Precept 5: Virtual Memory

COS 318: Fall 2020

Project 5 Schedule

- **See website**

Project 5 Overview

- **Goal:** Add memory management + virtual memory support to the kernel
- Read the project spec for more details
- Starter code can be found on the lab machines (/u/318/code/project5)
- Start early

Project 5 Overview

- Add demand-paged VMM + restrict user processes from kernel level privileges
- Need to implement:
  - Virtual address spaces for user processes
  - Page allocation
  - Paging to / from disk
  - Page fault handler

Implementation Checklist

- **memory.h**
  - page_map_entry_t
- **memory.c**
  - page_addr()
  - page_alloc()
  - init_mem()
  - setup_page_table()
  - page_fault_handler()
  - page_swap_in()
  - page_replacement_policy()
  - page_swap_out()

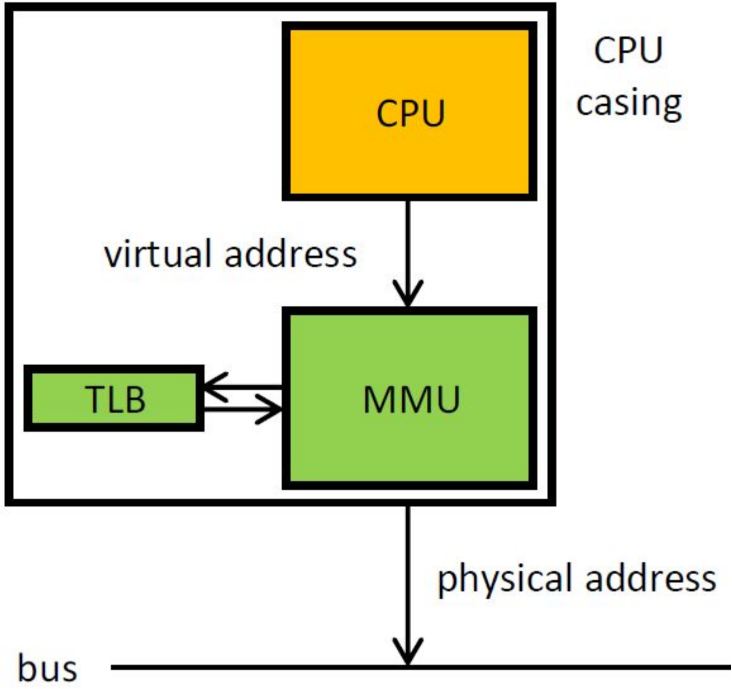
Big Picture

- Set up kernel memory
- Set up VA to PA mapping for each process on creation
  - Processes now run in virtual memory
  - Hardware uses mapping when executing instructions
- Implement the page fault handler
  - If virtual page not in memory, page it in from disk and map it to a physical page

Address Translation Review

VA to PA Translation: Overview

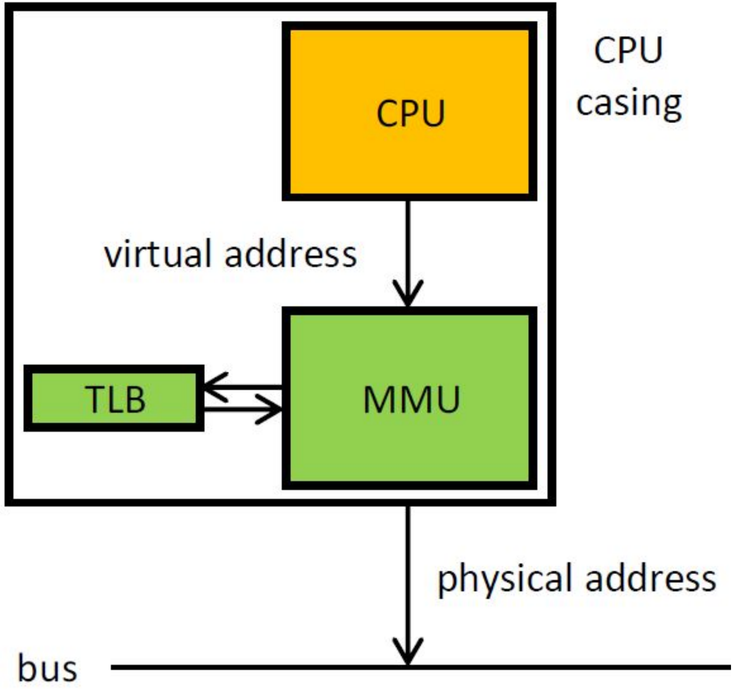
- All addresses are virtual
  => must go through MMU
- MMU checks TLB first
- On miss: performs translation using page tables
- [Image Source]



