CS 537
Section 7
Virtual memory

Michael Swift

Quiz # 3 Answers

- Question 1: accessing other memory
- Question 2: page tables (covered in class)
- Question 3: TLBs

Project 2 Questions

- I'd like to meet with every group for about 15 minutes this week.
  - Tell me about your design and plan
    - Break the problem into modules
    - Figure out what each module does
    - Figure out the interface
      - function calls in to the module
      - function calls out of the module
  - Write and test modules separately
    - Page table
    - Page replacement
  - Integrate them all

Implementing Page replacement

- How should you implement LRU
- How should you implement 2nd chance FIFO
- How should you implement Clock

Data structures

- What data structures do you need?
  - On every memory reference, you need to check whether a **virtual page number** is in memory
    - you probably want an array indexed by virtual page number
  - On every memory reference, you may need to:
    - update a referenced bit
    - rearrange the order of an LRU list
    - Where do these go?
  - To remove a page, you need to look at the set of physical page frames
    - to find one to remove
  - To remove a page, you need access to the virtual page to mark it invalid

Modules

- What are some modules to have in your code?
  - Clearly, page replacement policies should be separate modules
- What goes in the common code?
  - The main loop
  - Handling a hit in the page table
  - Common data structures
    - Inserting a virtual->physical mapping
    - removing a virtual->physical mapping

Linux Virtual Memory

Memory Management

- Linux's physical memory-management system deals with allocating and freeing pages, groups of pages, and small blocks of memory
- It has additional mechanisms for handling virtual memory, memory mapped into the address space of running processes
- Splits memory into 3 different **zones** due to hardware characteristics

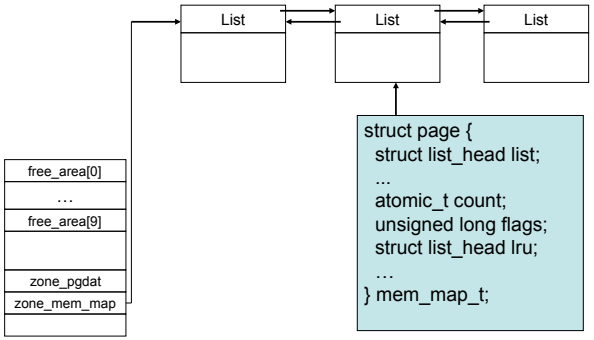
Relationship of Zones and Physical Addresses on 80x86

zone	physical memory
ZONE_DMA	< 16 MB
ZONE_NORMAL	16 .. 896 MB
ZONE_HIGHMEM	> 896 MB

3 Zones

- ZONE_DMA
  - 0-16M (i386)
  - DMA capability: some device driver need to use this memory for I/O
- ZONE_NORMAL
  - 16M-896M (i386)
  - Normal memory direct mapped by kernel
- ZONE_HIGHMEM
  - >896M (i396)
  - Not used in 64-bit architecture

Physical Pages – Page Frame DB



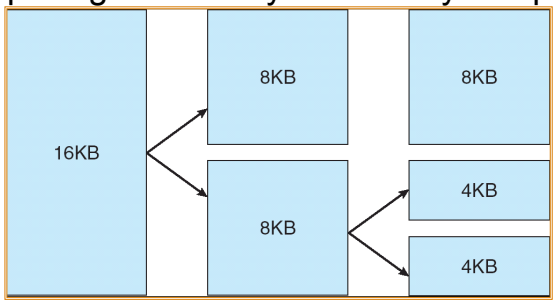
Managing Physical Memory

- The page allocator allocates and frees all physical pages; it can allocate ranges of physically-contiguous pages on request
- The allocator uses a buddy-heap algorithm to keep track of available physical pages
  - Each allocatable memory region is paired with an adjacent partner
  - Whenever two allocated partner regions are both freed up they are combined to form a larger region
  - If a small memory request cannot be satisfied by allocating an existing small free region, then a larger free region will be subdivided into two partners to satisfy the request
- Memory allocations in the Linux kernel occur either statically (drivers reserve a contiguous area of memory during system boot time) or dynamically (via the page allocator)
- Also uses **slab allocator** for kernel memory

