Logistics

• Due: Friday, November 13 at noon

• Open-ended design
  – Read assignment spec carefully
  – Think before you code

• Challenging (start early!)
VM Overview

• Key VM ideas
  – Isolation/Protection
    • Each user process can only touch its own memory
  – Resource sharing
    • Allow total memory use by all running processes to exceed physical memory
  – Abstraction
    • VM implementation should be transparent to user programs
VM Overview

• Key VM Terms
  – **Page**: Contiguous virtual memory region (4096 bytes in Pintos)
  – **Frame**: Contiguous physical memory region (same size)
  – **Swap**: (non-FS) Disk storage used to hold evicted pages
  – **Page table**: Contains active virtual-to-physical address mappings
  – **Supplementary page table**: Contains other* information about pages (active + inactive)
  – **Frame table**: Data structure for tracking frame allocation and eviction
  – **Swap table**: Data structure for tracking Swap usage
VM Overview

• **Requirements**
  – Design + implement Frame Table
  – Design + implement Supplementary Page Table (S.P.T.)
  – Stack growth
  – Memory-mapped files (new syscalls!)
  – Implement eviction + swap table
  – Handling page faults
  – Resource cleanup on exit
Frame Table

• Goal: Encapsulate allocation state of physical memory, and provide free frames when requested
  – Track which frames are currently in use
  – Return free frame if available
  – Otherwise, evict a frame and then return it (more on this later)
  – Allow a frame to be pinned/locked into place (can’t be evicted!)
Supplementary Page Table

- **Goal:** Track bookkeeping information about a process’s virtual address space
- Used to handle valid user memory access page faults, clean up resources (and to determine which user memory accesses are invalid)
- Design question: best way to store/track mappings?
- Enables lazy loading of executables (**requirement**)
  - Install a page in the S.P.T. (but not into a frame), then first access causes a page fault which then loads the page

- **Tip:** You will be replacing most calls to `palloc_get_page` with calls to your frame and page table methods
- **Tip:** Hash table (lib/kernel/hash.c) might prove useful here…
Stack Growth

• Project 2: stack limited to a single page
• Now: stack grows dynamically (and lazily)
• Triggering stack growth
  – Situation: stack pointer grows beyond allocated region, then next stack access triggers a page fault
  – Problem: How can we distinguish this page fault from any other actual invalid pointer dereference?
  – Solution: Heuristics!
    • Compare faulting address to the stack pointer (esp)
    • Set total stack size limit
    • See assignment handout section 4.4.3\(^1\) for details

\(^1\) http://www.scs.stanford.edu/15au–cs140/pintos/pintos_4.html#SEC71
Memory-Mapped Files

• New way to interface with file system: map file to a contiguous memory region
  – Use memory instructions directly on file data
  – Lazily load page-sized parts of file when accessed
  – Track location using supplementary page table
  – Advantages?

• New system calls:

  System Call: mapid_t mmap (int fd, void *addr)
  Maps the file open as fd into the process's virtual address space. The entire file is mapped into consecutive virtual pages starting at addr.

  System Call: void munmap (mapid_t mapping)
  Unmaps the mapping designated by mapping, which must be a mapping ID returned by a previous call to mmap by the same process that has not yet been unmapped.

• Tip: Similar requirements to loading executables. Difference?
Eviction and Swap

- **Goal**: Transparently give each process a virtual memory address space from 0 up to PHYS_BASE
- **Issue**: Physical memory is smaller than virtual address space (and/or total process resource requirements)
- **Solution**: Treat physical memory as a cache; use other resource (disk) as backing store.
  - Frame table maintains the state of this cache
  - Use an eviction policy that approximates LRU (clock algorithm?)
  - Use filesystem or swap to store evicted data as appropriate
  - Use S.P.T. to track location of evicted entries
Eviction and Swap, cont.

- **Eviction Policy**
  - Exact LRU is expensive (requires updating timestamp on each access)
  - Idea: approximating LRU is “good enough”, so we leverage the accessed bit already supported by hardware
  - **Clock algorithm** (also see [VM lecture notes](http://www.scs.stanford.edu/15au-cs140/notes/vm_os.pdf)):
    - Maintain circular list of frames and pointer to some frame in the list
    - Second chance replacement
      - if accessed bit is 1, clear the bit, advance the clock hand, and try again
      - If accessed bit is 0, evict the page
    - Optimization: add a second clock hand at a fixed distance ahead which only clears accessed bits to reduce worst-case eviction time
    - Reminder: can’t evict pinned/locked frames
  - Eviction is **lazy**: only evict when a new frame is needed
  - **Aliasing**: single physical page can be accessed using both the kernel and user virtual address; must always use the same one, or keep track of both (consider accessed and dirty bits)

Eviction and Swap, cont.

• **Eviction Implementation**
  – Once we have found a frame to evict, what do we do with memory contents?
  – Use S.P.T. to determine where that data should reside, if anywhere (i.e., mapped file)
    • Use dirty bit to determine if content is modified from backing store
  – Default to swap if no other backing store exists
  – Prevent page from being accessed during eviction (how?)
    • **Tip:** Consider timing of clearing page table entry vs. checking dirty bit

• **Swap:** special disk partition dedicated to storing evicted pages
  – Consists of \( n \) identical page-sized “slots” that can be used by any process
  – Can store a page and obtain the \( \text{slot}_\text{id} \) for later retrieval
  – **Tip:** bitmap (lib/kernel/bitmap.c) might be useful here…
Eviction and Swap, cont.

• **Eviction parallelism**
  
  – **Requirement:** any page fault that triggers I/O (i.e. evicting to swap, loading page from filesystem, etc) should not block other page faults/processes that do not require I/O
    - i.e. can’t hold global lock on frame table during I/O
  
  – **Requirement:** prevent kernel deadlocks caused by accessing evicted pages
    - Example: `file_read` page faults while holding filesystem lock, but the page fault handler may need to write to the filesystem to evict a page...
    - Solution: pin/lock pages while they are being accessed by the kernel (but no longer than needed!)

  – Consider other cases, i.e., process A faults on a page whose frame is being evicted by process B
    - Sensible locking/granularity will solve most such issues
Page Fault Handling

• Project 2: User page faults always terminate process

• VM: Some user page faults are valid accesses and must be handled
  – Which ones?
    • Evicted page
    • Stack access
    • Memory-mapped file access
    • Lazy executable loading
  – Allocate frame, find/load page data, update page table, etc
Resource Cleanup

• On process exit, need to free all system resources used by that process
  – Before VM this was easier (less state to track)
  – Now: more resources need cleanup
    • Memory-mapped files (unmap)
    • Swap slots (free)
    • Frames (free)
    • S.P.T. for the process (free)
Recap

• **Requirements**
  – Design + implement Frame Table
  – Design + implement Supplementary Page Table (S.P.T.)
  – Stack growth
  – Memory-mapped files (new syscalls!)
  – Implement eviction + swap table
  – Handling page faults
  – Resource cleanup on exit
Other Tips

• **Data structure choices**: consider primary access patterns (i.e. random vs. iterative). Don’t roll your own

• **Leverage existing code**: helper functions like `pg_ofs` and `pg_round_down` in `threads/vaddr.h`

• **New modules**: Decompose new code into logical modules (i.e., `frame.c/h, page.c/h, swap.c/h`)
  – Follow pintos naming conventions for exported methods!

• **Synchronization**: Think through synchronization issues before implementing (can be tricky)
  – Decide on locking granularity and entry/exit points
  – Consider which data is accessed by a single thread vs. multiple

• **Jitter**: might be useful to expose synchronization issues when testing
Questions?