Pintos Virtual Memory

Daniel Chiu

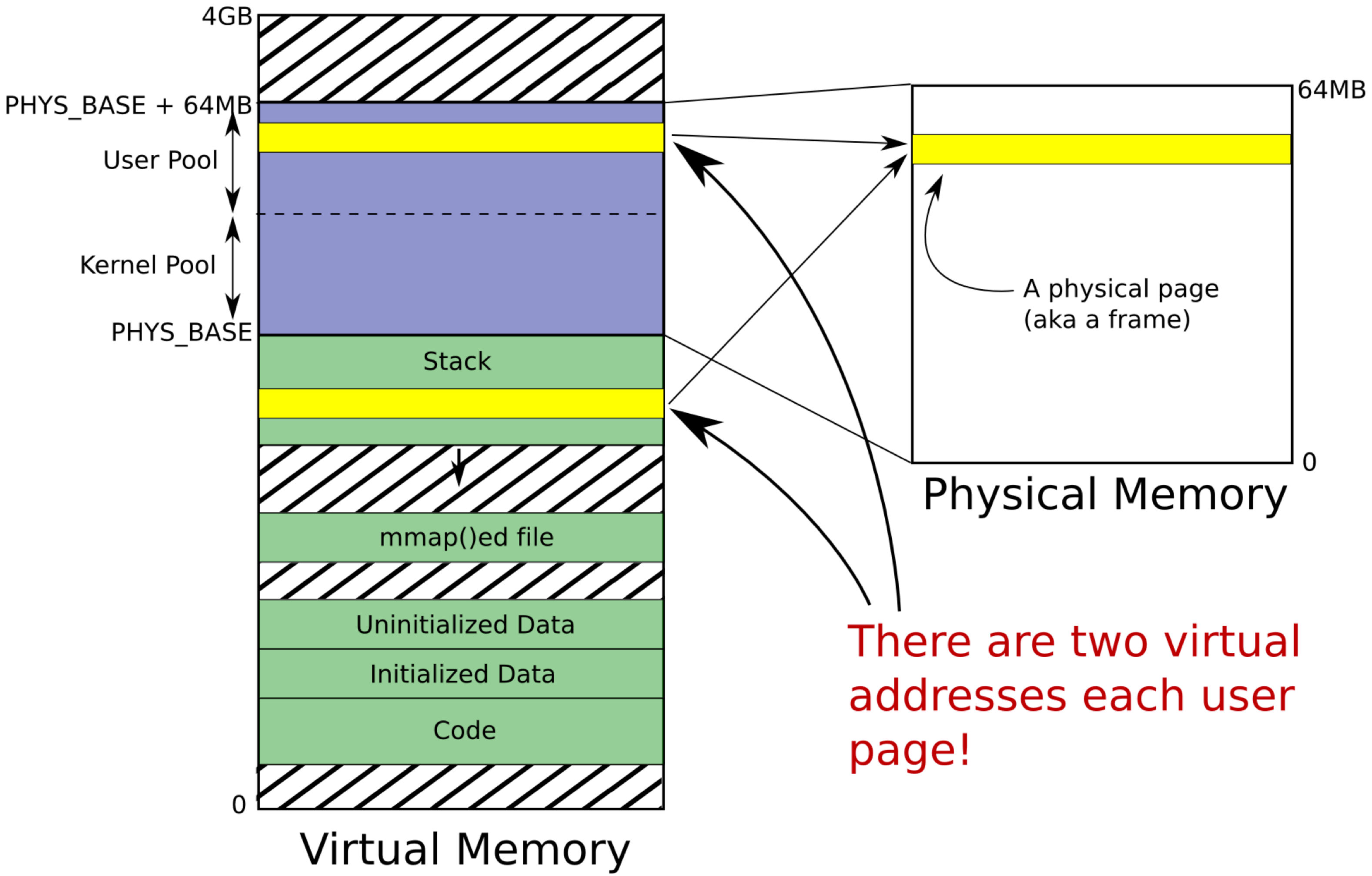
Slides adapted from previous quarters

Overview: Project Components

- High Level Goal: Implement Virtual Memory
- Page Table Management
    - Page Fault Handler
        - Interacts with most of the rest of the system
    - Eviction Policy
        - Approximation of LRU (e.g. “clock algorithm”)
- Swap Disk Management
    - Page{in,out} evicted pages
- Stack Growth
- Memory Mapped Files (including lazy loading executables)

First: Memory Layout

- All user virtual addresses have corresponding kernel virtual addresses
- The hardware page table (discussed in lecture) allows the MMU to map addresses automatically
- You will need to keep around additional information
    - e.g. keep track of where to fetch pages that aren’t in memory
    - Explained in a few more slides



There are two virtual addresses each user page!

