We adapt. You succeed.

Tour de Linux memory management

Michal Hocko
Suse Labs
mhocko@suse.com

Documentation & Upstream development

- **`Documentation/vm`**
  - Quite ad-hoc – systematic design documentation is missing
- **lwn.net**
  - Many very good articles
- **Understanding The Linux Virtual Manager – by Mel Gorman**
  - Very good and systematic coverage but too old – from 2.4 era (with What’s new in 2.6 sections)
  - Still very useful to understand core design principals
  - https://www.kernel.org/doc/gorman/
- **Upstream development**
  - Mailing list linux-mm@kvack.org
  - Patches routed usually via Andrew Morton <akpm@linux-foundation.org> and hist mm tree
  - Code lives mostly in `mm/` and many include files

Purpose and the scope of MM

- **Manage system RAM**
  - Architecture independent view on the memory
  - Support for UMA/NUMA architectures
  - Memory hotplug support – used by NVDIMMs
- **Support for memory over-commit**
  - Virtual memory
  - On demand memory reclaim
  - CopyOnWrite
- **Support also for MMUless architectures**

Purpose and the scope of MM APIs for kernel

- **Bootmem/memblock allocator – early initialization**
- **Page allocator – page order (2^N physically contiguous pages)**
- **SLAB allocator – sub page granularity, internal fragmentation management**
  - SLOB – very rudimentary for small devices
  - SLAB – based on Solaris design – optimized for CPU cache efficiency, NUMA aware
  - SLUB – new generation design – aimed for better scalability
- **Vmalloc – virtually contiguous memory allocator – via page tables**
- **Mempool allocator**
  - Guarantee for a forward progress – mostly for IO paths
- **Page cache management for filesystems**
- **Memory tracking for userspace – process management**
- **Page table management**
  - get_user_pages – virtual → struct page translation
- **On-demand memory paging**

Purpose and the scope of MM APIs for userspace

- **Syscalls to manage memory**
  - mmap, munmap, mprotect, brk, mlock – POSIX
  - madvise – hints from userspace e.g. MADV_DONTNEED, MADV_FREE etc...
  - userfaultfd – page fault handling from userspace
  - SystemV shared memory – IPC, shmget, shmat, shmdt
  - memfd_create – anonymous memory referenced by a file descriptor – for IPC
- **Memory backed filesystems**
  - Ramdisk – fixed sized memory backed block device
  - Ramfs – simple memory backed filesystem
  - Tmpfs – more advanced memory backed filesystem, support for swapout, ACL, extended attributes
- **Memory cgroups controller – more fine grained partitioning of the system memory**
  - Mostly for user space consumption limiting, kernel allocations are opt-in
  - Support for hard limit, soft/low limit, swap configuration, userspace OOM killer
- **Access to huge pages (2MB, 1GB)**
  - Hugetlbfs – filesystem backed by preallocated huge pages
  - THP – transparent huge pages
- **NUMA allocation policies**
  - Mbind, set_mempolicy, get_mempolicy

Physical Memory representation

- **Managed in page size granularity – arch specific, mostly 4kB**
- **Each page is represented by `struct page`**
- **Heavily packed – 64B on 64b systems (~1.5% with 4kB pages)**
  - Lots of unions to distinguish different usage
  - Special tricks to save space – set bottom bits in addresses etc...
- **Statically allocated during boot/memory hotplug - `memmap`**
- **Reference counted**
  - `get_page(), put_page(), get_page_unless_zero(), put_page_test_zero()`
  - memory is returned to the page allocator when 0
- **`pfn_valid(), pfn_to_page(), page_to_pfn()` – physical page frame number to struct page translation**
- **page_owner – tracks stack of the allocation request – very useful for debugging**

