Linux Memory Management

COMS W4118
Prof. Kaustubh R. Joshi

krj@cs.columbia.edu

http://www.cs.columbia.edu/~krj/os

References: Operating Systems Concepts (9e), Understanding the Linux Kernel (3rd edition) by Bovet and Cesati, previous W4118s

Copyright notice: care has been taken to use only those web images deemed by the instructor to be in the public domain. If you see a copyrighted image on any slide and are the copyright owner, please contact the instructor. It will be removed.
Why aren’t Page Tables Sufficient?

• How to device if a memory region unallocated vs. unloaded?
  – Virtual memory areas (VMAs)

• How to manage physical memory allocation?
  – Page descriptors
  – Page allocators (e.g., buddy algorithm, SLOB, SLUB, SLAB)

• Where to read a demand fetched page from?
  – Radix trees (page_tree)

• How to identify which PTEs map a physical page when evicting?
  – Reverse mappings
  – anon vmas (anon_vma), and radix priority trees (i_mmap)

• How to unify file accesses and swapping?
  – Page Cache
Linux Memory Subsystem Outline

• Memory data structures
• Virtual Memory Areas (VMA)
• Page Mappings and Page Fault Management
• Reverse Mappings
• Page Cache and Swapping
• Physical Page Management
Linux MM Objects Glossary

- struct mm: memory descriptor (mm_types.h)
- struct vm_area_struct mmap: vma (mm_types.h)
- struct page: page descriptor (mm_types.h)
- pgd, pud, pmd, pte: pgtable entries (arch/x86/include/asm/page.h, page_32.h, pgtable.h, pgtable_32.h)
  - pgd: page global directory
  - pud page upper directory
  - pmd: page middle directory
  - pte: page table entry
- struct anon_vma: anon vma reverse map (rmap.h)
- struct prio_tree_root i_mmap: priority tree reverse map (fs.h)
- struct radix_tree_root page_tree: page cache radix tree (fs.h)
The mm_struct Structure

• Main memory descriptor
  – One per address space
  – Each task_struct has a pointer to one
  – May be shared between tasks (e.g., threads)

• Contains two main substructures
  – Memory map of virtual memory areas (vma)
  – Pointer to arch specific page tables
  – Other data, e.g., locks, reference counts, accounting information
struct mm_struct

struct mm_struct {
    struct vm_area_struct * mmap; /* list of VMAs */
    struct rb_root mm_rb;
    struct vm_area_struct * mmap_cache; /* last find_vma result */
    unsigned long mmap_base; /* base of mmap area */
    unsigned long task_size; /* size of task vm space */
    pgd_t * pgd;
    atomic_t mm_users; /* How many users with user space? */
    atomic_t mm_count; /* How many references to "struct mm_struct */
    int map_count; /* number of VMAs */
    struct rw_semaphore mmap_sem;
    spinlock_t page_table_lock; /* Protects page tables and some counters */
    unsigned long hiwater_rss; /* High-watermark of RSS usage */
    unsigned long hiwater_vm; /* High-water virtual memory usage */
    unsigned long total_vm, locked_vm, shared_vm, exec_vm;
    unsigned long stack_vm, reserved_vm, def_flags, nr_ptes;
    cpumask_t cpu_vm_mask;
    unsigned long flags; /* Must use atomic bitops to access the bits */
};
Virtual Memory Areas (vma)

Access to memory map is protected by mmap_sem read/write semaphore

Types of VMA Mappings

- File based mappings (mmap):
  - Code pages (binaries), libraries
  - Data files
  - Shared memory
  - Devices

- Anonymous mappings:
  - Stack
  - Heap
  - CoW pages

- Use different mechanisms for reverse mapping, demand fetching, swapping
Virtual Memory Areas

http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory
Anatomy of a VMA

- Pointer to start and end of region in address space (virtual addresses)
- Data structures to index vmas efficiently
- Page protection bits
- VMA protection bits/flags (superset of page bits)
- Reverse mapping data structures
- Which file this vma loaded from?
- Pointers to functions that implement vma operations
  - E.g., page fault, open, close, etc.
struct vm_area_struct

struct vm_area_struct {
    struct mm_struct *vm_mm;  /* The address space we belong to. */
    unsigned long vm_start;   /* Our start address within vm_mm. */
    unsigned long vm_end;
    struct vm_area_struct *vm_next;
    pgprot_t vm_page_prot;    /* Access permissions of this VMA. */
    unsigned long vm_flags;   /* Flags, see mm.h. */
    struct rb_node vm_rb;
    struct raw_prio_tree_node prio_tree_node;
    struct list_head anon_vma_node;  /* Serialized by anon_vma->lock */
    struct anon_vma *anon_vma;     /* Serialized by page_table_lock */
    struct vm_operations_struct *vm_ops;
    unsigned long vm_pgoff;
    struct file *vm_file;         /* File we map to (can be NULL). */
    void *vm_private_data;        /* was vm_pte (shared mem) */
};
VMA Addition and Removal

