Project-3: Virtual Memory

Masoud Moshref

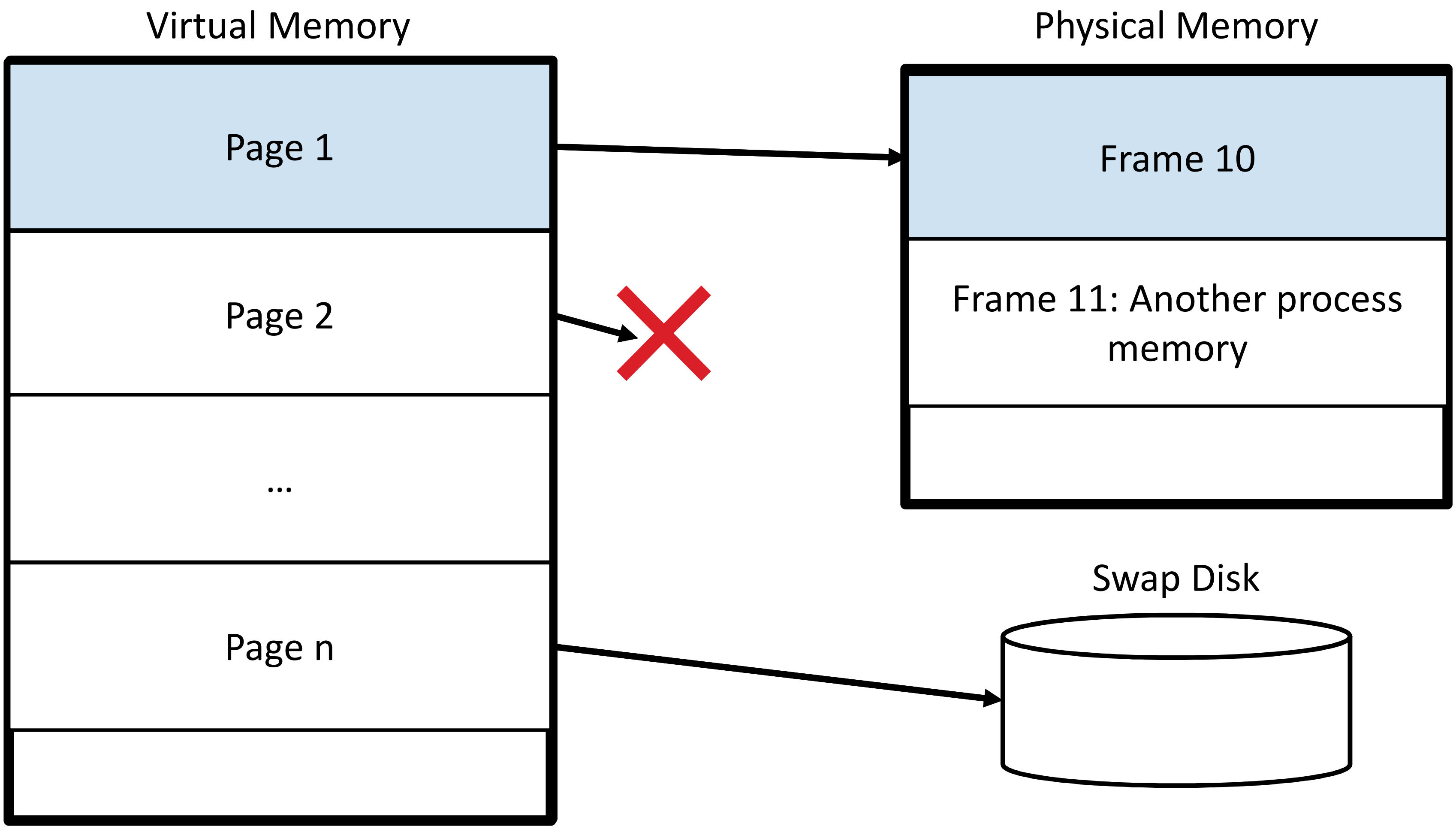
CSCI-350

Fall 2014

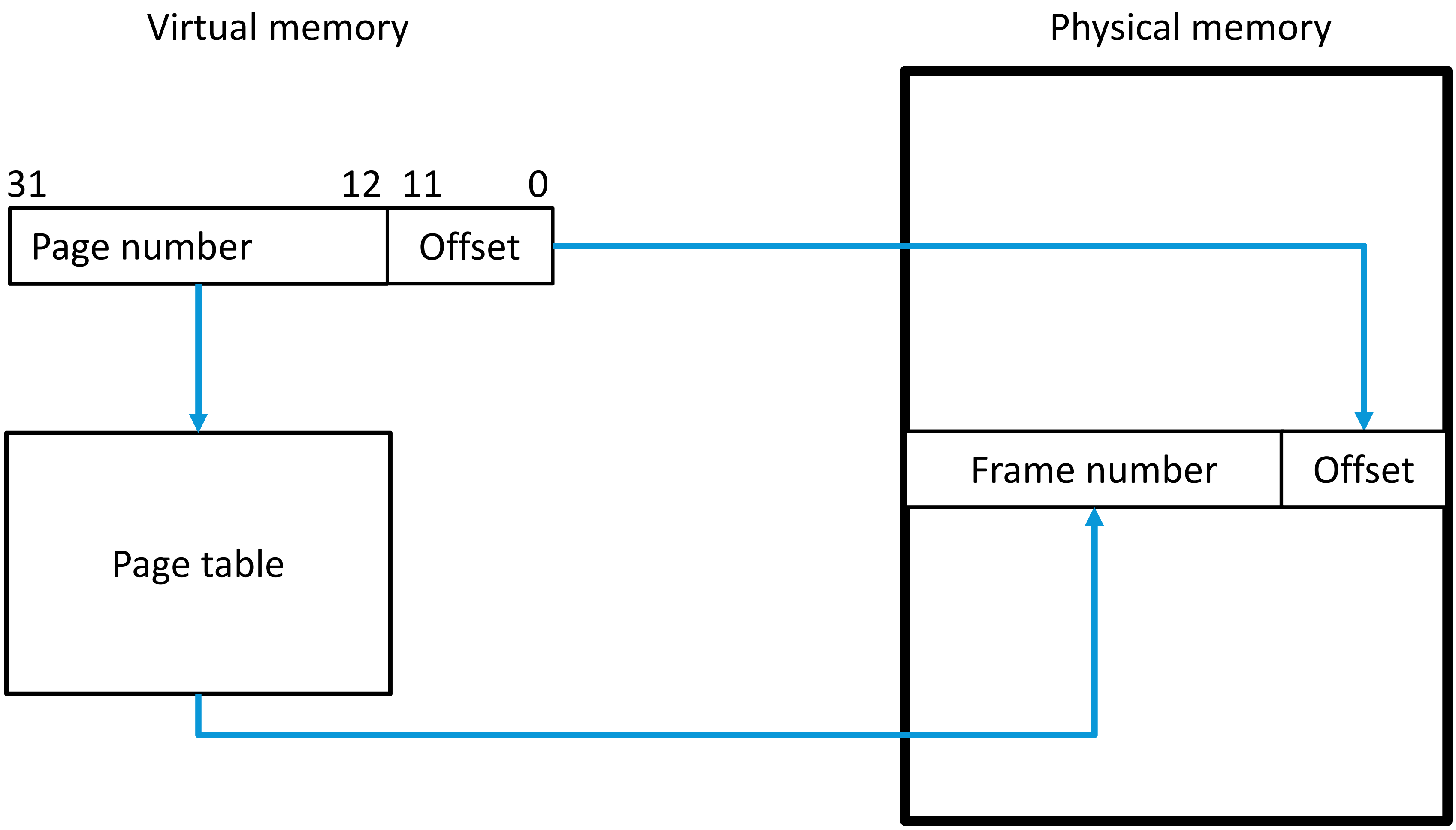
USC University of Southern California

Background

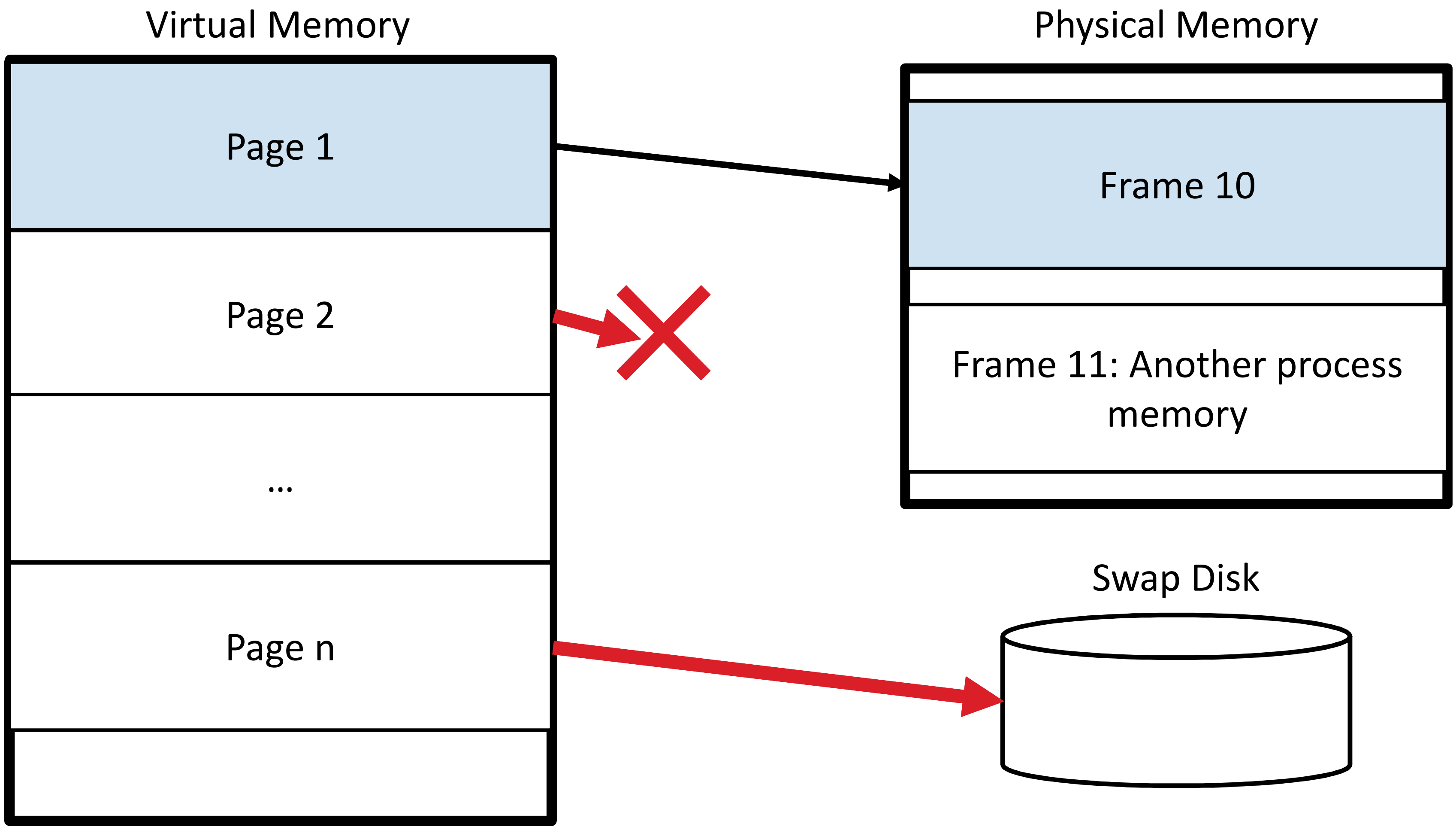
- Limited machine memory limits user programs
- Virtual memory removes that limitation



Background: Virtual Memory



Page Fault


Virtual Memory
Page 1
Page 2
...
Page n
Physical Memory
Frame 10
Frame 11: Another process memory
Swap Disk

Page Fault

1. Where is the data

   Supplemental page table

   - Not mapped or kernel
   - Swap
   - not loaded from file system (executable, mapped files)
   - zero but not initialized (data segment, stack growth)

