Project 3: Virtual Memory

CS140 Winter 2020

Overview

- Due Date: Friday, February 28, by 12pm (two weeks from today)

- Goal
  - Total size of programs running > size of physical memory
  - 80/20 rule, store data that isn't currently used on disk
- Solution
  - Demand paging
    - Divide memory into fixed-sized "pages"
    - If access data not currently in memory (page fault), "page in"
      - May involve eviction
      - Try to evict the page that will be used furthest in the future

Other Requirements

- Stack growth
  - Allocate new stack pages as necessary
- Memory mapped files
  - “map” a file into virtual pages
  - Operate on file with memory instructions instead of read/write system calls
- Accessing user memory
  - Make sure data the kernel is currently operating on doesn’t get paged out
  - Might be holding resources needed to handle the page fault
    - Avoid deadlock

Disclaimer

- Most people think this assignment is really hard
- This assignment is really fun!
- System design is really fun!!!

Terminology

- Page
  - Contiguous region of virtual memory (e.g. virtual page)
- Frame
  - Contiguous region of physical memory (e.g. physical page)
- Page table
  - Data structure to translate a virtual address to physical address (page to a frame)
- Swap slot
  - Contiguous, page-size region of disk space in the swap partition
  - Some evicted pages are written to swap (e.g. stack pages)

Handling Page Faults

- Page fault
  - User accesses memory address for data that isn't currently loaded into memory
- How to "page in"?
  - Determine if memory access was valid
    - Might need new stack page
    - If not valid, terminate process
  - Find a frame to use*
  - Locate data that belongs in the page, fetch data into frame
  - Install page table entry for faulting virtual address to the physical page
- Where is this information?
  - Create/use *per-process* **supplemental page table** (SPT)
    - Determine valid addresses
    - Locate data that belongs in the page

*more on this soon

Finding a Frame

- Check if any available
  - `palloc_get_page(PAL_USER)` allocates new user frames
- If not, evict
  - Create/use *global* **frame table** to iterate over all frames used by any process
  - Global page replacement algorithm
    - Approximates LRU; at least as good as clock / "second chance"
      - If page accessed, set not accessed.
      - If page not accessed, evict.
  - Clear evicted page
    - Remove references to the frame from any page table that refers to it
    - If dirty, write to file system or swap
- If no frame can be evicted without allocating a swap slot, but swap is full, panic the kernel.

Memory Mapped Files

- `mapid_t mmap (int fd, void *addr)`
  - Maps file into consecutive virtual pages in the process's virtual address space, starting at `addr`
  - Operate on file with memory instructions instead of read/write system calls
  - Fails if address invalid
- `void munmap (mapid_t mapping)`
  - Removes the mapping
- Lazily load pages
- File is backing store (on eviction, writes back to file)
- Create/use **file mapping table**

