Project 3: Virtual Memory

Winter 2023

Section Outline

- Review of paging
- Project Background
- Project Requirements
    - Data structures to implement
    - Paging
    - Stack growth
    - Memory Mapped Files
    - Accessing User Memory in System Calls
- Advice for getting started

Review of Paging

Terminology

- **Page (Virtual Page)**: A contiguous region of virtual memory
    - Each process has its own set of **user pages**
    - The kernel has **global pages** that are active no matter which thread or process is running
- **Frame (Physical Page)**: A contiguous region of physical memory
    - In Pintos, frames are mapped directly to kernel pages (so to access frames, one would use kernel pages)
- **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

Paging Basics

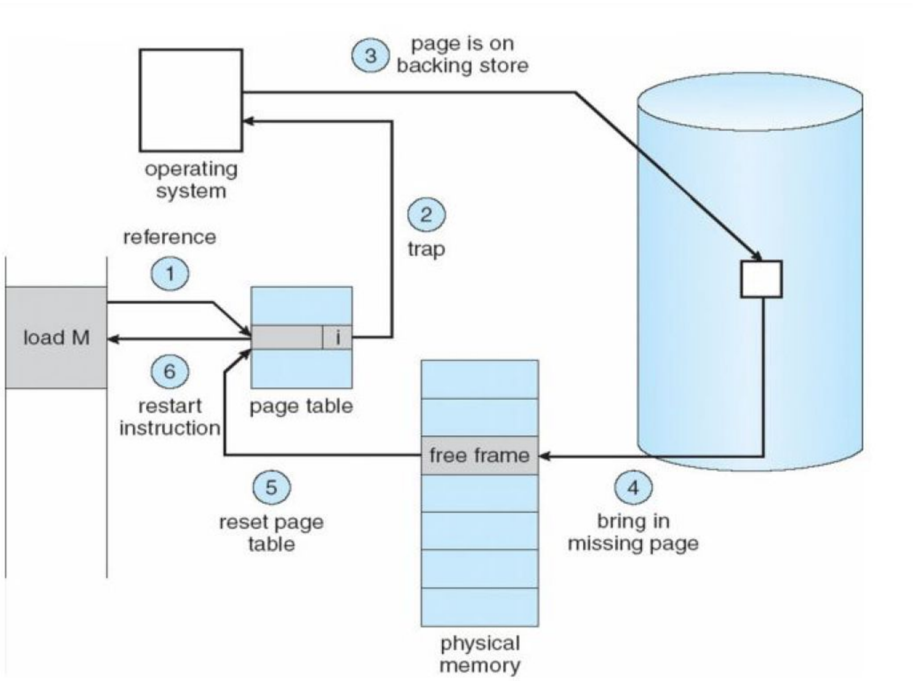
- Goal: Use disk to simulate virtual memory that is larger than available physical memory
    - Achieve the speed of memory with the size of disk
    - Done with 80/20 rule: keeping “hot” 20% in memory and “cold” 80% on disk
- Requires implementing **paging in** and **eviction**

Paging In

Used to bring in a page in disk to memory when an address on the page is accessed and causes a page fault because page is not in memory

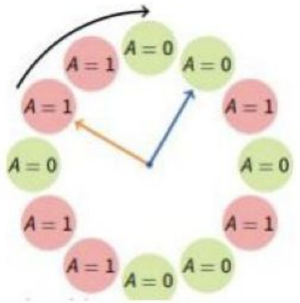
Upon page fault, the following steps are taken:

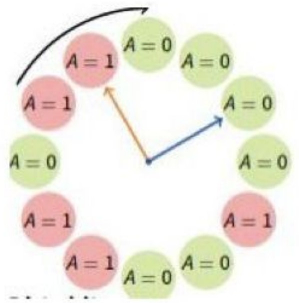
1. Locate page data is on using page tables (1)
2. Determine if access is valid (2 + 3)
3. Load page from disk into memory (4 + 5))
4. Return control to prior instructions

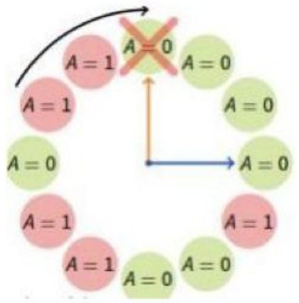


Eviction

- When physical memory is full and a page needs to be brought in, some other page must be **evicted**
- Strategy: Evict the Least Recently Used (LRU) page, assuming that past predicts future
- Implement the clock algorithm:
  - Pages in a circular FIFO list, with one clock hand clearing accessed bit and another evicting if A = 0
  - For a smaller memory, can implement using a singular clock hand





