Project 5: Virtual Memory

INF-2201 Staff

Spring 2024

Contents

Project 5

Project Overview

Project 5: Administrative Info (mostly like the others)

- Mandatory assignment
- Groups of two
- Design review
- Hand-in
  - Hand-in via Canvas

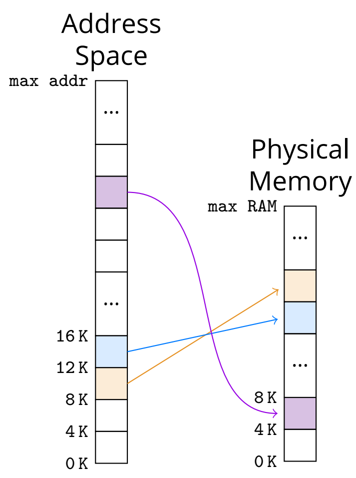

Figure 1: Mapping from virtual address space to physical memory

  - Code: whole repository (working tree + `.git/` dir)
  - Report: place in a `report/` directory in your repo
  - Zip up and upload to Canvas
  - **New!-ish** Also upload the report PDF separately

Project 5: Virtual Memory

- In Project 4:
  - We started enforcing user/kernel separation
  - Processes had separate address spaces
  - Virtual memory turned on, but very basic
- In Project 5:
  - We will improve the virtual memory
  - Swap memory pages on demand
- You will:
  - Manage virtual memory with dynamic swapping
  - Manage page tables
  - Implement a page fault handler
  - Implement page allocation scheme
  - Swap to/from disk

Virtual Memory

Virtual Memory Overview

- Virtual memory
  - Add indirection!
  - Map virtual addresses $\rightarrow$ physical memory

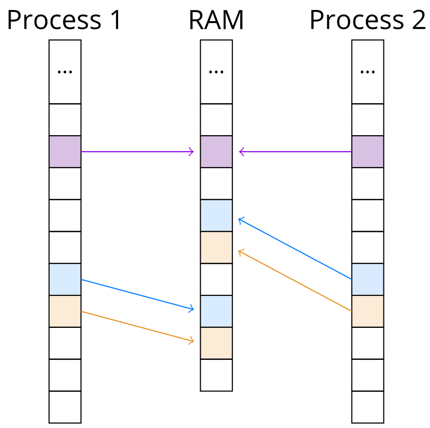

Figure 2: Processes with separate address spaces

- Divide memory into *pages*
  - Common size: 4 KiB
  - 32-bit address space:
    4 GiB space = 4 KiB/page $\times$ 2^{20} pages/space
- Benefits
  - Use whole address space!
  - Swap pages to disk as needed!

Multiple Address Spaces

- Each process can have its own address space
- Great way to enforce process separation
  - Can't tamper with what you can't address
  - Can have per-page permissions
- But can map some pages into multiple spaces
  - Common to share kernel pages

x86-32 Page Mapping

x86-32 Page Mapping: More Info

- To map a 32-bit address space:
  - 4 GiB space = 4 KiB/page $\times$ 2^{20} pages/space
  - 20 bits: can select page in space
  - 12 bits: can select byte in page
- Two-level hierarchy
  - Level 2: Page Directory
  - Level 1: Page Table

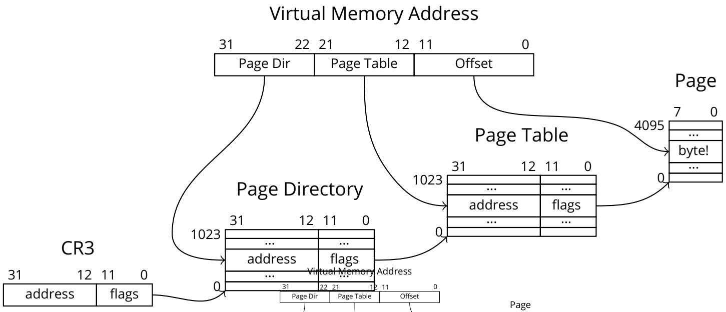

Figure 3: Page mapping on x86-32

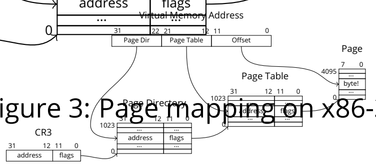

Figure 4: Page mapping on x86-32

- Level 0: Physical Page

- Each step in hierarchy is also a page
  - 4 KiB page = 32 bits/entry $\times$ 1024 entries/page
  - 20 bits: phys addr of next page in hierarchy
  - 12 bits: flags
- CR3 $\rightarrow$ 1 page dir $\rightarrow$ 1024 tables $\rightarrow$ 2^{20} pages

x86-32 Page Mapping: MORE Info

- CR3 $\rightarrow$ 1 page dir $\rightarrow$ 1024 tables $\rightarrow$ 2^{20} pages
  - Up to 1025 pages of metadata per space
  - Just over 4 MiB
- In practice, rarely need all 4 MiB
  - Rare to use entire address space
  - Don't need page tables for unused regions
  - Can reuse page tables for shared mappings (e.g. kernel)
- Most important entry flag: P (Present)
  - Directory entry's P not set? No table
  - Table entry's P not set? No page
  - Try to access not-present page? Page Fault
- Page Fault (#PF, exception 14)
  - Handler can load missing page, then resume
  - This is swapping

