-use and free swap slots
- Used for eviction (when a page is evicted it is placed in swap) and paging in (when a page is read back into memory the swap table should mark the slot as free)

Project Requirements: File Mapping Table

- Track which pages are used by each memory mapped file
  - Used by mmap() and munmap
- Stores the pages in use by a memory mapped file

Mechanisms

Project Requirements: Paging In

- A page fault may be caused by a dereference of an address which is not in memory
- To page in the necessary page, modify the page_fault handler to do the following:

1) Locate the page that faulted in the SPT
2) If the reference is valid, use SPT to locate the page's data. Could be in the filesystem, in a swap slot, or be all-zero page. If reference is invalid, kill the process
3) Obtain a free frame to store the page
4) Fetch the data into the free frame
5) Update the page table entry to point the virtual address to the new physical address

Project Requirements: Eviction

- If no frames are available for a page during paging in, then a page must be evicted from its frame
- You must implement an algorithm for eviction at least as good as the clock algorithm

1) Choose a page to evict using the algorithm
2) Remove references to the frame from any page table entry that refers to it
3) If needed, write the evicted page to file system or swap

Project Requirements: Stack Growth

- In project 2 the stack was limited to a single page - now allocate a new stack page if the stack grows beyond its current page (seen by a page fault from a stack access)
- Only allocate a new stack page if an access appears to be a stack access by devising a heuristic to tell if an access is a stack access
- Ensure stack pages can be evicted
- Impose an absolute limit on stack size for a process

Project Requirements: Memory Mapped Files

- Allows you to use demand paging on the data of a file
    - Has a one-one correspondence between the contents of a file and main memory
- You will need to implement the following functions:
    - mapid_t mmap (int fd, void *addr)
        - Maps file open at fd into consecutive virtual pages starting at addr.
        - Lazily load file data into pages when accesses occur.
        - When page is evicted, load data back into file.
    - void munmap (mapid_t mapping)
        - Unmaps mapping designated by mapid_t returned by prior call to mmap.
        - All mappings remain until process exits or munmap is called

Project Requirements: System Calls

- A page may be evicted from its frame while it is being accessed by kernel code
    - E.g you might evict a buffer from a userpage that a system call is relying on and is halfway through executing
- You must either make your kernel handle such page faults, or prevent them from occurring in the first place
    - You can implement **pinning** (marking certain pages as unevictable) so that pages that are currently in use by the kernel are not able to be evicted

Getting Started

Order of Implementation

1) Fix all remaining project 2 bugs (you must build project 3 on top of project 2)

2) Implement the frame table without eviction

    a. should still pass project 2 tests at this point

3) Implement the Supplemental Page Table and Paging

    a. Should still pass project 2 tests and some of the robustness tests

4) Implement stack growth, memory mapped files and freeing pages on process exit

    a. Can be done in parallel

5) Implement eviction

Tips

- Start early! Many students consider this the hardest assignment of the course
- Carefully consider your data structures and their uses before writing code
- Think about how you will handle synchronisation to prevent deadlock
- Add new files to the vm directory to make your code cleaner (you can see what new files were introduced in the reference solution)