Project 5
Virtual Memory

COS 318

Fall 2015

Project 5: Virtual Memory

- Goal: Add memory management and support for virtual memory to the kernel.
- Read the project spec for the details.
- Get a fresh copy of the start code from the lab machines (/u/318/code/project5/).
- Start as early as you can and get as much done as possible by the design review.

Project 5: Schedule

- Design Review:
  - Monday 12/07
  - Sign up on the project page;
  - Please, draw pictures and write your idea.
- Due date: Wednesday, 12/16, 11:55pm.

Project 5: Overview

- You will extended the provided kernel with a demand-paged virtual memory manager and restrict user processes to user mode privileges (ring 3) instead of kernel mode privileges (ring 0). You will implement:
  - virtual address spaces for user processes;
  - page allocation;
  - paging to and from disk;
  - page fault handler.

Design Review

- Design Review:
  - Explain how virtual addresses are translated to physical addresses on i386.
    - ✧ When are page faults triggered?
    - ✧ How are you going to figure out which address caused a fault?
  - You will need a data structure to track information about pages.
    - ✧ What information should you track?
  - For the functions `page_alloc`, `page_swap_in`, `page_swap_out`, and `page_fault_handler`, please describe the caller-callee relationship graph.

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()`

General Notes

- Familiarize yourself with the 2-level page description of i386.
  - Read sections 3.7.1, 3.7.6, and 4.2 of the Intel manual, linked off project website.
- Make sure that you understand the new PCB structure in kernel.h.
- Look at interrupt.c:exception_14() to understand how a page fault is initially handled.
- Testing is tricky. A few hints later.

Big Picture

- Set up memory for the kernel.
- Set up virtual memory for each process: done in the kernel when you create a new process.
  - Each process now runs in virtual memory;
  - Mapping virtual memory to physical memory is now responsibility of the kernel;
  - Hardware uses the mapping when instructions are actually executed.
- Implement the page_fault_handler() in the kernel:
  - If a virtual page is not in memory, the kernel pages it in from disk, and maps it to a physical page;
  - Physical page frames are static;
  - Virtual pages are moved between physical memory and disk.

Virtual-to-Physical Mapping

- A linear address is divided into three sections:
  - (Level 1) Page-directory entry: bits 22 to 31 provide an offset to an entry in the page directory. The selected entry provides the base physical address of a page table.
  - (Level 2) Page-table entry: bits 12 to 21 of the linear address provide an offset to an entry in the selected page table. This entry provides the base physical address of a page in physical memory.
  - Page offset: bits 0 to 11 provides an offset to a physical address in the page.

