Project 5

Virtual Memory

Goals

- Two-level page tables
  - Setup Page Directory & Page Tables
  - Read Soft. Devel. Manual Vol. 3 (Ch 2-4)
- Page fault handler
  - Allocate physical page and bring in virtual page
- Physical page frame management
  - page allocation & replacement
  - swap in & out
Two Level Virtual Memory

CR3

PDB Flags

32 12 0

Page Directory

Entry

Virtual address

dir table offset

32 22 12 0

Page table

Entry

Physical address

Base Addr. offset

32 12 0

In Words…

- MMU uses CR3 and the first 10 bits of the virtual addr to index into the page directory and find the physical address of the page table we need.
- Then it uses the next 10 bits of the virtual addr to index into the page table, find the physical address of the actual page.
- The lowest 12 bits are used as the offset in the page.
Properties

- Size of one Page Directory or one Page Table or one Page is 4KB ($2^{12}$)
- Page Directory is a Page Table for the Page Tables
  - Avoids million entry page tables
- Each Entry is 4 bytes (32 bits)
  - So one page can have $2^{10}$ entries!
- Each page must be page aligned

Two-Level Page Tables (cont’d)

Page table entry

<table>
<thead>
<tr>
<th>Base Address</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Dirty
- Accessed
- User/Supervisor
- Read/Write
- Present/Absent
Protection bits

- Present Bit (P)
  - If 1, then physical page is in memory
  - If 0, then other bits can be used to provide information to help the OS bring in the page
- Read/Write Bit (RW)
  - All pages can be read
  - If 1, page can be written to
- User/Supervisor Bit (US)
  - If 1, page can be accessed in kernel or user mode
  - If 0, page can only be accessed in kernel mode

How are page tables used?

- Each process (including the kernel) has its own page directory and a set of page tables.
- The address of page directory is in CR3 (page directory register) when the process is running
- On context switch pcb->page_directory is loaded into CR3
  - Done for you in the given code
How are page tables used? (cont’d)

- Don’t forget to mask off extra bits when using the base address from page table
- PE_BASE_ADDR_MASK (see memory.h)

Physical Memory Layout

- BIOS data
- Kernel code/data
- Video memory
- page0
- page1
- page2
- pageN
- free page frames

MEM_START (0x100000)

0x1000

0xB0000

next_free_page

MAX_PHY_MEM
Virtual Memory (Process) Layout

Kernel address space
- Kernel code/data, Kernel stacks, etc

User address space
- Process code/data
- Process user stack (one page, pinned in mem)

Virtual-Physical Mapping

Kernel address Space only accessible in supervisor mode (except Video mem)
- BIOS data
- Kernel code/data
- Video mem
- page0
- page1
- page2
- ...
- pageN
- free page frames

Virtual memory
- code/data accessible in user mode
- user stack (one page, pinned in mem)

Physical memory
Virtual address Mapping

- Kernel addresses are mapped to the same physical addresses (identity mapped)
- All threads share the same kernel address space
- Each process has its own address space. It must also map the kernel address space
  - Allows direct access to the video buffer

So what do we need to do?

- Setup kernel page tables that are shared by all the threads. (In init_memory())
- Setup process page tables when creating the process (In setup_page_table())
  - Note: create_thread() also calls setup_page_table
Some clarifications:

- It is OK to setup only one page table for each of the following:
  kernel, process’ data/code and process’ user-stack.
  (We assume that our data/code/stack size is not too big.)

- The page directories and page tables are themselves pages and must be allocated using page_alloc()
Setup Kernel Page Table

- Allocate and pin two physical pages: one for kernel page directory and the other for kernel page table
  - Do we need to allocate pages for kernel code/data?