2. Obtain a frame from memory

   Frame management

   - May need eviction

3. Fetch/init the data
   - Fetch from swap

     Swap management
   - Fetch from filesystem

     Memory mapped files

4. Update the page table

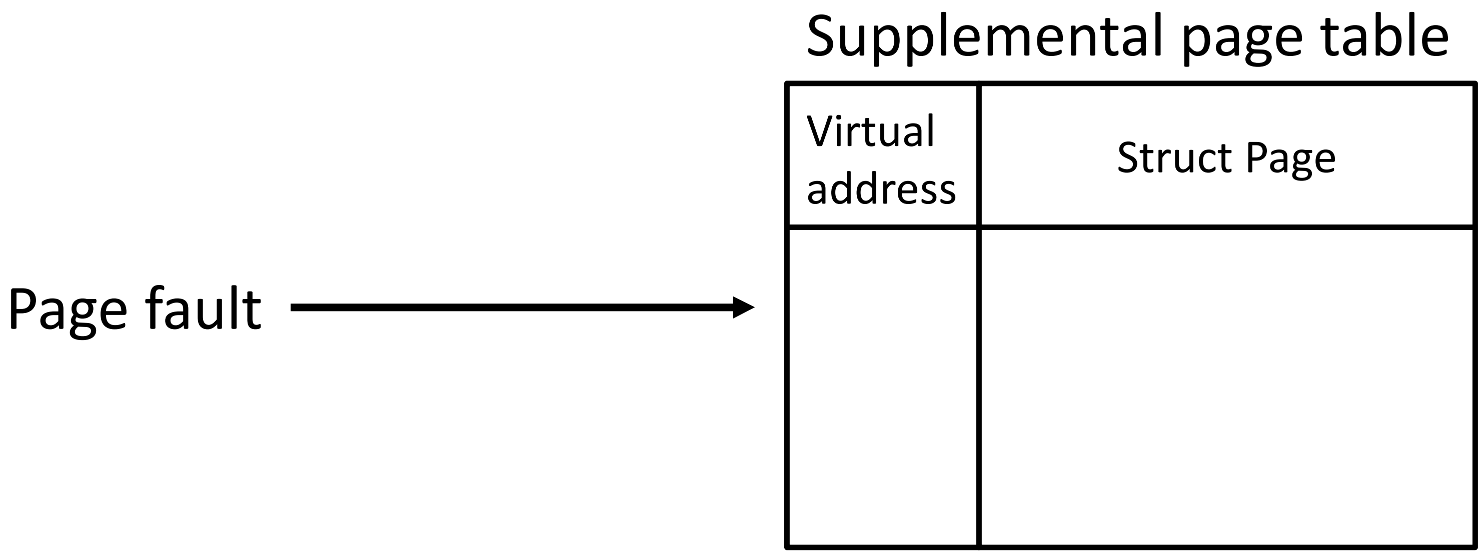
   pagedir_set_page()

Project Requirements

- Supplemental page table
- Frame management
- Swap management
- Memory mapped files

Supplemental Page Table

- CPU page table can only keep limited information
  - Dirty
  - Accessed
- You may need more for
  - At page fault, not mapped or in swap (where is it in swap)
  - At thread exit must free reserved resources in memory & swap



Supplemental page table

Virtual address	Struct Page

Managing Frames

- Allocate a frame for a page at page fault
  - Use palloc_get_page (PAL_USER) to get a frame from user pool
- If no free frame, pick a page to evict
  - How to know if there is no more
    - palloc_get_page returns null
  - Which frame? Approximate Least Recently Used (LRU)
    - Second chance / clock algorithm
    - Use accessed bit for pages (pagedir.c functions)
  - How?
    - Unmap its page from its process page table (pagedir.c functions)
    - Write it to swap if <u>necessary</u>
    - Update supplemental page table

Swap

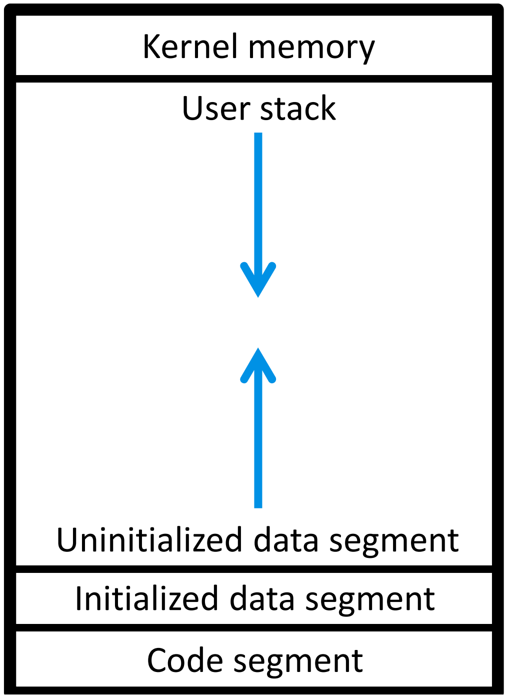
- A block device (see devices/block.h)
- Add “--swap-size=4” to your pintos command to ask a temporary swap disk with size 4MB
- Swap structure:
  - Each sector is 512 bytes (not each page is 4KB)
- How to access swap device
  - struct block * swap =block_get_role (BLOCK_SWAP);
- How to write to swap
  - block_write (struct block *block, block_sector_t sector, const void *buffer)
- Must keep track of which sectors are free (e.g., by bitmap)