Virtual-to-Physical Mapping

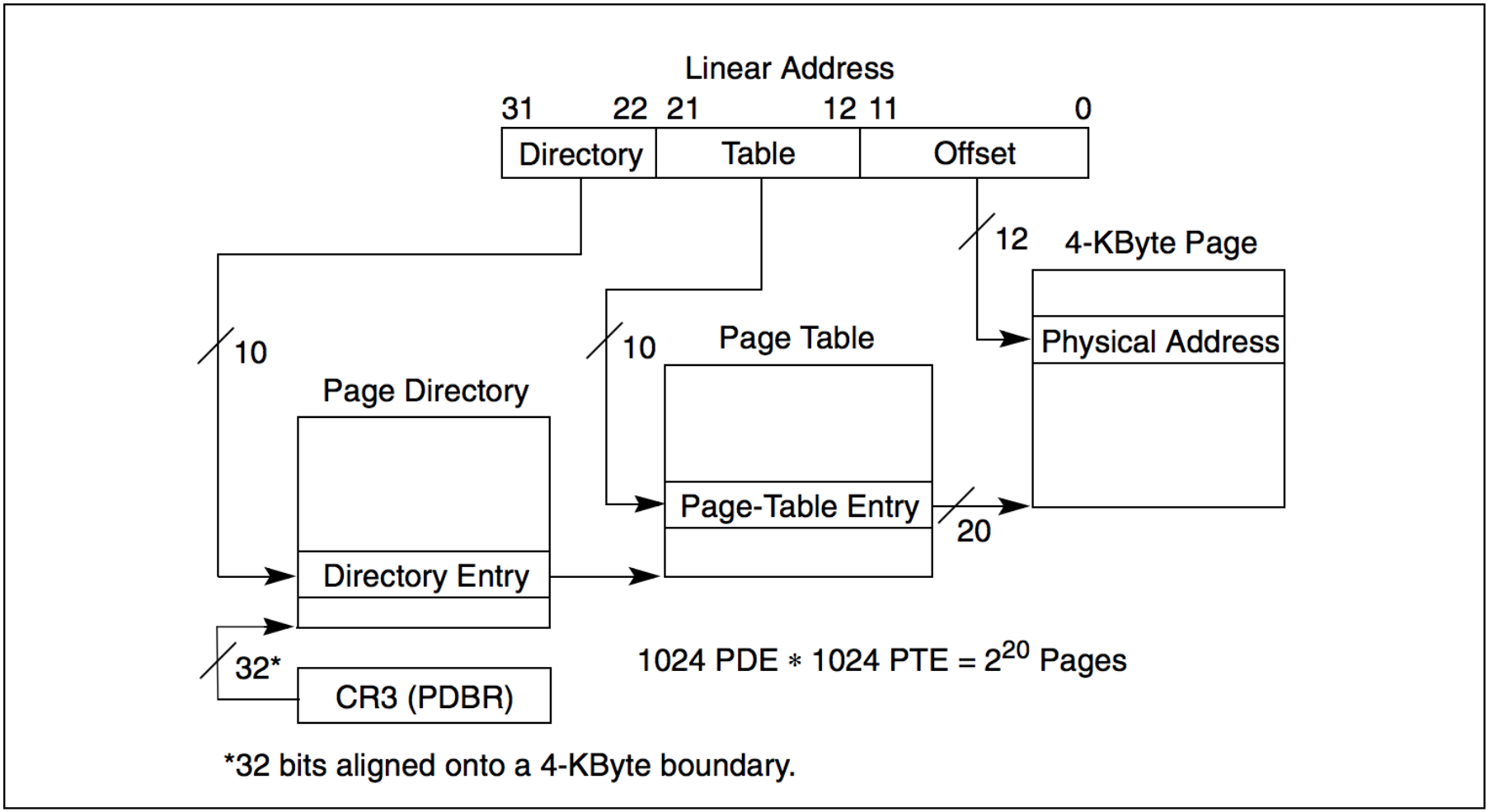