Page Allocation

- Contiguous and non-contiguous allocation
- The Buddy System Algorithm
  - Pages are allocated in blocks which are powers of 2 in size (1, 2, 4, 8, 16, 32, 64, 128, 256, 512 pages)
  - Each size with its own free list (called free area)
  - De-allocation: if the adjacent buddy block is also free, combine to form a new free block for the next size block of pages
- Data structure: free_area_t

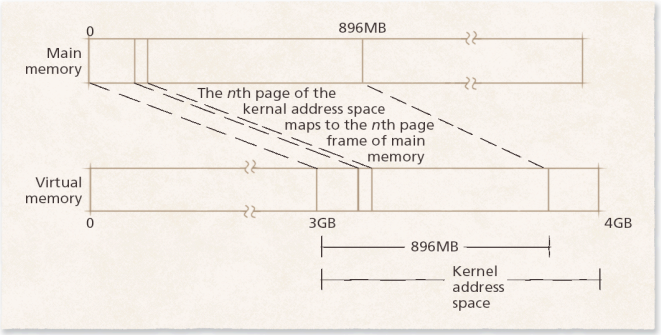
Splitting of Memory in a Buddy Heap



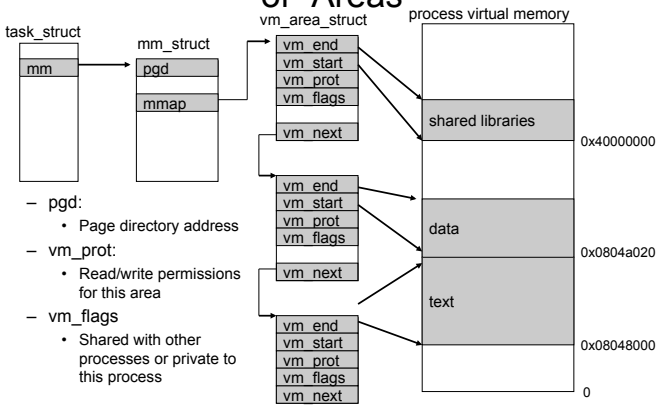
Virtual Memory (Cont.)

- The Linux kernel reserves a constant, architecture-dependent region of the virtual address space of every process for its own internal use
- This kernel virtual-memory area contains two regions:
  - A static area that contains page table references to every available physical page of memory in the system, so that there is a simple translation from physical to virtual addresses when running kernel code
  - The reminder of the reserved section is not reserved for any specific purpose; its page-table entries can be modified to point to any other areas of memory

Kernel virtual address space mapping



Linux Organizes VM as Collection of "Areas"



- pgd:
  - Page directory address
- vm_prot:
  - Read/write permissions for this area
- vm_flags
  - Shared with other processes or private to this process

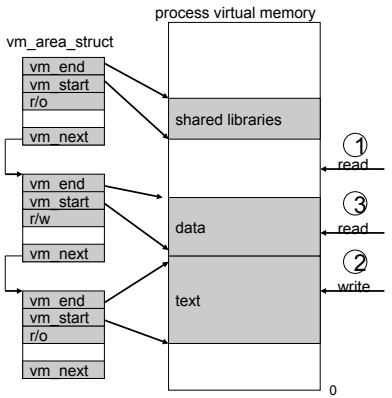
'mm_struct'

- Each application-process uses a different mapping of physical memory regions into the Pentium's "virtual" address-space, so one process can't access memory owned by another process (i.e., private regions)
- The OS switches these mapping when it suspends one process to resume another
- Each task's mapping is in its 'task_struct'

Page Fault

- Exception handler
  - Raised by address translation (hardware)
  - Call do_page_fault() to handle this interrupt
- do_page_fault()
  - Architecture-specific
  - i386: arch/i386/mm/fault.c
  - Find a page frame in the physical memory
  - Load the missing page in
  - Update the page tables