- Occurs whenever a new file is mmaped, a new shared memory segment is created, or a new section is created (e.g., library, code, heap, stack)
- Kernel tries to merge with adjacent sections
VMA Search

• VMA is very frequently accessed structure
  – Must often map virtual address to vma
  – Whenever we have a fault, mmap, etc.
  – Need efficient lookup

• Two Indexes for different uses
  – Linear linked list
    • Allows efficient traversal of entire address space
    • vma->vm_next
  – Red-black tree of vmas
    • Allows efficient search based on virtual address
    • vma->vm_rb
Efficient Search of VMAs

- Red-black trees allow $O(\lg n)$ search of vma based on virtual address
- Indexed by `vm_end` ending address
struct vm_operations_struct

struct vm_operations_struct {
    void (*open)(struct vm_area_struct * area);
    void (*close)(struct vm_area_struct * area);
    int (*fault)(struct vm_area_struct *vma, struct vm_fault *vmf);

    /* notification that a previously read-only page is about to become
     * writable, if an error is returned it will cause a SIGBUS */
    int (*page_mkwrite)(struct vm_area_struct *vma, struct page *page);

    /* called by access_process_vm when get_user_pages() fails, typically
     * for use by special VMAs that can switch between memory and hardware
     */
    int (*access)(struct vm_area_struct *vma, unsigned long addr,
                  void *buf, int len, int write);
};
Demand Fetching via Page Faults

1. Program calls brk() to grow its heap

2. brk() enlarges heap VMA. New pages are not mapped onto physical memory.

3. Program tries to access new memory. Processor page faults.

4. Kernel assigns page frame to process, creates PTE, resumes execution. Program is unaware anything happened.

http://duartes.org/gustavo/blog/post/how-the-kernel-manages-your-memory
Fault Handling

• Entry point: handle_pte_fault (mm/memory.c)
• Identify which VMA faulting address falls in
• Identify if VMA has registered a fault handler
• Default fault handlers
  – do_anonymous_page: no page and no file
  – do_linear_fault: vm_ops registered?
  – do_swap_page: page backed by swap
  – do_nonlinear_fault: page backed by file
  – do_wp_page: write protected page (CoW)
The Page Fault Handler

Complex logic: easier to read code than read a book!
Copy on Write

- PTE entry is marked as un-writeable
- But VMA is marked as writeable
- Page fault handler notices difference
  - Must mean CoW
  - Make a duplicate of physical page
  - Update PTEs, flush TLB entry
  - do_wp_page
Which page to map when no PTE?

• If PTE doesn’t exist for an anonymous mapping, it’s easy
  – Map standard zero page
  – Allocate new page (depending on read/write)

• What if mapping is a memory map? Or shared memory?
  – Need some additional data structures to map logical object to set of pages
  – Independent of memory map of individual task

• The address_space structure
  – One per file, device, shared memory segment, etc.
  – Mapping between logical offset in object to page in memory
  – Pages in memory are called “page cache”
  – Files can be large: need efficient data structure

• You don’t have to use address_space for hw4. Use a simple array to maintain your offset->page mapping.
The Page Cache Radix Tree

(a) radix tree of height 1

(b) radix tree of height 2

radix_tree_root

height = 1

radix_tree_node

count = 2

slots[0] slots[4]

page page

index = 0 index = 4

radix_tree_root

height = 2

radix_tree_node

count = 2

slots[0] slots[2]

page

index = 0

radix_tree_node

count = 1

slots[3]

page

index = 131
Physical pages: struct page

- Each physical page has a page descriptor associated with it
- Contains reference count for the page
- Contains a pointer to the reverse map (struct address space or struct anon_vma)
- Contains pointers to lru lists (to evict the page)
- Descriptor to address: void * page_address(struct page *page)

```c
struct page {
    unsigned long flags;
    atomic_t _count;
    atomic_t _mapcount;
    struct address_space *mapping;
    pgoff_t index;
    struct list_head lru;
};
```
Allocating a Physical Page

• Physical memory is divided into “zones”
  – ZONE_DMA: low order memory (<16MB) certain older devices can only access so much
  – ZONE_NORMAL: normal kernel memory mapping into the kernel’s address space
  – ZONE_HIGHMEM: high memory not mapped by kernel. Identified through (struct page *). Must create temporary mapping to access