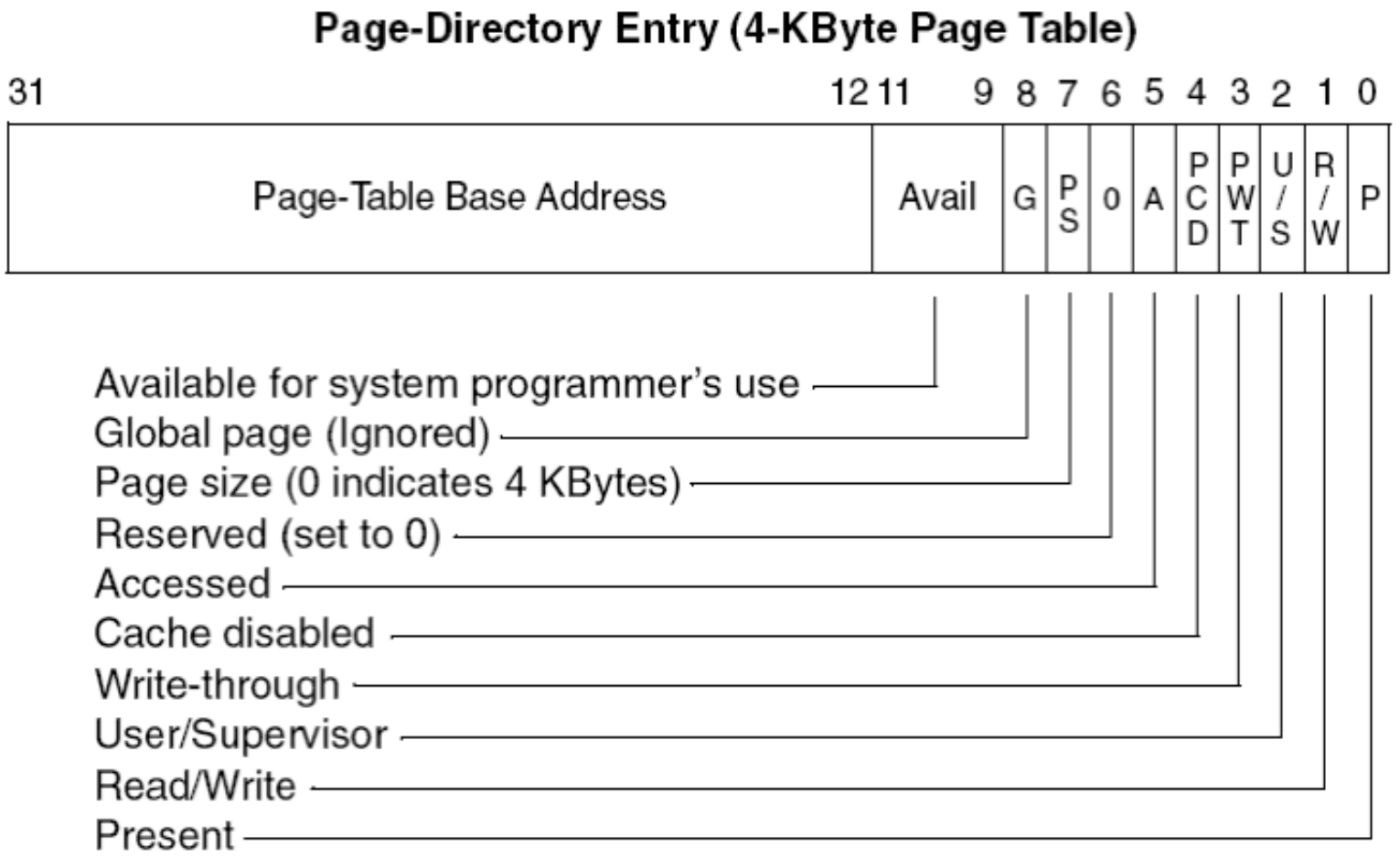
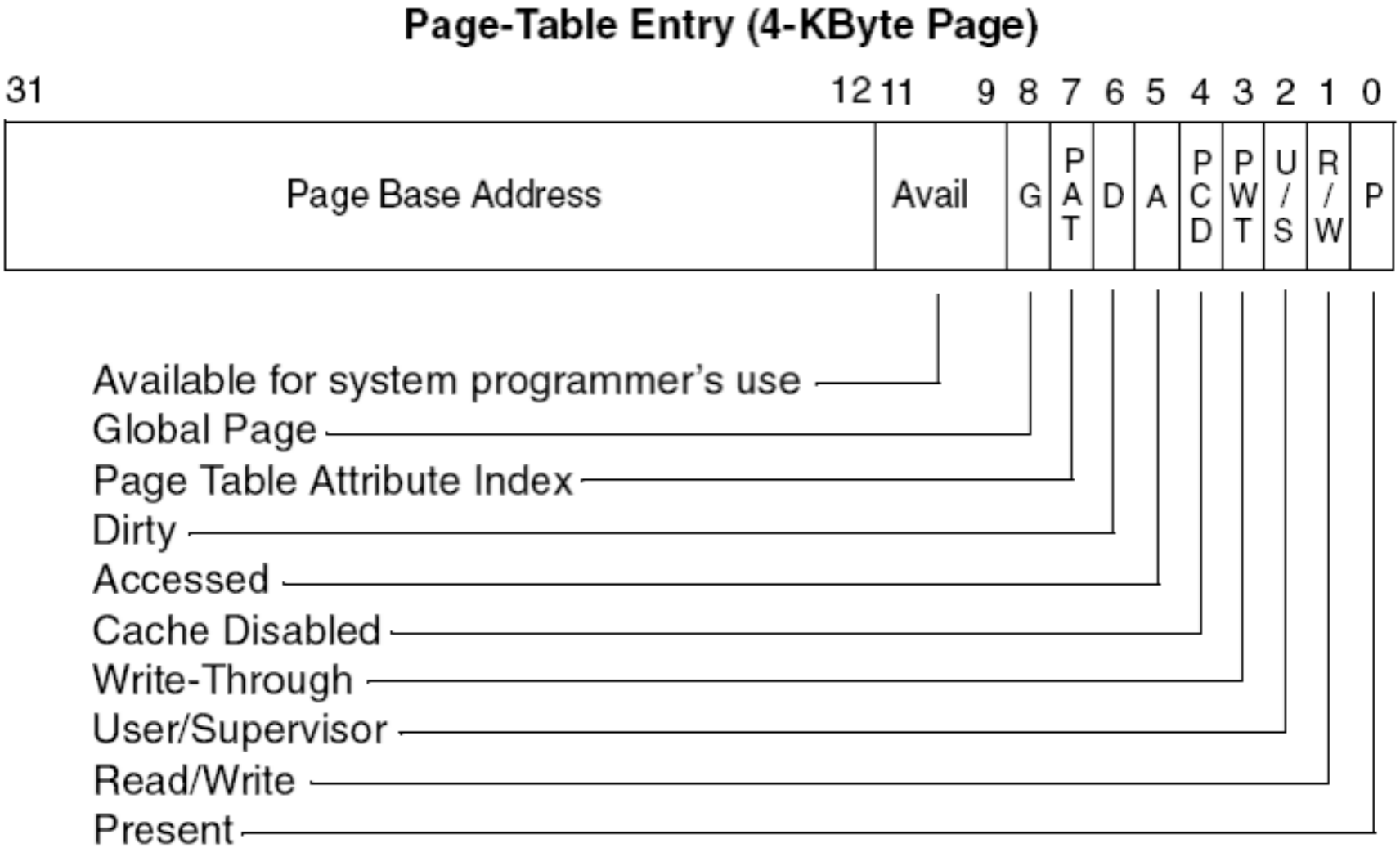