Linux Page Fault Handling



- Is the VA legal?
  - I.e. is it in an area defined by a vm_area_struct?
  - If not then signal segmentation violation (e.g. (1))
- Is the operation legal?
  - I.e., can the process read/write this area?
  - If not then signal protection violation (e.g., (2))
- If OK, handle fault
  - E.g., (3)

do_page_fault()

```
…
vma = find_vma(mm, address);
if (!vma)
        goto bad_area;
if (vma->vm_start <= address)
        goto good_area;
…
good_area:
…
handle_mm_fault(mm, vma, address, write);
…
bad_area:
…
force_sig_info(SIGSEGV, &info, tsk);
…
```

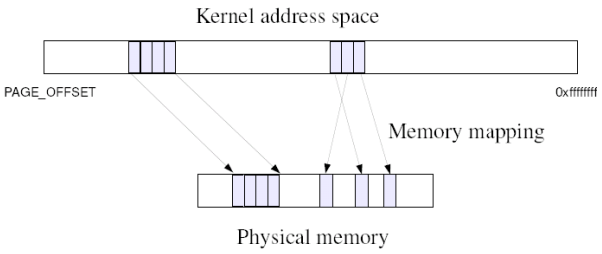
handle_mm_fault()

```
…
pgd = pgd_offset(mm, address);
pmd = pmd_alloc(mm, pgd, address);
if (pmd) {
        pte = pte_alloc(mm, pmd, address);
        if (pte)
                return handle_pte_fault(...);
}
…
```

handle_pte_fault():

- do_no_page() if pte entry is all-zero
- Do_swap_page() if pte entry is none-zero

Contiguous vs Non-Contiguous



Noncontiguous Memory

- Noncontiguous page frames in contiguous linear address
  - Not all virtual memory maps to the contiguous page frames
  - Make sense for infrequent use
- To allocate (in include/linux/vmalloc.h)
  - void *vmalloc(unsigned long size)
- To release
  - void vfree(const void * addr)

Backing Store

- Each VM area can be mapped to a file (in secondary memory)
- Explicit memory mapping through system call
  - mmap(), munmap(), mremap()
- Implicit mmaping
  - Code segment (loading from an excutable binary file)
  - Swapping (mapped to the swap file)

VM Area: Backing Store

- Data structure (in include/linux/mm.h)

```
struct vm_area_struct {
    ....
    unsigned long vm_pgoff; /* offset in page */
    struct file * vm_file; /* mapped file */
    ....
};
```

Swapping and Page Cache

- For Pages in a Process's User Space
  - Swap: Secondary Memory on the disk
  - Page Cache: Main Memory
- Data Structure: 3 sets of lists
  - Active pages, usually mapped by a process's PTE
    - active_list (in mm/page_alloc.c)
  - Inactive, unmapped, clean or dirty
    - inactive_dirty_list (in mm/page_alloc.c)
  - Clean pages, unmapped (one list per zone)
    - zone_t.inactive_clean_list (in include/linux/mmzone.h)

Kernel Swap Daemon

- Implemented as a kernel thread
  - kswapd() (in mm/vmscan.c)
  - Wake up periodically
    - Wake up more frequently if memory shortage
- Check memory and if memory is tight
  - Age pages that have not be used
  - Move pages to inactive lists
  - Write dirty pages to disk
  - Swap pages out if necessary

More kswapd()

- Call swap_out() to scan inactive page lists
  - Removes page reference from process' s page table
  - Actual swapping is done independently by file I/O
- Call refill_inactive_scan() to
  - Scan the active_list to find unused page
  - Call age_page_down() to reduce page->age count
  - If page->age is zero, move to inactive_dirty_list
- Call page_launder() to clean dirty pages:
  - Scan inactive_dirty_list for dirty pages, write to disk
  - Move clean pages to the zone' s inactive_clean_list

Demand Paging

- Page frame for a VM area is not in core
  - Page frame is not allocated when VM area is created
  - Page frame can be swapped out
- Handled by page fault