• To allocate, use kmalloc or related set of functions. Specify zone and options in mask
  – kmalloc, __get_free_pages, __get_free_page, get_zeroed_page: return virtual address (must be mapped)
  – alloc_pages, alloc_page: return struct page *
Page Table Structure
Working with Page Tables

- Access page table through mm_struct->pg_d
- Must to a recursive walk, pgd, pud, pmd, pte
  - Kernel includes code to assist walking
  - mm/pagewalk.c: walk_page_range
  - Can specific your own function to execute for each entry

- Working with PTE entries
  - Lots of macros provided (asm/pgtable.h, page.h)
  - Set/get entries, set/get various bits
  - E.g., pte_mkyoung(pte_t): clear accessed bit,
    pte_wrprotect(pte_t): clear write bit
  - Must also flush TLB whenever entries are changed
    - include/asm-generic/tkb.h: tlb_remove_tlb_entry(tlb)
Reverse Mappings

• Problem: how to swap out a shared mapping?
  – Many PTEs may point to it
  – But, we know only identity of physical page
    • Could maintain reverse PTE
    • i.e., for every page, list of PTEs that point to it
    • Could get large. Very inefficient.

• Solution: reverse maps
  – Anonymous reverse maps: anon_vma
  – Idea: maintain one reverse mapping per vma (logical object) rather than one reverse mapping per page
  – Based on observation most pages in VMA or other logical object (e.g., file) have the same set of mappers
  – rmap contains VMAs that may map a page
  – Kernel needs to search for actual PTE at runtime
Anonymous rmaps: anon_vma
anon_vma in Action

Reference: Virtual Memory II: the return of objrmap. http://lwn.net/Articles/75198/
anon_vma in Action

Reference: Virtual Memory II: the return of objrmap. http://lwn.net/Articles/75198/
Reverse Mapping for Memory Maps

• Problem: anon_vma is good for limited sharing
  – Memory maps can be shared by large numbers of processes
  – E.g., libc shared by everyone
  – I.e., need to do linear search for every eviction
  – Also, different processes may map different ranges of a memory map into their address space

• Need efficient data structure
  – Basic operation: given an offset in an object (such as a file), or a range of offsets, return vmas that map that range
  – Enter priority search trees
  – Allows efficient interval queries

• Note: you don’t need this for hw4. Use anon_vma
i_mmap Priority Tree

Part of struct address_space in fs.h
Page Frame Reclaiming (Swapping)

- Generic subsystem for memory and files (vmscan.c)
  - Handles anonymous pages (swapping)
  - Memory mapped files (synchronizing)
- Handles anonymous/file pages differently
  - Unreclaimable: pages locked in memory (PG_locked)
  - Swappable: anonymous user mode pages
  - Syncable: memory mapped pages, synchronize with original file they were loaded from
  - Discardable: unused pages in memory caches, non-dirty pages in page cache
- PFRA Design
  - Identify pages to evict using simplified LRU
  - Unmap all mappers of shared using reverse map (try_to_unmap function)
When is PFRA Invoked?

- Invoked on three different occasions:
  - Kernel detects low on memory condition
    - E.g., during alloc_pages
  - Periodic reclaiming
    - kernel thread kswapd
  - Hibernation reclaiming
    - for suspend-to-disk
Page Frame Reclaiming Algorithm

LOW ON MEMORY RECLAIMING
- Low memory on buffer allocation
  - __getblk()
  - alloc_page_buffers()
- free_more_memory()
  - try_to_free_pages()
  - out_of_memory()

LOW ON MEMORY RECLAIMING
- Low memory on page allocation
  - __alloc_pages()

HIbernATION RECLAIMING
- Suspend to disk (hibernation)
  - pm_suspend_disk()

PERIODIC RECLAIMING
- kswapd
  - kswapd()
  - cache_reap()
  - slab_destroy()

- balance_pgdats
  - shrink_slab()
  - shrink_caches()
  - shrink_zone()
  - shrink_cache()
  - shrink_list()
  - refill_inactive_zone()
  - page_referenced()
  - pageout()
The Swap Area Descriptor
The Swap Cache

- **Goal:** prevent race conditions due to concurrent page-in and page-out
- **Solution:** page-in and page-out serialized through a single entity: swap cache
- Page to be swapped out simply moved to cache
- Process must check if swap cache has a page when it wants to swap in
  - If the page is there in the cache already: **minor page fault**
  - If page requires disk activity: **major page fault**
The Swap Cache
Page Allocation

• Buddy Allocator
• SLAB allocator: data structure specific
• SLOB: simple list of blocks
• SLUB: efficient SLAB

We’ll see next