Contents

- Memory Management
- Kernel Memory Allocation
- Page Allocation
- Page Cache
Virtual Memory Management

- Separation of user virtual memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Using a disk as an extension of RAM so that the effective size of usable memory grows correspondingly.
MMU (Memory Management Unit)

- Translates the virtual address into the physical RAM address
- Maintains the system's page table
- Normally 4KB for each page in 32bit architecture
Physical Memory Management

- **Node (struct pglist_data)**
  - For each memory bank
  - Different access cost depending on its distance from CPU
  - 1 node for UMA

- **Zone (struct zone_struct)**
  - ZONE_DMA: lower physical memory ranges for ISA DMA
  - ZONE_NORMAL: directly mapped by kernel
  - ZONE_HIGH: not directly mapped by kernel

- **Page (struct page)**
  - Fixed-size chunks (page frames)
Zone (32bit)
Demand Paging

- Demand fetching
  - Page is only fetched from swap space when hardware raise a page fault exception, which then the OS traps and allocates a page
  - A number of pages after the faulting page is prefetched
COW (Copy-On-Write)

- In Linux, fork() is implemented through the use of copy-on-write
- Copy-on-write (COW)
  - Technique to delay copying of the data
  - Sharing process address space until task writes a page
  - Parent and child share read-only
  - Occurred page_fault, kernel copy pages for new task
- Reduce copy overhead when exec() is called immediately after fork()
  - Never need to be copied
COW (Copy-On-Write)
kmalloc()

- A simple interface for byte-sized allocations
- Allocates physically contiguous memory
- Although only mandatory for hardware devices it’s faster than vmalloc
**vmalloc()**

- Allocate more than 4MB
- Allocate logically contiguous memory, but physically not
- Used when modules are dynamically inserted into kernel
- Large overhead
  - Change kernel page table
Page Allocator

- Linux use both of the allocator
- Slab Allocator
  - To enhance memory allocation of small
  - Frequently-used data structures (< sizeof page)
- Buddy Allocator
  - Manages physical memory in pages (8KB)
Slab Allocator

- It reduces internal fragmentation, inside pages
- It works upon Buddy algorithm
- How it works:
  - It creates a cache for each frequently used kernel data structures
  - Cache is a collection of slabs
  - Slab size maybe one or two pages
  - Slab contains a group of objects of similar type
Slab Allocator

- **SLAB**
  - A set of one or more contiguous pages of memory handled by the slab allocator for an individual cache

- **SLOB**
  - Designed for small systems with limited amounts of memory
  - Embedded Linux systems

- **SLUB**
  - Currently the default slab allocator in the Linux kernel
  - Implemented to solve some drawbacks of the SLAB design
Slab Allocator

- Cache: stores a different type of object
- Slabs: composed of one or more physically contiguous pages
- Object: data structures being cached
Allocate Memory from fixed-size segment consisting of physically contiguous pages

Free page frames are grouped into lists of blocks containing \(2^n\) contiguous page frames.

To avoid external fragmentation without paging
Buddy Allocator

A diagram shows a structure named `struct free_area` with fields ordered from 0 to `MAX_ORDER-1`. Each field represents the number of free page blocks of a certain size, starting from 2^0 up to 2^4. The diagram illustrates how these sizes are organized in memory, with larger sizes grouped together.
Memory Allocator

Some Kernel Code

kmalloc allocator
Uses a set of anonymous SLAB caches

SLAB Allocator
Allows to create caches, each cache storing objects of the same size. Size can be lower or greater than a page size.

vmalloc Allocator
Non-physically contiguous memory

Page Allocator
Allows to allocate contiguous areas of physical pages (4k, 8k, 16k, 32k, etc)
CPU Cache
Page Cache

- Page cache also called the disk cache, file cache
- Basic thing is similar to CPU cache
- Goal is to minimize disk I/O
  - Disk access is slower than memory access
Page Cache

- When kernel reads data, checks if the data is in page cache
  - **Cache hit**
    - The data is read off the page cache
  - **Cache Miss**
    - The data is read off the disk and is populated in page cache
Write cache

- When kernel writes data, there are three different strategies
  - **No-write**
    - Writes invalidate the page cache and go directly to disk
  - **Write-through**
    - Write data to page cache and disk
  - **Write-back**
    - If pages are modified, set Dirty bit
    - Dirty pages are added to dirty list
    - Periodically dirty pages are written back to disk
### Target Page Selection

- If low memory, situation arises then kernel calls reclaiming algorithm

<table>
<thead>
<tr>
<th>Type of pages</th>
<th>Description</th>
<th>Reclaim action</th>
</tr>
</thead>
</table>
| **Unreclaimable** | Free pages (included in buddy system lists)  
Reserved pages (with PG_reserved flag set)  
Pages dynamically allocated by the kernel  
Pages in the Kernel Mode stacks of the processes  
Temporarily locked pages (with PG_locked flag set)  
Memory locked pages (in memory regions with VM_LOCKED flag set) | (No reclaiming allowed or needed) |
| **Swappable** | Anonymous pages in User Mode address spaces  
Mapped pages of *tmpfs* filesystem (e.g., pages of IPC shared memory) | Save the page contents in a swap area |
| **Syncable** | Mapped pages in User Mode address spaces  
Pages included in the page cache and containing data of disk files  
Block device buffer pages  
Pages of some disk caches (e.g., the inode cache) | Synchronize the page with its image on disk, if necessary |
| **Discardable** | Unused pages included in memory caches (e.g., slab allocator caches)  
Unused pages of the dentry cache | Nothing to be done |
PFRA (Page Frame Reclaiming Algorithm)

- Free the “harmless” pages first
  - Pages included in disk and memory caches not referenced by any process. (without modifying PTE)
- Make all pages of a User Mode process reclaimable.
  - With the exception of locked pages, the PFRA must be able to steal any page of a User Mode process
- Reclaim a shared page frame by unmapping at once all page table entries that reference it
  - Reverse mapping (anon_vma, priority search tree)
- Reclaim “unused” pages only
  - LRU replacement algorithm
PFRA (Page Frame Reclaiming Algorithm)

- The kernel stores two doubly-linked lists of frames:
  - active_list - Recently accessed frames
  - inactive_list - Frames that wasn’t accessed recently
- Those lists contains all the frames that can be evicted
- Non-overlapping lists
LRU (Least Recently Used)

- Replaces the page that has not been referenced for the longest time
- By the principle of locality, this should be the page least likely to be referenced in the near future
Two-list strategy

- Modified version of LRU
- Active list (hot) and Inactive list (reclaim candidate)
- Pages when first allocated are placed on inactive list
- If referenced while on that list, it will be placed on active list
Radix Tree

- Each address space has a unique radix tree
- Enable quick searching for the desired page, given only the file offset
  - Index is the offset inside the corresponding file
Buffer Cache

- Buffer cache is part of the page cache
  - Buffer is the in-memory representation of a single disk block
- To Reduce the frequency of disk access
- Buffer header contains the metadata information
  - Device number and the block number range for which this buffer holds the data
Buffer Cache
Q&A
Reference

- Understanding the Linux Kernel, 3rd-ed.
- Professional Linux Kernel Architecture
- Understanding the Linux Virtual Memory Manager