CPU: Central Processing Unit
MMU: Memory Management Unit
TLB: Translation lookaside buffer

VA to PA Translation: Overview

- Page tables defined in software
- Use CR3 register to find root page table in RAM
- Checks page permissions - faults if invalid
- [Image Source]



CPU: Central Processing Unit
MMU: Memory Management Unit
TLB: Translation lookaside buffer

Paging System: Linear to Physical

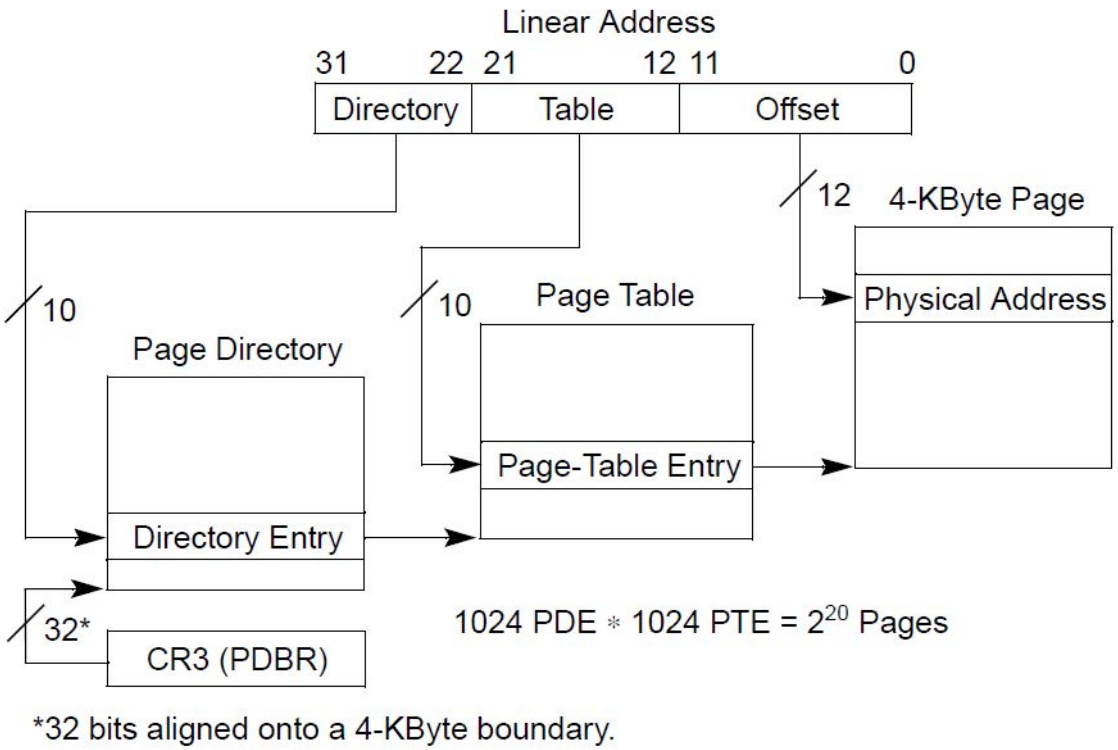


Image Source

Paging System: Dir. / Table Entries

- Hierarchical System:
  - Directory Entries hold page table start address
  - Table Entries hold page start address
  - Page start address + offset = Physical address

