Project 3:
Virtual Memory

CS 212, Winter 2022

Section Outline

- Paging Review
- Project Background
- Project Requirements
  - Supplemental Page Table
  - Frame Table
  - Swap Table
  - File mappings table
- Getting Started

Terminology

- **Page** - continuous region of virtual memory of size PGSIZE (virtual page).
- **Frame** - continuous region of physical memory of size PGSIZE (physical page)
- **Page Table** - data structure for translating virtual address to physical address (page to frame)
- **Swap Slot** - continuous, page-size region of disk space

Paging

Goal

Use disk to simulate virtual memory that is larger than physical memory.

Achieve speed of memory but size of disk.

- Disk access is orders of magnitude slower than memory.
- How can we accomplish this?

80/20 Rule

20% of memory receives 80% of accesses.

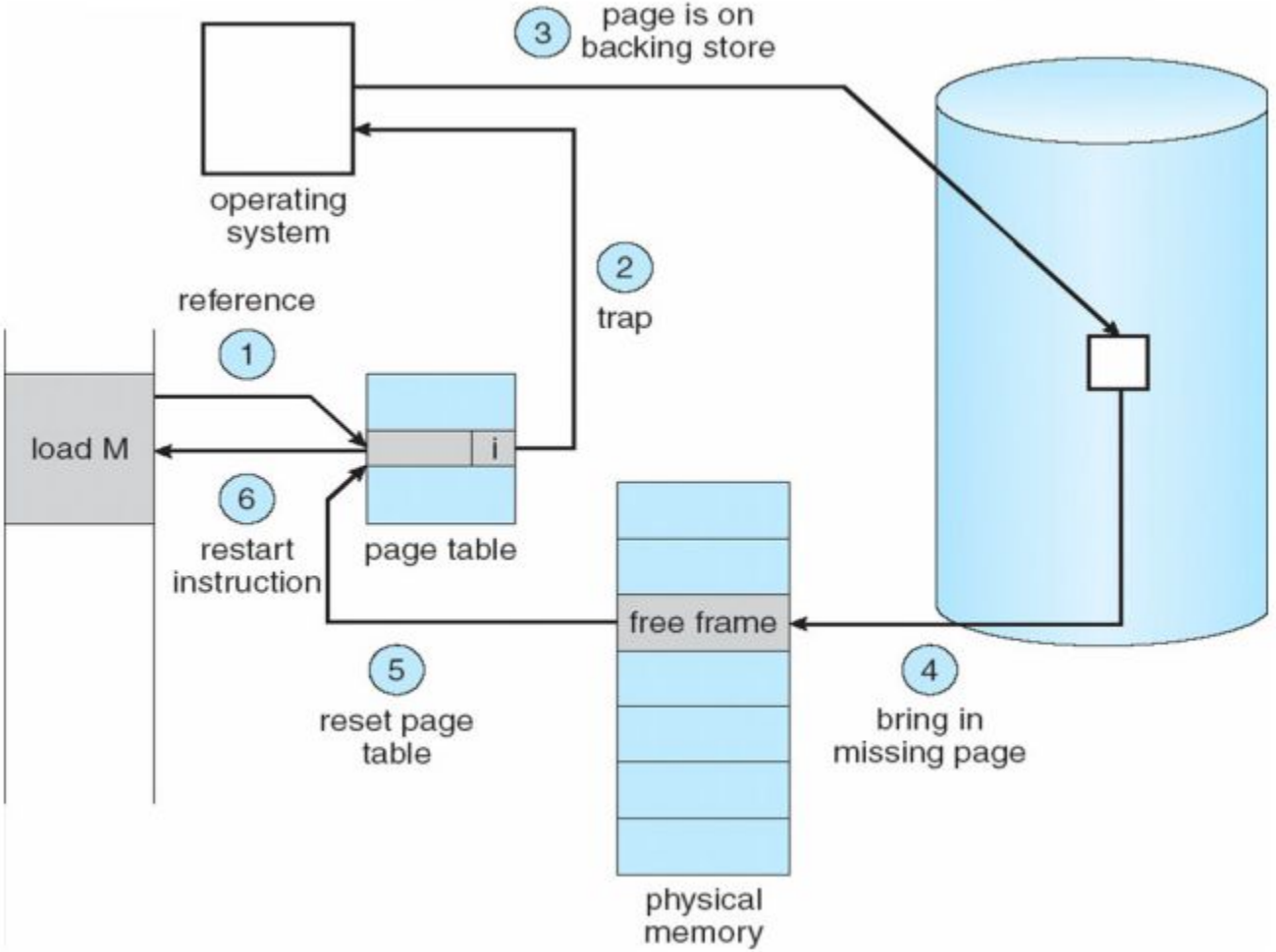
- Keep “hot” 20% in memory.
- Keep “cold” 80% on disk.