Memory Mapped Files

- Map a file into a memory address: Changing the content of memory is the same as changing the file
- System call
  - mapid_t mmap (*int fd, void *addr*)
  - void munmap (*mapid t mapping*)
- Consecutive virtual address
  - Requests n page from VM
- Fail if
  - Addr=0
  - Not page aligned address
  - Overlapping with already assigned virtual address
  - Invalid fd or filesize=0

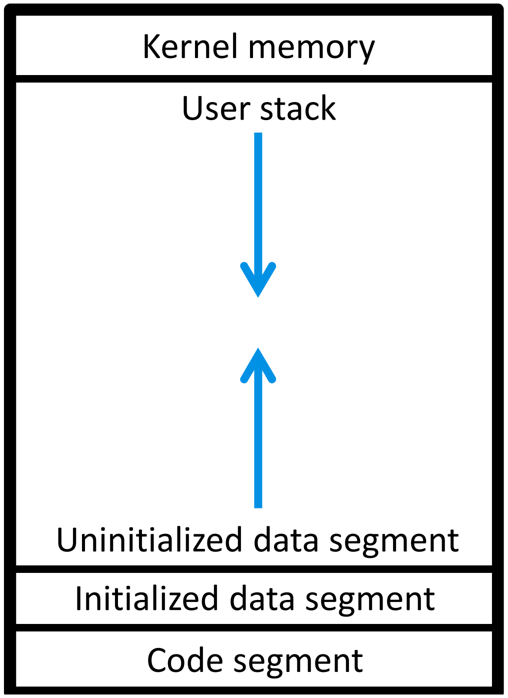
Types of Pages and Lazy Swap

- From executable file
  - Requested from load_segment() in process.c
  - Load at page fault
  - Evict: Don’t write to swap, just read them again from filesystem
- Data segment
  - Requested at loading the executable (load_segment() in process.c)
  - Init at page fault
  - Evict: Don’t write to swap, if not dirty
  - There can be one page both code and data!



Type of Pages

- Stack
  - Requested:
    - Initial page is requested at setup_stack() in process.c →Load at request
    - Requested by stack growth (grows at most 32Bytes below sp)→ load at fault
    - At most 8MB stack
  - Evict: Put in swap
- Memory mapped file
  - Request: at mmap systemcall
  - Load: At page fault
  - Evict: Save in file if changed (no swap device)



Kernel memory

User stack

Uninitialized data segment

Initialized data segment

Code segment

Corner Cases

- User process passes a virtual address to system calls
  - May not be in memory → page fault in kernel!
  - Thus bring it back before accessing it in kernel
- Don’t let it evict during system call
  - How can it happen?
    - X writes to file from buffer at page A
    - Meanwhile Y receives CPU and page faults on stack for page B
    - Frame manager evicts A for B
  - Solution: pinning pages to memory
- If a page fault needs IO, other threads must be able to run and even page fault

Page Sharing

- If two process use the same executable
  - Share read-only pages
- Challenges
  - At page fault, the data may be in a memory frame.
    - Only update page table
  - At eviction, all pages that refer to the content of the victim frame must be updated
  - Accessed/dirty bit is only updated for the current process page table entry

Submission Requirement

- Make Project-3 branch
- Implement your code in src/vm directory
  - src/vm is empty!
  - How to add a file to be built?
    - Add *.c file names in src/Makefile.build (vm_SRC variable)
  - You may change other directories as well (look at FAQ)
- Must pass project-2 test cases
  - 0% project-2 test cases
  - 80% project-3 specific test cases
  - 20% design doc
  - 20% page sharing (extra)
    - No test case, we go through the design doc and code

Design Document

- src/vm/designdoc.pdf
- What happens during a page-fault?
  - Data structures & algorithms
- How did you implement eviction & pinning?
  - Data structures & algorithms
- How do you implement memory mapped files?
  - System call handler code
  - Data structures to keep track of them
- How did you implement page sharing

Hints

- First think about a design
  - What do the data structures look like?
  - What APIs or methods should you have?
  - For each data structure is it per process or global?
- Learn how to use bitmap and hash table
  - Read pintos doc Appendix
  - Look into lib/kernel/bitmap.h lib/kernel/hash.h

Hints

- Order of implementation
  - Frame table without swapping only for loading executables, data segment & stack in process.c
  - Supplemental page table for page fault handler
    - Now not load anything in process.c but let page fault loads it
  - Page reclamation on process exit
  - Stack growth
  - Page eviction & swapping
  - Memory mapped files
- Order of test cases
  - Project 2 should pass anytime
  - pt-*
  - page-*
  - mmap-*

Questions?