Project 3

Thierry Sans

Overview

Goal

- Total size of programs running > size of physical memory
- Store data that is not currently used on disk (80/20 rule)

Solution

➡ Demand paging - divide memory into fixed-sized "pages"

- if access data not currently in memory (page fault), "page in"
- if the memory is full, "page out" (page eviction algorithm)

Other requirements

Stack growth
Allocate new stack pages as necessary

Memory mapped files (for page in and page out)

- "map" a file into virtual pages
- Operate on file with memory instructions instead of read/write system calls

Accessing user memory

- Make sure kernel's data doesn't get paged out
- Might be holding resources needed to handle the page fault (avoid deadlock)

Scope of the work

```
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(-)
```

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

What is a page fault?
User accesses memory address for data that isn't currently loaded into memory

How to "page in"?

1. Determine if memory access was valid
   (If not valid, terminate process; might need new stack page)
2. Find a frame to use (more next slide)
3. Locate data that belongs in the page, fetch data into frame
4. Install page table entry for faulting virtual address to the physical page

Where is this information?
Create/use per-process supplemental page table (SPT)

1. Determine valid addresses
2. Locate data that belongs in the page

Finding a Frame

Check if any available
`palloc_get_page(PAL_USER)` allocates new user frames

If not, evict

1. Create/use global frame table to iterate over all frames used by any process
2. Implement global page replacement algorithm that approximates LRU (clock/"second chance")
   - If page accessed, set not accessed.
   - If page not accessed, evict.
3. Clear evicted page
   - Remove references to the frame from any page table that refers to it
   - If "dirty" (i.e page has been modified), 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

➡ Create/use file mapping table

➡ On load, create page in file at first (lazy loading)

➡ On evict, writes back to file (backing store)

Accessing User Memory

- ➡ 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
- ➡ Maintain 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 directly

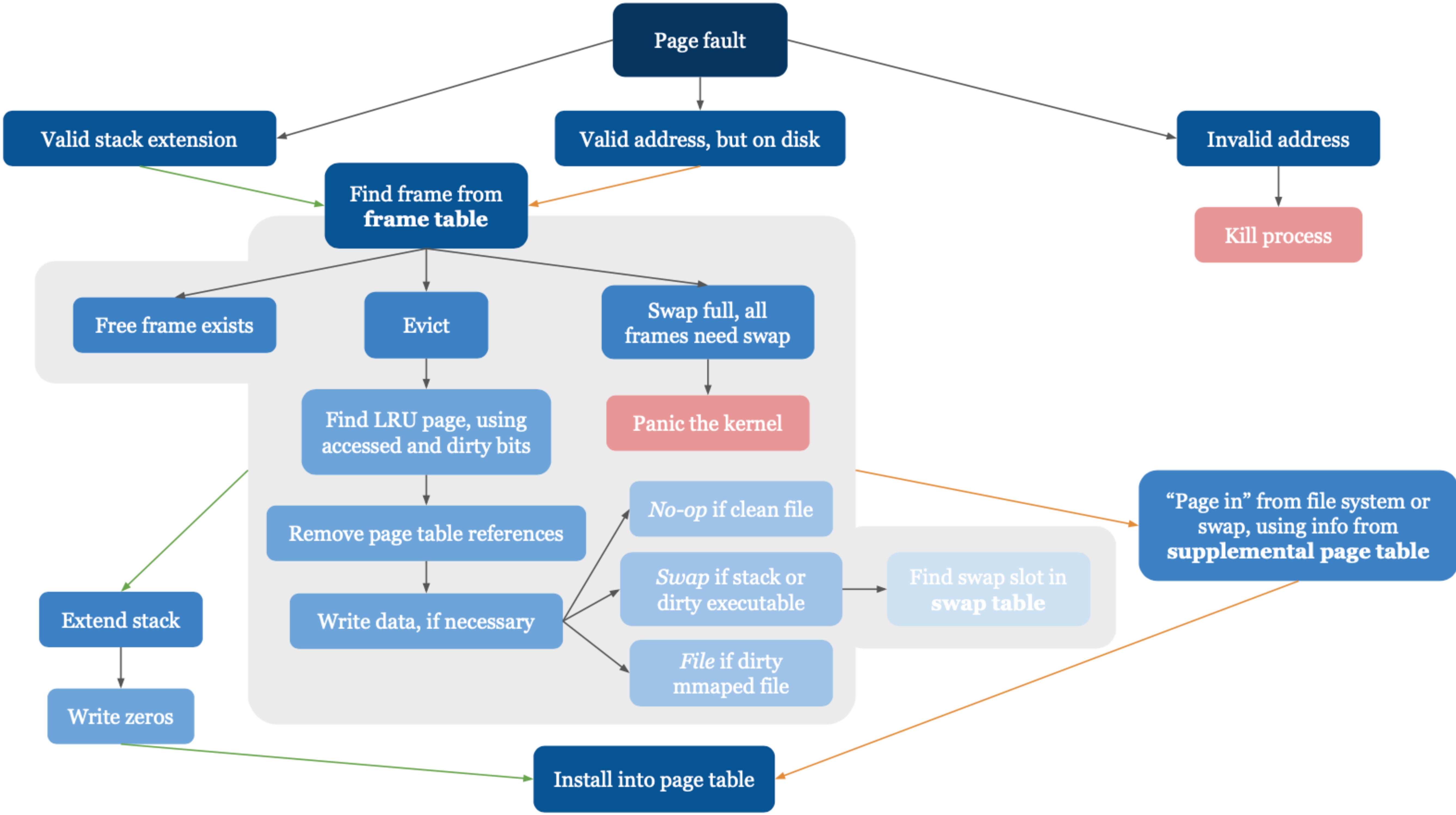
Stack

- 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

Features overview



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
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

Acknowledgments

Some of the course materials and projects are from

- Ryan Huang - teaching CS 318 at *John Hopkins University*
- David Mazière - teaching CS 140 at *Stanford*