Paging In

If some virtual memory is now actually kept on disk, an access will lead to a page fault. We must handle page fault exceptions as follows:

1. Locate needed data using page tables.
2. Determine if access is valid.
3. **Load data back into memory (from disk).**
    a. This is called “paging in.”
4. Return control to prior instruction by exiting exception handler.

What happens if physical memory is full? We must **evict** some other data.


3
page is on
backing store
operating
system
2
trap
reference
1
load M
i
6
restart
instruction
page table
free frame
5
reset page
table
4
bring in
missing page
physical
memory

Eviction

Optimal strategy: replace page that will not be used for longest period of time.

Approximate strategy: Replace least recently used (LRU) page.

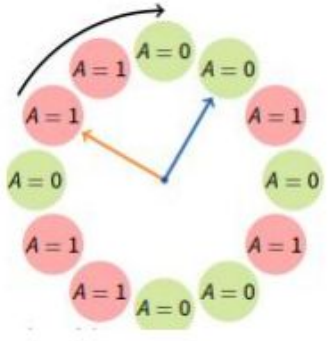
- Past often predicts the future.
- One implementation: clock algorithm (lecture 6)

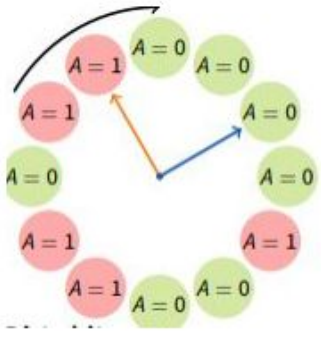
Clock Algorithm

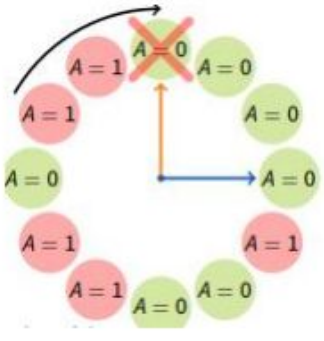
Keep pages in FIFO circular list. Evict on FIFO basis, but skip a page if “Accessed” bit is 1.

First clock hand clears Accessed bit, second clock hand evicts.

When total memory is smaller, can be done with one clock hand that evicts if A = 0, or clears bit & skips page if A = 1.







Project Background

Motivation

After proj2, the number and size of user programs is limited by the machine's main memory size.

Your goal is to resolve this limitation through **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

All physical memory is divided into frames.

Two pools for physical memory: user pool and kernel pool

```
palloc_get_page (PAL_USER)
```

- Gets physical page (i.e. frame) from user pool.

All physical addresses are accessible through kernel virtual address space

```
PHYS_BASE + phys_address
```

- Accesses physical address `phys_address`

Accessed and Dirty Bits

Built-in hardware support for tracking access and edits to pages

On any read to a page, the CPU sets the Page Table Entry's "Accessed" bit to 1.

On any write to a page, the CPU sets the Page Table Entry's "Dirty" bit to 1.

CPU never sets bits to 0, but the OS can.

Beware of Aliases!

Every user virtual page (upage) has an *alias*, which is the kernel virtual page (kpage).