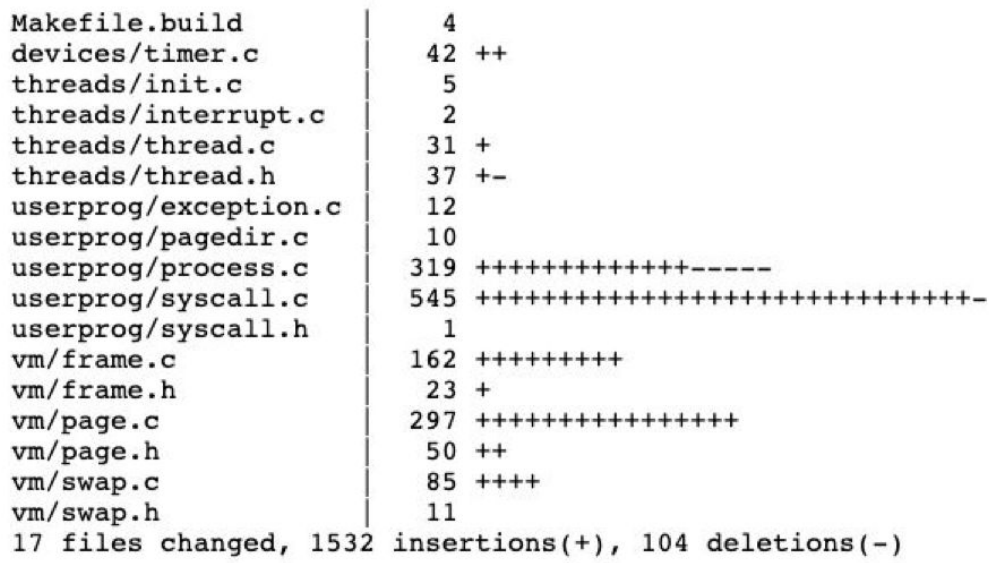


Project Background

Motivation

- The user program functionality implemented in project 2 is limited by machine's main memory size
- In this project, this limitation is removed by implementing memory virtualization and paging

Reference Implementation



```
Makefile.build           |    4
devices/timer.c          |   42 ++
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/frame.c               |  162 +++++++++
vm/frame.h               |   23 +
vm/page.c                |  297 ++++++++++++++++
vm/page.h                |   50 ++
vm/swap.c                |   85 ++++
vm/swap.h                |   11
17 files changed, 1532 insertions(+), 104 deletions(-)
```

Memory in Pintos

- 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

Aliases!

- In Pintos, every user virtual page (upage) is aliased with a kernel virtual page(kpage)
  - i.e. every frame can be accessed from a user virtual address and a kernel virtual address

- Accessing a frame will only set the accessed and dirty bits for the PTE of the virtual page used to access the frame
  - In order to check if a frame has been accessed, you either need to check both PTE’s, or only access frames through the user virtual page (and thus only check the upage PTE)

Project Requirements

Data Structures

Data Structures

- Need to implement 4 data structures
  - Supplemental Page Table
  - Frame Table
  - Swap Table
  - File Mappings Table
-

- Can wholly/partially merge these data structures as you see fit
- Make sure to prevent these data structures from being evicted! (i.e. make them not pageable)
- For each data structure, decide the information needed per element, if it is per-process or global scope, and number of instances needed per scope
- See 4.1.3 for pros/cons of different underlying data structures you can use (ie array, list, etc)

1. Supplemental Page Table

What: Supplements the page table with additional information about each page

Purpose: 2 main usages:

Deciding which resources to free when a process terminates

Handling page faults

Data Stored: Additional information per page, organized in terms of segments or pages

2. Frame Table

What: Stores a mapping between frames and the user page occupying the frame

Purpose: Used for obtaining new frames

Use palloc_get_page if frame is available, otherwise use frame table to evict a page from its frame

Data Stored: Pointer to page occupying the frame and other information as needed

3. Swap Table

What: Tracks in use and free swap slots (page-sized regions of disk)

Purpose: Used for eviction (find a swap slot for the evicted page to be placed in, allocate new slots lazily) and paging in (free a swap slot when a page is read into memory)

Data Stored: If a slot is being used or is free

4.File Mapping Table

What: Tracks which pages are used by each memory mapped file

Purpose: Used to implement the mmap() and munmap() system calls

Data Stored: Pages in use by a memory mapped file

Paging

Paging In

- A page fault may be caused by a dereference of user virtual memory that is not loaded into a frame
- Thus, modify page_fault() to bring in the user page in this case, following these steps:

1. Locate the page that faulted in the SPT.
2. If 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 frame to store the page.
4. Fetch data into the frame.
5. Update the page table entry to point the virtual address to the new physical address

Eviction

If no frames are available for a page during paging in, then a page must be evicted from its frame

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 that refers to it
3. If needed, write the evicted page to file system or swap

Stack Growth

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

Memory Mapped Files

Memory Mapped Files

Implement the following system calls:

```
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

Accessing User Memory

Accessing User Memory in System Calls

- With the implementation of paging, page faults may now occur in system calls if the pages containing the user virtual addresses have been evicted
- The kernel needs to be modified to handle these page faults or prevent them from occurring
    - Prevent page faults by “pinning” pages so they cannot be evicted

Getting Started

Suggested 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

Advice

- Start early! Many students find this project harder than projects 1 and 2
- Design your data structures and their interactions before beginning to write code
- Be willing to change your design if it becomes overly complicated - a simple solution is possible
- Be mindful of synchronization to avoid deadlock
- Add files to the vm directory (see the reference diff)