Figure 3-12. Linear Address Translation (4-KByte Pages)

Directory Entry



Page Entry


Page-Table Entry (4-KByte Page)
31
12 11
9 8 7 6 5 4 3 2 1 0
Page Base Address
Avail
G
PAT
D
A
PCD
PWT
U/S
R/W
P
Available for system programmer's use
Global Page
Page Table Attribute Index
Dirty
Accessed
Cache Disabled
Write-Through
User/Supervisor
Read/Write
Present

Initializing Kernel Memory

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

Setting up Process Memory

- Processes need four types of pages:
  - Page directory;
  - Page tables;
  - Stack page table;
  - Stack pages.
- PROCESS_START (virtual address of code + data):
  - Use one page table and set the entries relative to the process address space as not present (let demand paging work when needed);
  - Process needs pcb->swap_size memory.
- PROCESS_STACK (vaddr of stack top)
  - Allocate N_PROCESS_STACK_PAGES for each process.

Page Faults

- A page fault happens because the virtual page is not resident on a physical page frame.
- How does the hardware know that a page fault happened?
- You need to keep track of metada of physical page frames:
  - Free or not?
  - Information to implement a replacement policy (FIFO is sufficient for this assignment);
  - Pinned? 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 (How are you going to figure out the swap location?);
  - Update the page table of the process.

Paging from disk

- To resolve a page fault, you might have to evict contents of a physical page frame to disk:
  - Might need to save the content of the physical page frame;
  - Bring in contents of virtual page, which is on the disk, and copy contents into the physical page frame.
- Use a USB disk image for swap storage (usb/scsi.h).
  - Just use `scsi_write()` and `scsi_read()`.
- Assume that processes do not change size (no dynamic memory allocation).
- Update page tables.
- Decide if you need to flush TLB.

Some Tips

- One page table is enough for a process memory space (code+data).
- Some functions (esp. the 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.

Implementation Hints

- Use `bochs-gdb` to debug (you will not be able to use `bochsdbg`).
  - Uncomment Line 9 of `bochsrc`.
- Start `bochs-gdb` and then `gdb`.
- On gdb, type **target remote localhost:1234**
- Use gdb commands to set breakpoints, step, continue, etc.
- gdb with emacs is very helpful (you can see the source code while debugging).

Implementation and Testing Hints

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

Implementation and Testing Hints

- After the 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.

Implementation and Testing Hints

- After the shell is working:
  - Finish the implementation of `page_alloc()`.
  - Implement `page_replacement_policy()`.
  - Implement `page_swap_out()`.

Extra Credit

- Implement the FIFO with second chance paging algorithm (2 pts).
- Implement the Not Recently Used (NRU) page replacement algorithm (2 pts).