Paging System: Dir. / Table Entries

- Dirs and Tables must fit onto a 4KB page!
  - Therefore, the lower 12 bits of the start address are always 0
- Higher 20 bits hold start address, lower 12 bits store permissions / status

Paging System: Directory Entries

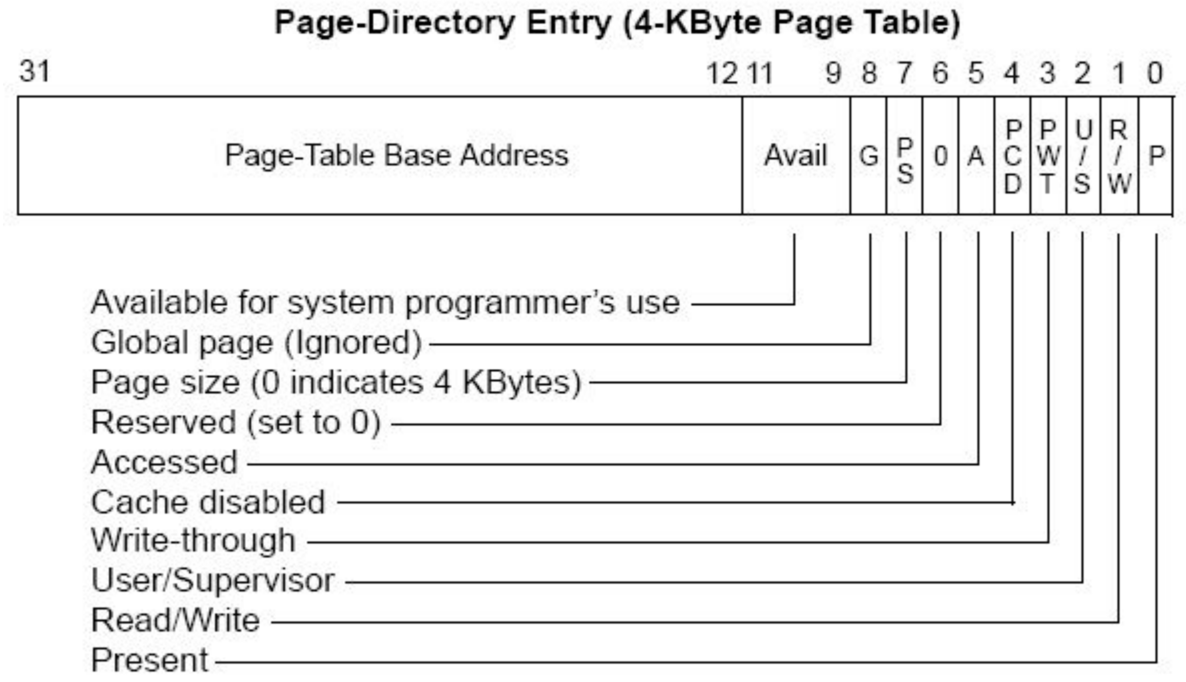


Image Source

Paging System: Table Entries

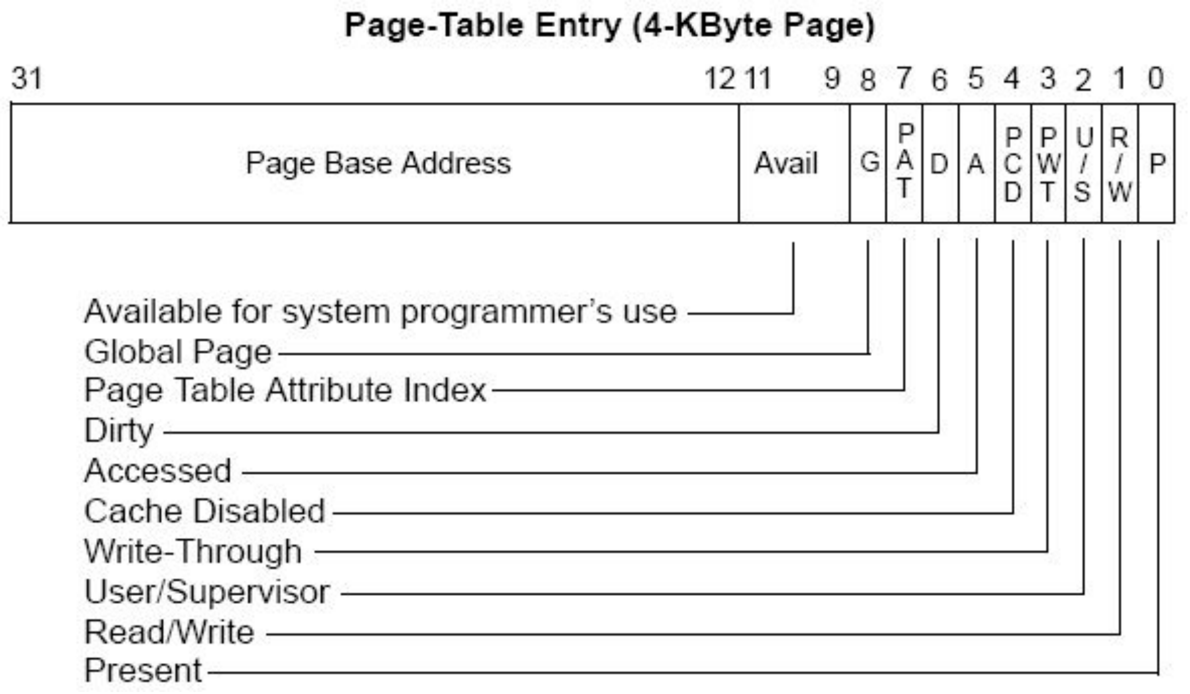