Setup Kernel Page Table (cont’d)

- Set US bit for video memory area (SCREEN_ADDR in common.h)
  - User process’ require direct access
  - One page is enough
- Don’t forget to map kernel page table into kernel page directory
- For threads, just store the address of the kernel page directory into the pcb
Set up a Process’ Page Tables

- Allocate and pin four physical pages for each of the following:
  - Page directory, page table for code/data, page table for stack, and stack page
- Page Table entries in the Page Directory that point to kernel page tables should be user accessible
  - However, the kernel pages themselves should not be user accessible, except for video memory

Set up a Process’ Page Tables (cont’d)

- Map the page tables into the page directory
- Fill in the page table for code/data pages
  - Which bits should be set?
- Fill in the page table for user stack page
  - Which bits should be set here?
- Don’t forget to store the physical address of the page directory into
  - pcb->page_directory
Paging Mechanism

- After init_memory(), the kernel enables paging mode by setting CR0[PG] to one
  - Done in kernel.c
- In dispatch(), the kernel load CR3 register with current_running->page_directory
  - Done in scheduler.c

Paging Mechanism (Cont’d)

- When the physical page of a virtual address is not present in memory (the P bit is not set), the MMU hardware will trigger a page fault interrupt (int 14).
- The exception handler saves the faulting virtual address in current_running->fault_addr and then calls page_fault_handler()
  - done in interrupt.c
Page Fault Handler

- That’s what you are to implement
- Only code/data pages will incur page fault
  - all other pages (page directory, page tables, stack page) are pinned in memory
- So assume the page table is always there and go directly to find the corresponding entry for the faulting virtual address
  - You should never page fault on a page directory or page table access

Page Fault Handler (Cont’d)

- Allocate a physical page
  - Swap out another page if no free page is available
- Fill in the page_map structure
  - Discussed in more detail later
- Swap in the page from disk and map the virtual page to the physical page
  - Similar to last assignment, use USB disk as backing store
Physical Page Management—
The page_map structure

- Defined in memory.c
- An array that maintains the management information of each physical page. All physical pages are indexed by a page #
- Fields in each page_map structure
  - Pinned or not
  - The page table entry that points to this page
  - The pcb that owns the page
  - Page_aligned virtual address of the page

Page Allocation

- Implement page_alloc() in memory.c
- A simple page allocation algorithm
  
  \[
  \text{If (there is a free page)} \\
  \quad \text{allocate it} \\
  \text{Else} \\
  \quad \text{swap out a page and allocate it}
  \]
Page Allocation (Cont’d)

- How do we know whether there is a free page and where it is?
- If no free pages, which page to swap out?
  - Completely at your discretion
  - Be careful not to swap out a pinned page

Swap in and Swap out

- From where and to where?
  - The process' image is on the USB disk
  - Location and size are stored in pcb->swap_loc and pcb->swap_size
  - Note: swap_loc, swap_size is in term of sectors!
- The read()/write() utilities will be useful (usb functions)
- If the dirty bit (D bit) of the page table entry is clear, do you still need to write the page back?
Swap in and Swap out (Cont’d)

- Be careful when reading or writing
  - The images on disk are sector-aligned (512 bytes) not page-aligned (4KB)
  - Only swap in the data belonging to this process
  - Be careful not to overwrite other process’s image when swapping out
  - Example: Swapping in a page of process 1, but the page on the disk actually contains 3 sectors of process 1 followed by 5 sectors of process 2
  - Don’t forget to modify the protection bits of the corresponding page table entry after swapping in or swapping out

Swap in and Swap out (Cont’d)

- Invalidate TLB entry when swapping out a page.
  - Use invalidate_page() which is done in memory.c
- Note: we do not have different swap space for different instances of same process. When we swap a page out for a process, that page will be written to the space allocated to store that process on disk.
- So in our implementation, each process can only be started once.
Synchronization Issue

- The page map array is accessed and modified by multiple processes during setup_page_table() and page_fault_handler().
- So what should we do?

Some clarifications:

- Only the process’ code/data pages could be swapped in or out. The following pages are allocated once and pinned in memory forever:
  - Page directories, page tables, user stack pages
- It is OK not to reclaim the pages when a process exits
Summary

- You need to implement the following three functions in memory.c:
  - init_memory()
  - setup_page_table(pcb_t *),
  - page_fault_handler()
- You need also implement the following auxiliary functions and use them in the above three functions:
  - page_alloc(), page_replacement_policy(),
  - page_swap_out(), page_swap_in()
- Add whatever other auxiliary functions you need to make your code more readable

Extra Credit

- FIFO replacement policy
  - Queue structure
- FIFO with second chance
  - Use accessed bit
- You may need to modify the page_map structure we discussed here