W4118 Operating Systems

Instructor: Junfeng Yang
Outline

- x86 segmentation and paging hardware
- Linux address space translation
- Copy-on-write
- Linux page replacement algorithm
- Linux dynamic memory allocation
x86 segmentation and paging

- Using Pentium as example
- CPU generates virtual address (seg, offset)
  - Given to segmentation unit
    - Which produces linear addresses
  - Linear address given to paging unit
    - Which generates physical address in main memory
    - Paging units form equivalent of MMU
x86 segmentation hardware

![Diagram of x86 segmentation hardware](image)
Specifying segment selector

- virtual address: segment selector + offset

- Segment selector stored in segment registers (16-bit)
  - cs: code segment selector
  - ss: stack segment selector
  - ds: data segment selector
  - es, fs, gs

- Segment register can be implicitly or explicitly specified
  - Implicit by type of memory reference
    - jmp $8049780 // implicitly use cs
    - mov $8049780, %eax // implicitly use ds
  - Through special registers (cs, ss, es, ds, fs, gs on x86)
    - mov %ss:$8049780, %eax // explicitly use ss
x86 paging hardware
Outline

- x86 segmentation and paging hardware
- Linux address space translation
- Copy-on-write
- Linux page replacement algorithm
- Linux dynamic memory allocation
Linux address translation

- Linux uses paging to translate virtual addresses to physical addresses
- Linux does not use segmentation

Advantages
- More portable since some RISC architectures don’t support segmentation
- Hierarchical paging is flexible enough
Linux segmentation

- Since x86 segmentation hardware cannot be disabled, Linux just uses NULL mappings.

- Linux defines four segments:
  - Set segment base to 0x00000000, limit to 0xffffffff,
    - segment offset == linear addresses
  - User code (segment selector: __USER_CS)
  - User data (segment selector: __USER_DS)
  - Kernel code (segment selector: __KERNEL_CS)
  - Kernel data (segment selector: __KERNEL_DATA)

- arch/i386/kernel/head.S
Segment protection

- **Current Privilege level (CPL)** specifies privileged mode or user mode
  - Stored in current code segment descriptor
  - User code segment: CPL = 3
  - Kernel code segment: CPL = 0

- **Descriptor Privilege Level (DPL)** specifies protection
  - Only accessible if CPL <= DPL

- **Switch between user mode and kernel mode (e.g. system call and return)**
  - Hardware load the corresponding segment selector (__USER_CS or __KERNEL_CS) into register cs
Paging

- Linux uses up to 4-level hierarchical paging
- A linear address is split into five parts, to seamlessly handle a range of different addressing modes
  - Page Global Dir
  - Page Upper Dir
  - Page Middle Dir
  - Page Table
  - Page Offset

- Example: 32-bit address space, 4KB page without physical address extension (hardware mechanism to extend address range of physical memory)
  - Page Global dir: 10 bits
  - Page Upper dir and Page Middle dir are not used
  - Page Table: 10 bits
  - Page Offset: 12 bits
## Paging in 64 bit Linux

<table>
<thead>
<tr>
<th>Platform</th>
<th>Page Size</th>
<th>Address Bits Used</th>
<th>Paging Levels</th>
<th>Address Splitting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha</td>
<td>8 KB</td>
<td>43</td>
<td>3</td>
<td>10+10+10+13</td>
</tr>
<tr>
<td>IA64</td>
<td>4 KB</td>
<td>39</td>
<td>3</td>
<td>9+9+9+12</td>
</tr>
<tr>
<td>PPC64</td>
<td>4 KB</td>
<td>41</td>
<td>3</td>
<td>10+10+9+12</td>
</tr>
<tr>
<td>sh64</td>
<td>4 KB</td>
<td>41</td>
<td>3</td>
<td>10+10+9+12</td>
</tr>
<tr>
<td>X86_64</td>
<td>4 KB</td>
<td>48</td>
<td>4</td>
<td>9+9+9+9+12</td>
</tr>
</tbody>
</table>
Page table operations

- Linux provides data structures and operations to create, delete, read and write page directories
  - `include/asm-i386/pgtable.h`
  - `arch/i386/mm/hugetlbpage.c`

- Naming convention
  - `pgd`: Page Global Directory
  - `pmd`: Page Middle Directory
  - `pud`: Page Upper Directory
  - `pte`: Page Table Entry
  - Example: `mk_pte(p, prot)`
TLB operations

- x86 uses hardware TLB
  - OS does not manage TLB

- Only operation: flush TLB entries
  - include/asm-i386/tlbflush.h
  - movl %0 cr3: flush all TLB entries
  - invlpg addr: flush a single TLB entry
    - More efficient than flushing all TLB entries
Outline

- x86 segmentation and paging hardware
- Linux address space translation
- Copy-on-write
- Linux page replacement algorithm
- Linux dynamic memory allocation
A cool trick: copy-on-write

- In `fork()`, parent and child often share significant amount of memory
  - Expensive to copy all pages