Pintos Page Tables

- Base implementation already creates basic page directory & page table structure mappings. (Look at paging_init() in init.c)
- e.g. user wants a new upage at vaddr:
  - palloc_get_page(PAL_USER) returns a page
  - Register with pagedir_set_page()
- Layout is very similar to the one discussed in lecture, except:
  - Only two levels of page tables
  - 4 byte page table entries
- Note that hardware bits (present, writable, user, accessed, dirty) are PER PAGE TABLE ENTRY, not per physical frame!

Handling Virtual Memory

- Page Fault Handler
  - Verify that the access is legal (read/write)
  - If page has been evicted, load from disk/swap
  - Else, trigger stack growth if necessary
  - Paging something in might require paging something else out
  - Beware of race conditions while accessing data structures
  - You may need to "pin" pages to prevent them from being evicted
    - Don't pin for too long, but
    - Don't worry about running out of pages to pin (you can panic the kernel)
- Global Data Structure: Frame Table
  - Keeps track of the *physical frames* that are allocated/free
- Per-Process Data Structure: Supplemental Page Table
  - Keeps track of supplemental data about each page
  - i.e. location of data (frame/disk/swap); pointer to corresponding kernel virtual address; list of aliases, etc.

Stack Growth

- As part of the page fault handler, devise a heuristic to identify valid stack accesses that require stack growth
- Valid stack accesses may cause the following page faults:
    - PUSH: 4 bytes below %esp
    - PUSHA: 32 bytes below %esp
    - Other accesses: anywhere between %esp and PHYS_BASE
        - Accesses to stack pages that have been paged out
        - Accesses above the stack pointer after %esp has been decremented (i.e. with SUB $n %esp)
- All pages on the stack (except the first one) must be allocated lazily
- You should limit stack size to a constant as most OSes do
    - Why is this a good idea?

Memory Mapped Files

- Implement the mmap() and munmap() system calls
- Per-Process Data Structure: mmap Page Table
    - Keeps track of file -> page mappings for the process
- mmap()ed pages must be loaded from disk lazily
- mmap() should fail (return -1) if:
    - The size of the file is zero bytes
    - The range of pages mapped overlaps with existing mapped pages
- All mappings are implicitly unmapped when a process exits

Page Replacement Mechanism

- Eviction Policy
  - Approximation of LRU replacement (e.g. “clock” algorithm)
  - Pages that can be deallocated without writing to swap/disk
    - Read only (code) pages
    - Clean data pages
    - Clean mmap()ed pages
  - Pages that must be written out to swap
    - Stack pages
    - Dirty data pages
  - Write dirty mmap()ed pages to their corresponding file on disk
- Global Data Structure: Swap Table
  - Keeps track of the *swap slots* that are allocated/free

Implementation Order

Frame Table: allocate frames (physical pages)

Supplemental Page Table: handle page faults

Stack growth, mmap()ed files, lazy executable loading [work in parallel]

Eviction/paging

But these are all interdependent! It WON'T work to design them separately and try to piece everything together later!

Useful Code

- Page Allocator (threads/palloc.{h,c})
  - Allows you to obtain a virtual address for a physical frame
  - Pages can be allocated from either the user pool or the kernel pool
- Page Directory (userprog/pagedir.{h,c}, threads/pte.h)
  - Interface to the x86 hardware page directory
  - Allows you to perform operations on the hardware page tables for each process
  - pte.h gives access to hardware bits (p/w/u/a/d)
- General data structures
  - Hash Table (lib/kernel/hash.{h,c})
  - Bitmap (lib/kernel/bitmap.{h,c})
  - List (lib/kernel/list.{h,c}) [you should already know this one!]
- Refer to Appendix A of the Pintos Reference Guide for a more elaborate description of the available APIs

No Extra Credit

- In previous years there was extra credit available for this assignment
- There is NO extra credit being offered for this assignment (or for assignment 4) despite what it says in the assignment handout!

Good Luck!

- Any Questions?