Operating Systems
ECE344

Virtual Memory Implementation

Ashvin Goel
ECE
University of Toronto
Outline

- Managing virtual address space
- Managing physical memory
- Managing swap area
The OS virtual memory system interacts with paging hardware, physical memory and the disk.

It implements three main abstractions for managing these hardware resources:
- Per-process virtual address space management
- Physical memory management
- Swap management

These abstractions are used to implement the virtual memory hierarchy.
Address Space

- Recall that the address space is the set of memory addresses that are accessible to a process.
- OS provides each process its own independent virtual address space.
  - i.e., each process sees contiguous (virtual) memory addresses starting from 0.
- An address space is broken into regions or segments.
- Regions hold different kinds of information, e.g., stack, heap, data, text (or code).
Address Space API

- The OS must track the address space of each process
  - All regions within an address space
  - A page table that holds translations for the address space

- The address space API includes:
  - id = as_create()
    - Create empty address space, allocate new page table associated with id, with no pages mapped to frames
  - as_destroy(id)
    - Destroy address space id
    - Free page table associated with id
  - as_define_region(id, …)
    - Add a new region to an address space
Address Space API

- `as_modify_region(id, …)`
  - Change the size of a region
  - This will be needed to implement heap and stack regions
- `as_find_region(id, vaddr)`
  - Find a region, given a virtual address
- `as_load_page(id, …)`
  - Load a page in memory from file (demand paging)
- `new_id = as_copy(id)`
  - Create a full copy of the address space and the page table with same mappings
- `as_switch(id)`
  - Switch the address space to id. Mappings in the TLB from the previous address space must be removed.
Managing Physical Memory

- The OS needs to manage physical memory frames
- A **coremap** allows tracking usage of frames
- Coremap implementation can use an array of structures, one per frame
- Each struct tracks
  - Whether a frame is allocated or free (similar to bitmap)
  - For allocated frame, it tracks one (or more) address spaces and pages that map to the frame
    - Helps implement swapping of pages to disk and shared pages
Coremap API

- frame = allocate_frame(n)
  - Allocate n free contiguous memory frames, returns physical address of first frame

- free_frame(frame)
  - Free the block of frames, starting at frame

- map(id, page_nr, frame)
  - Maps frame to virtual address (page_nr) in address space id
  - Allocate and mark page table entry as valid
  - Note that a frame may be mapped to multiple pages in the same or in different address spaces (these are called shared pages)

- unmap(id, page_nr)
  - Removes frame corresponding to page_nr from address space

- evict(frame)
  - Used by swap handler to unmap all pages associated with frame
Managing Swap Area

- Dirty pages need to be placed in swap
- Use technique similar to managing memory
  - Use a bitmap or linked list of “swap frames”
- When evicting a page to swap
  - Find a free swap frame
  - Write the page to this location on the disk
  - Can use invalid PTE to note the location of the swap frame
  - On page fault, handler uses PTE to locate frame in swap area
    - See next slide
  - Next time page is swapped out, it may be written elsewhere
### Managing Swap Area

<table>
<thead>
<tr>
<th>31</th>
<th>1211</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>frame number (PFN)</td>
<td>unused</td>
<td>R</td>
</tr>
</tbody>
</table>

Page Table Entry (PTE) when virtual page is in memory:

<table>
<thead>
<tr>
<th>31</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>All zeroes</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Invalid PTE (never allocated):

<table>
<thead>
<tr>
<th>31</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index in swap area (non-zero)</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

PTE when virtual page is on swap:
Summary

- The virtual memory system of an OS is complicated because it interacts with paging hardware, physical memory and the disk.
- It implements three main abstractions for managing these hardware resources:
  - Per-process virtual address space
    - e.g., create address space, page table, VM region, load page
  - Physical memory management
    - e.g., allocate/deallocate frames, map frame to page, evict frame
  - Swap management
    - e.g., allocate/deallocate swap page
- OS implements the virtual memory hierarchy with these abstractions.
Think Time

- How is managing swap similar to managing memory?
- What is a shared frame?
- How does the OS to evict a shared frame to swap?
- How does the OS load a shared frame from swap?
Think Time Answers

- How is managing swap similar to managing memory?
  - Both require keeping track of used and free frames

- What is a shared frame?
  - When multiple pages (from the same or different processes) map to the same frame, the frame is called a shared frame (since it is shared by different pages). Later, we will see why the operating system shares frames.

- How does the OS to evict a shared frame to swap?
  - The OS uses the coremap to find all page tables entries that map to the shared frame. Then it invalidates all these entries. It allocates a swap frame, copies the frame data to the swap frame, and updates all the page table entries to map to the swap frame. This swap frame is now shared by multiple page table entries.
Think Time Answers

- How does the OS load a shared frame from swap?
  - When a fault occurs on one of the pages that map to the swap frame, then the OS needs load the swap frame into memory and update all the page table entries that map to the swap frame so that they map to the new frame in memory. This becomes much more complicated with fork(), which uses a copy-on-write implementation as described later.