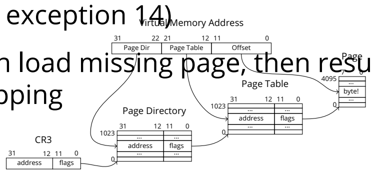

Figure 5: Page mapping on x86-32

Project Tasks

Tasks/Precode Overview

- You will:
  - Manage virtual memory with dynamic swapping
  - Manage page tables
  - Implement a page fault handler
  - Implement page allocation scheme
  - Swap to/from disk
- Precode infrastructure:
  - Handles saving/restoring CR3 on task switch
  - Installs hooks to fns in `memory.c`:
    * `void init_memory(void)` — Called on kernel start
    * `void setup_process_vmem(pcb_t *p)` — Called on process creation
    * `void page_fault_handler(...)` — Called on page fault
- `memory.c` has been hollowed out
  - Only low-level helper fns remain
  - You must to implement the above hooks and mid-level support fns

Task: Set Up New Process's Address Space

- You may need to…
  - Allocate a physical page
  - Identity map all kernel pages
  - Allocate and "pin" physical pages for page tables if needed
  - Set user access flag for the video buffer page (0xb 8000)
  - Mark all process pages as not present, so they will be loaded on demand
  - Allocate and "pin" a page for user stack
- What do we mean by "pin"?
  - Certain pages are important enough to not swap out
  - You may want to create a "pinned" flag for pages you don't want to swap
- Where do you mark pages as pinned?
  - That's up to you
  - You'll need to track more than the what CPU data structures hold
  - You'll need your own additional data structures

Task: Handle Page Faults (Swap)

- On page fault:
  - CPU stores fault address in CR2

- CPU creates standard exception stack frame (SS, ESP, EFLAGS, CS, EIP, error code)
- Page fault error code contains flags with info about fault
- CPU invokes Page Fault (#PF, exc 14) handler

• Handlers:
  - Low-level handler: `page_fault_handler_entry(...)` [interrupt_asm.S, given]
  - Calls C handler: `page_fault_handler(...)` [memory.c, up to you]

• You may need to…
  - Check errror code flags
  - Get fault address from CR2
  - Swap in missing page
    * Allocate a physical page
    * Swap out an old page if necessary (write to disk if dirty)
    * Read needed page from disk
    * Invalidate page cache for that page

Subtask: Physical Page Allocation

• Remnants of Project 4's page allocation are intact in memory.c
  - You can reuse and enhance them

• If pages are free, simply use a free page and mark it as allocated

• If pages are not free, possible eviction algorithm:

```
loop:
    pick a random page
    if page is not pinned:
        unmap physical page from address space (mark not present)
        if dirty: write page back to disk
        return page for use
```

• Design review Q: What info will you track for allocated pages?

• When swapping out, you can simply write back into the image
  - This will affect the state of the process in the image on reboot
  - Extra challenge: create a separate dedicated swap area

Extra Challenges

• Implement a better page eviction strategy
  - FIFO?
  - FIFO with second chance?
  - Other?
  - See textbook for common algorithms

• Create a dedicated swap area on disk

  - Write to swap area instead of overwriting process on disk

Hints

- Be sure to `invalidate_page(...)` [cpu_x86.h] when updating mappings
  - Design review Q: When exactly do you need to do this?
- Double-check your bit operations
- Triple-check your protection bits
- Use given helper functions and create your own helper functions
- Use serial I/O for debugging
- Beware: the page fault handler itself can be interrupted by IRQ

Administrative Details

Procedure is (Mostly) the Same as Other Projects

- Design reviews start on Monday (!!)
- Code
- Report
- Hand in via Canvas: zip up entire repository plus report
- **New!-ish** Also upload the report PDF separately

Report

- Structure: scientific paper
- Length: around 4 pages
- Citations: required
  - You must cite sources you use
  - At the very least, you should have the textbook as a source
- File format: PDF
- AI tools: discouraged but not banned. Use must be declared
- **New!-ish** See Howto doc and template in repository
  - `doc/how-to-write-a-report.md`: gives more guidelines and advice
  - `report/latex-src/template.tex`: LaTeX tutorial / template
  - If you use the template, be sure to remove all existing content

Hand In via Canvas

- Put your report in your repository, under a `report/` dir
  - Report shoud be a PDF format

- If you write in Word or other WYSIWYG word processor, export to PDF

- If you write in a document prep system like Markdown or LaTeX, you can include the report source if you like, but it's not required.
- Zip up your entire repository (code tree + report + `.git/` dir)
- Submit via Canvas
  - Upload zip
  - **New!-ish** Also upload report PDF
  - Having both makes it easier for us to grade