Image Source

Paging System: VA Structure

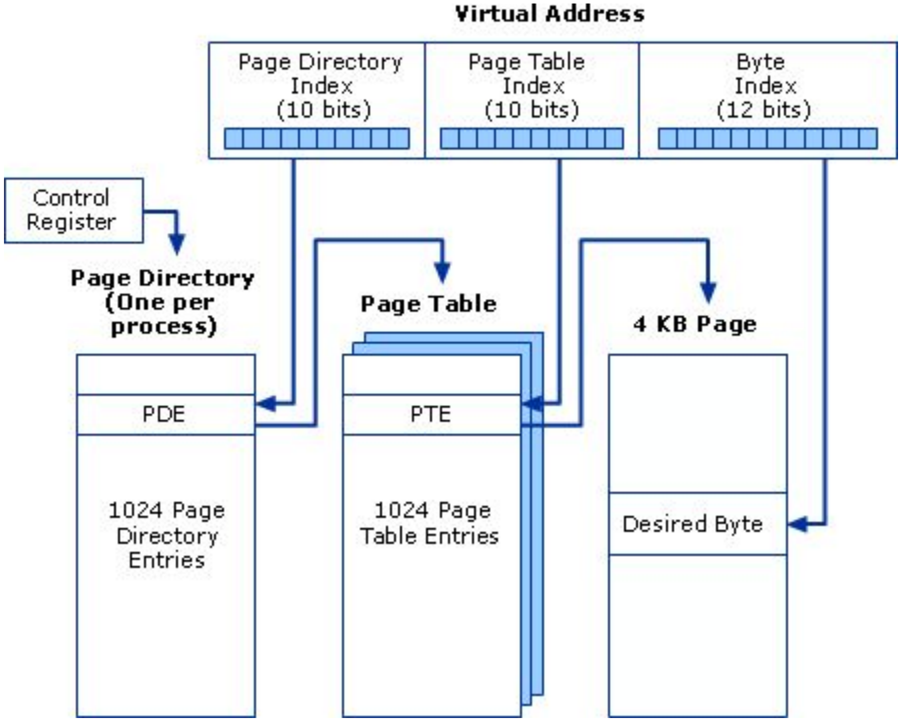


Image Source

Check: VA Space = Paging Space

- We use 32-bit (4-byte) VAs, 4KB pages, and a two level page table system
    - 4KB per page / 4 bytes per entry = 1K entries
- 2^10 (p.d.e) * 2^10 (p.t.e) * 2^12 (bytes per page) = 2^32 addressable bytes
- 32 bits can address 2^32 locations