Accessing User Memory

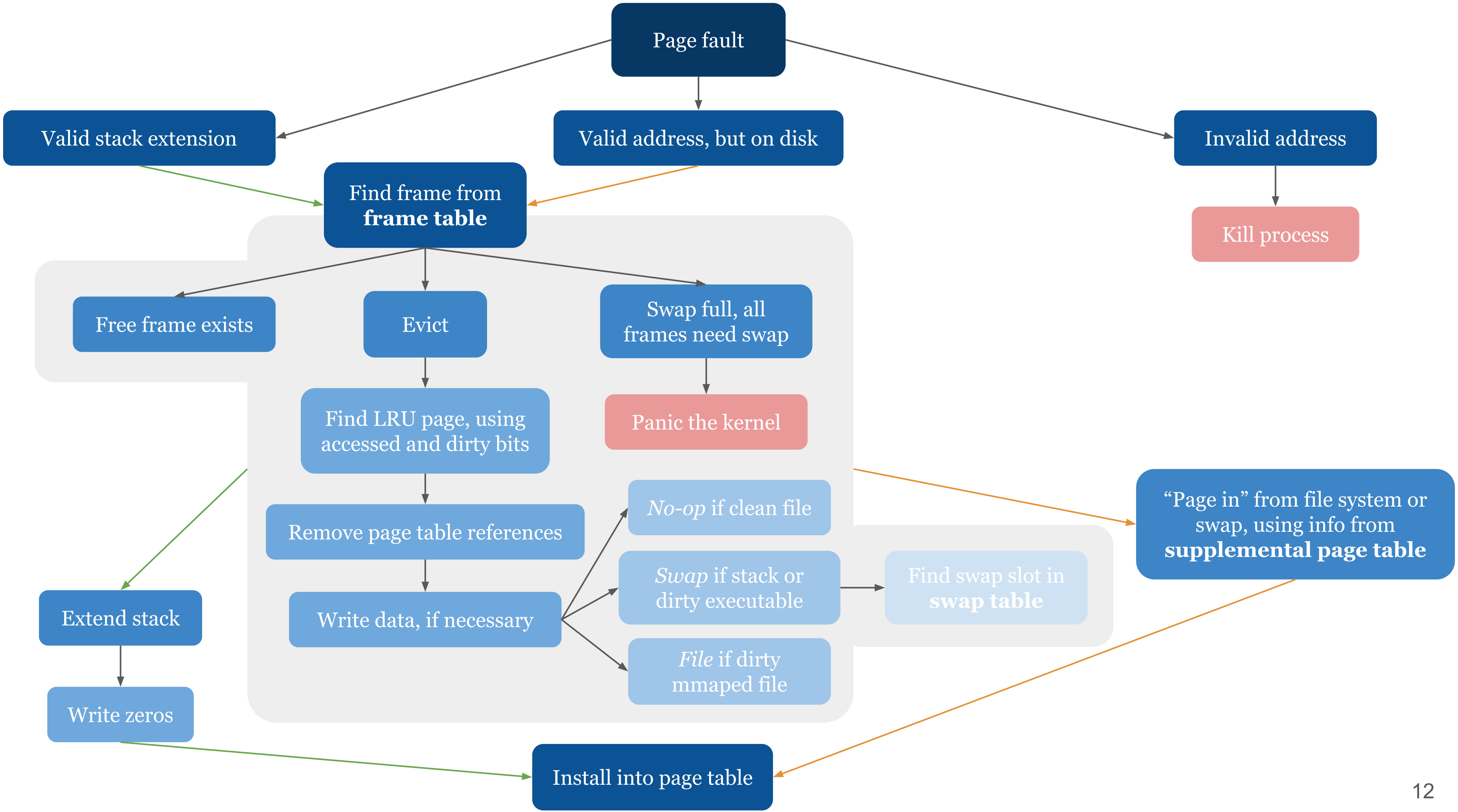
- In project 2, you rejected user addresses for data not in memory
    - An address could be valid but not currently mapped
- Make sure pages aren’t evicted from frames while accessed by kernel
    - Might be holding resources needed to handle the page fault
    - Can implement “pinning” or “locking” to make sure page isn’t evicted
- Accessed / dirty bits different per page
    - Always access user data through the user virtual address

Swap

- Storage for stack pages and dirty executable pages
  - `block_get_role (BLOCK_SWAP)`
- Create/use *global* **swap table** to track in-use and free swap slots
  - Pick swap slot during eviction
  - Free swap slot when paged back in or process terminates

Types of Data in Memory

- Executables
    - Loaded lazily
    - Written to swap if dirty (if *ever* dirty)
    - Read-only and unmodified pages can be read back from executable
- Stack
    - Allocate additional pages only if they “appear” to be stack accesses
        - `PUSH`: 4 bytes below `%esp`
        - `PUSHA`: 32 bytes below `%esp`
        - Get `%esp` from `struct intr_frame` passed to `page_fault()`
    - Written to swap when evicted
- Files, from mmap
    - Loaded lazily
    - Written back to file if dirty


Page fault
Valid stack extension
Valid address, but on disk
Invalid address
Find frame from **frame table**
Kill process
Free frame exists
Evict
Swap full, all frames need swap
Find LRU page, using accessed and dirty bits
Panic the kernel
Remove page table references
No-op if clean file
"Page in" from file system or swap, using info from **supplemental page table**
Swap if stack or dirty executable
Find swap slot in **swap table**
Extend stack
Write data, if necessary
File if dirty mmaped file
Write zeros
Install into page table

Suggested Order

1. Must have working project 2
    - Fix any bugs!
2. Frame table
    - Don't implement swapping yet
    - You should still pass all project 2 tests
3. Supplemental page table and page fault handler
    - Lazily load code and data segments via page fault handler
    - You should pass all project 2 functionality tests, but only some robustness tests
4. Stack growth, mapped files, page reclamation
5. Eviction
    - Don't forget synchronization
        - What if a process accesses a page during eviction?
        - What if two processes are trying to evict pages at the same time?

Data Structure Choices

- Arrays
  - Simplest approach, sparsely populated array wastes memory
- Lists
  - Pretty simple, traversing a list can take lots of time
- Bitmaps
  - Array of bits each of which can be true or false
  - Track usage in a set of identical resources
- Hash Tables

Necessary conditions for deadlock

1. Limited access (mutual exclusion)
2. No preemption
3. Multiple independent requests (hold and wait)
4. **Circularity in graph of requests**
    - A holds mutex `x`, wants mutex `y`; B holds `y`, wants `x`

Advice

- Start early
- Design first
  - Make sure you really understand the assignment before coding
  - Design something that makes it easy for you to convince yourself it is correct — draw diagrams
- Be open to changing your design
  - If things feel really hard, take a step back
  - A better design might save you hours of debugging
- Avoid deadlock
  - **Organize your synchronization mechanisms hierarchically**
  - Write out all cases in which locks are acquired, and the order in which they are acquired
- Add files

Good luck & have fun!