When an aliased frame is accessed, the accessed/dirty bits are only set for the PTE of the virtual address used to access the frame.

To determine if a frame has been accessed, you must either check both upage and kpage PTEs, or only access the frame from the kernel using the upage PTE.

Requirements

Data Structures

- Supplemental Page Table
- Frame Table
- Swap Table
- File mappings table

Note: data structures can be wholly or partially merged (you don’t necessarily need 4 distinct structures).

Tip: Do not make these data structures page-able (i.e., never evict them from their frame).

Data Structures

For each data structure, you must decide:

- What information each element contains
- Per-process or global scope
- Number of instances required within scope

Possible options: array, list, bitmap, hash table

Implementations contained in `lib/kernel`

See **4.1.3 Resource Management Overview** for pros and cons of each option

Supplemental Page Table

Supplements the page table with additional data about each page

Used for two purposes:

1. Deciding which resources to free when a process terminates
2. **Handling page faults**

Page Fault Handling

Previously, a page fault always implied a bug (in user or kernel code).

This is no longer the case.

A dereference of valid virtual memory that is not loaded into a frame will lead to a page fault.

Page Fault Handling

Modify `page_fault()` in `userprog/exception.c`:

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 (using `userprog/pagedir.c`)

Frame Table

Contains one entry for each frame that contains a user page.

Each entry contains a pointer to page that occupies the frame and other data of your choice.

Most important operation: **obtaining a new frame**

1. Easy if frame is available (`palloc_get_page(PAL_USER)` not null).
2. If no frame is available, frame table must **evict** a frame.

Eviction

Decide on page replacement algorithm. Must be at least as good as Clock Algorithm.

Most groups choose to implement Clock Algorithm from lecture.

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

Swap Table

Tracks in-use and free swap slots.

Allow picking unused swap slot when page is evicted from frame → swap

Allow freeing swap slot when page is read back from swap → frame

Swap slots should be allocated *lazily*- only when required.

Run Pintos with `--swap-size=n` to run with *n*-MB swap partition.

Use `devices/block.h` to read/write from swap slot as a block device.

Memory Mapped Files

You must 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

Managing Memory Mapped Files

Your virtual memory solution must properly track which pages are used by memory mapped files.

When a memory-mapped file is evicted, it is written back to file (if dirty).

Stack Growth

If stack grows past its current size, allocate a new stack page.

Only allocate a new page if an access appears to be a stack access. Devise some heuristic to decide this.

Impose some absolute limit on stack size.

All stack pages are candidates for eviction.

Accessing User Memory

While accessing user memory from any system call, page faults may now occur if the pages containing the user virtual addresses have been evicted.

In `userprog/syscall.c`, you must be able to handle this case, either by

- Handling page faults when they occur.
- Avoiding page faults.

Your page eviction code can cooperate with the kernel code performing the access via “pinning,” or locking a page into its frame.

Only pin when necessary, and make sure to unpin when no longer needed.

Getting Started

Suggested Order of Implementation

1. Fix remaining project 2 bugs
2. Frame Table
    a. Do not implement eviction yet. If you run out of frames, fail allocator or panic kernel.
    b. Should still pass all project 2 tests.
3. Supplemental Page Table and page fault handling
    a. Should still pass all project 2 tests, and only some robustness tests.
4. Stack growth, memory mapped files, and page reclamation.
5. Eviction
    a. Pay attention to synchronization. What if two processes evict the same frame at the same time? What if a process accesses a frame as it's being evicted?

Advice

- Start early!
- Design first!
  - Decide how you will design each of the data structures before you start implementing.
  - Know the data they will store, if they are per-process or global, etc.
  - Know the interface they will expose and (roughly) where these functions will be called.
- Be willing to change your design
  - If things are getting very complicated during implementation, take a step back. Is there a simpler way to accomplish your goal?
- Pay attention to synchronization while designing and implementing.
  - Make sure to avoid deadlock by avoiding circularity in graph of synchronization requests.
  - Organize synchronization mechanisms hierarchically.
- Add files to `vm` directory.

Questions?