Project Description

Initializing Kernel Memory

- Allocate page directory
- Allocate N_KERNEL_PTS (page tables)
- For each page table, “allocate” pages until you reach MAX_PHYSICAL_MEMORY
- **physical addr. = virtual addr.** for the kernel
- Set the correct flags (i.e. give user the permission to use the memory pages associated with the screen)

Initializing User Memory

- User processes need four types of pages (page directory, page table, stack page table, and stack pages)
- PROCESS_START (virtual addr. of code + data):
  - Use one page table and set entries relative to process address space
  - Each process needs pcb->swap_size memory
- PROCESS_STACK (virtual addr. of top of stack):
  - Allocate N_PROCESS_STACK_PAGES for each process

Page Faults

- A page fault occurs when we access a physical page frame that is not mapped into the virtual address space of the user process
- How does the hardware know that a page fault occurred?
- Keep track of metadata of physical page frames:
  - Free or not?
  - Information to implement a replacement algorithm (FIFO is sufficient)
  - Pinned or not? When would you want to pin a physical page frame?

Page Faults

- You need to write `page_fault_handler()`:
  - Find the faulting page in the page directory and page table
  - Allocate a page frame of physical memory
  - Load the contents of the page from the appropriate swap location on the USB disk (think about how to figure out the swap location)
  - Update the page table of the process

Paging From Disk

- To resolve a page fault, you might need to evict the contents of a physical page frame to disk
- Use a USB disk image for swap storage (usb/scsi.h)
- Use scsi_write() and scsi_read(), which have already been implemented
- Assume that processes do not change size (no dynamic memory allocation)
- Update page tables
- Decide if you need to flush TLB

Tips + Other Notes

Some Tips

- One page table is enough for process memory space
- Some functions (i.e. page fault handler) can be interrupted
    - Use synchronization primitives!
- Some pages don’t need to be swapped out
    - Kernel pages, process page directory, page tables, stack page tables, and stack pages

Some Tips

- Test first with kernel threads
    - Implement `page_addr()`
    - Partially implement `page_alloc()` (assume number of pages is smaller than PAGEABLE_PAGES)
    - Implement `init_memory()`
    - Partially implement `setup_page_table()` (kernel threads only)
    - Comment out the loader thread in kernel.c and fix the value of NUM_THREADS in kernel.h

Some Tips

- After kernel threads are working
    - Finish the implementation of `setup_page_table()` (deal with processes)
    - Implement `page_fault_handler()`
    - Implement `page_swap_in()`
    - Uncomment the loader thread in kernel.c
- You should see a command shell on the screen

Some Tips

- After the shell is working
  - Finish the implementation of `page_alloc()`
  - Implement `page_replacement_policy()`
  - Implement `page_swap_out()`
- Use the provided `bochs` executable in /u/318/code/project5/bin and NOT in /u/318/bin for testing

`bochs-gdb` vs `bochsdbg`

- `bochsdbg` does not work on this assignment!
- Use `bochs-gdb` instead:
    - Uncomment line 9 in `bochsrc` (set port to free value)
    - Run `bochs-gdb`, then `gdb` in another window
    - Run `target remote localhost:<port>`
    - Run `file kernel`, then `break kernel_start` (up to you)
    - Continue, then debug with standard gdb commands

Design Review

Page Table + Page Faults

Explain how virtual addresses are translated to physical addresses on i386. When are page faults triggered? How are you going to figure out what address a fault occurred on?

Page Map

You're going to need a data structure to track information about pages. What information should you track?

Calling Relationships

For the functions `page_alloc`, `page_swap_in`, `page_swap_out`, and `page_fault_handler`, please describe the caller-callee relationship graph

Questions?