- COW Idea: exploit VA to PA indirection
  - Instead of copying all pages, share them
  - If either process writes to shared pages, only then is the page copied
  - How to detect page write?
    - Mark pages as read-only in both parent and child address space
    - On write, page fault occurs
Share pages

- `copy_process()` in `kernel/fork.c`
- `copy_mm()`
- `dup_mmap()` // copy page tables
- `copy_page_range()` in `mm/memory.c`
- `copy_pud_range()`
- `copy_pmd_range()`
- `copy_pte_range()`
- `copy_one_pte()` // mark readonly
Copy page on page fault

- `set_intr_gate(14, &page_fault)` in `arch/i386/kernel/traps.c`
- `ENTRY(page_fault)` calls `do_page_fault` in `arch/i386/kernel/entry.s`

- `do_page_fault` in `arch/i386/mm/fault.c`
- `cr2` stores faulting virtual address
- `handle_mm_fault` in `mm/memory.c`
- `handle_pste_fault` in `mm/memory.c`
- `if(write_access)`
- `do_wp_page()`
Outline

- x86 segmentation and paging hardware
- Linux address space translation
- Copy-on-write
- Linux page replacement algorithm
- Linux dynamic memory allocation
Linux page replacement algorithm

- Two lists in *struct zone*
  - `active_list`: hot pages
  - `inactive_list`: cold pages

- Two bits in *struct page*
  - `PG_active`: is page on active list?
  - `PG_referenced`: has page been referenced recently?

- Approximate LRU algorithm
  - Replace a page in inactive list
  - Move from active to inactive under memory pressure
  - Need two accesses to go from inactive to active
Functions for page replacement

- lru_cache_add*: add to inactive or active list
- mark_page_accessed(): called twice to move a page from inactive to active
- page_referenced(): test if a page is referenced
- refill_inactive_zone(): move pages from active to inactive
How to swap out page

- free_more_memory() in fs/buffer.c called
- try_to_free_pages in mm/vmscan.c
- shrink_caches
- shrink_zone
- refill_inactive_zone
- shrink_cache
- shrink_list
- if(PageDirty(page))
- pageout()
How to load page

- **On page fault**, `cr2` stores faulting virtual address
- `handle_mm_fault()` in `mm/memory.c`
- `handle_pte_fault()` if(!pte_present(entry))
- `do_no_page()` // anonymous page
- `do_file_page()` // file mapped page
- `do_swap_page()` // swapped out page
Outline

- x86 segmentation and paging hardware
- Linux address space translation
- Copy-on-write
- Linux page replacement algorithm
- Linux dynamic memory allocation
Dynamic memory allocation

- How to allocate pages?
  - Data structures for page allocation
  - Buddy algorithm for page allocation

- How to allocate objects?
  - Slab allocation
Page descriptor

- Keep track of the status of each physical page
  - `struct page`, `include/linux/mm.h`

- All stored in `mem_map` array

- Simple mapping between a page and its descriptor
  - Nth page’s descriptor is `mem_map[N]`
  - `virt_to_page`
  - `page_to_pfn`
Memory zone

- Keep track of pages in different zones
  - `struct zone`, include/linux/mmzone.h
  - ZONE_DMA: <16MB
  - ZONE_NORMAL: 16MB-896MB
  - ZONE_HIGHMEM: >896MB
Linux page allocator

- Linux use a **buddy allocator** for page allocation
  - Fast, simple allocation for blocks that are $2^n$ bytes [Knuth 1968]

- Idea: a free list for each size of block users want to allocate

- `__page_alloc()` in `mm/page_alloc.c`
Linux buddy allocator implementation

- **Data structure**
  - 11 free lists of blocks of pages of size $2^0, 2^1, \ldots, 2^{10}$

- **Allocation restrictions**: $2^n$ pages, $0 \leq n \leq 10$

- **Allocation of $2^n$ pages**:
  - Search free lists ($n, n+1, n+2, \ldots$) for appropriate size
    - Recursively divide larger blocks until reach block of correct size
    - Insert "buddy" blocks into free lists

- **Free**
  - Recursively coalesce block with buddy if buddy free
Pros and cons of buddy allocator

- **Advantages**
  - Fast and simple compared to general dynamic memory allocation
  - Avoid external fragmentation by keeping free physical pages contiguous

- **Disadvantages**
  - Internal fragmentation
    - Allocation of block of k pages when k != 2^n
Slab allocator

- For objects smaller than a page
- Implemented on top of page allocator
- Memory managed by slab allocator is called **cache**

- Two types of slab allocator
  - **Fixed-size slab allocator**: cache contains objects of same size
    - for frequently allocated objects
  - **General-purpose slab allocator**: caches contain objects of size $2^n$
    - for less frequently allocated objects
    - For allocation of object with size $k$, round to nearest $2^n$

- `_kmem_cache_create()` and `_kmalloc()` in `mm/slab.c`
Pros and cons of slab allocator

- **Advantages**
  - Reduce internal fragmentation: many objects in one page
  - Fast

- **Disadvantages**
  - Memory overhead for bookkeeping
  - Internal fragmentation for general-purpose slab allocator