Physical Memory representation

- **Memory ranges exported by BIOS/EFI firmware**
  - E820 for x86 systems

```
[    0.000000] e820: BIOS-provided physical RAM map:
[    0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009dbff] usable
[    0.000000] BIOS-e820: [mem 0x000000000009dc00-0x000000000009ffff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000000f0000-0x00000000000fffff] reserved
[    0.000000] BIOS-e820: [mem 0x0000000000100000-0x00000000bf61ffff] usable
[    0.000000] BIOS-e820: [mem 0x00000000bf620000-0x00000000bf63bfff] ACPI data
[    0.000000] BIOS-e820: [mem 0x00000000bf63c000-0x00000000bf63cfff] usable
[    0.000000] BIOS-e820: [mem 0x00000000bf63d000-0x00000000cfffffff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000fec00000-0x00000000fee0ffff] reserved
[    0.000000] BIOS-e820: [mem 0x00000000ff800000-0x00000000ffffffff] reserved
[    0.000000] BIOS-e820: [mem 0x0000000100000000-0x000000403fffefff] usable
```

- **Memory model defines how we represent physical memory ranges**
  - Flatmem – the simplest one, single range of physical memory, doesn’t support NUMA
  - Discontigmem – more advanced, supports holes, NUMA, doesn’t support memory hotplug
  - Sparsemem – the most widely used, supports NUMA, memory hotplug, keeps track of memory range in memory sections
    - Vmemmap sparsemem – virtually contiguous memory map via page tables, very efficient pfn_to_page (simple pointer arithmetic)

Page flags

- **`enum pageflags` – describes the state of the page**
- **PG_$NAME are accessed via Page$NAME(), SetPage$NAME(), TestSetPage$NAME(), ClearPage$NAME(), TestClearPage$NAME()**
  - Defined by macros

```
PAGEFLAG(Referenced, referenced, PF_HEAD)
        TESTCLEARFLAG(Referenced, referenced, PF_HEAD)
        __SETPAGEFLAG(Referenced, referenced, PF_HEAD)
```

  - Atomic updates
  - Non atomic variants `__SetPage$NAME, __ClearPage$NAME`
- **Page lock is implemented as bit lock**
- **Upper part of flags is used to encode numa node/section_nr, zone id**

Physical Memory representation

- NUMA node represented by `struct pglist_data`
- UMA machines have one static numa node, NUMA has an array of nodes
- SRAT tables on x86 systems – describe nodes, distances
- Kswapd kernel thread for the background memory reclaim
- LRU lists for the memory reclaim
- Free pages are maintained on the per-zone bases
- Counters - `/proc/zone_info`

```
[    0.000000] ACPI: SRAT: Node 0 PXM 0 [mem 0x00000000-0xbfffffff]
[    0.000000] ACPI: SRAT: Node 0 PXM 0 [mem 0x100000000-0x103fffffff]
[    0.000000] ACPI: SRAT: Node 1 PXM 1 [mem 0x1040000000-0x203fffffff]
[    0.000000] ACPI: SRAT: Node 2 PXM 2 [mem 0x2040000000-0x303fffffff]
[    0.000000] ACPI: SRAT: Node 3 PXM 3 [mem 0x3040000000-0x403fffffff]
[    0.000000] NUMA: Node 0 [mem 0x00000000-0xbfffffff] + [mem 0x100000000-0x103fffffff] -> [mem 0x00000000-0x103fffffff]
[    0.000000] NODE_DATA(0) allocated [mem 0x103ffde000-0x103fffffff]
[    0.000000] NODE_DATA(1) allocated [mem 0x203ffde000-0x203fffffff]
[    0.000000] NODE_DATA(2) allocated [mem 0x303ffde000-0x303fffffff]
[    0.000000] NODE_DATA(3) allocated [mem 0x403ffdd000-0x403fffefff]

$ numactl -H
available: 4 nodes (0-3)
node 0 cpus: 0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60
node 0 size: 64295 MB
node 0 free: 53958 MB
node 1 cpus: 1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61
node 1 size: 64509 MB
node 1 free: 48875 MB
node 2 cpus: 2 6 10 14 18 22 26 30 34 38 42 46 50 54 58 62
node 2 size: 64509 MB
node 2 free: 50959 MB
node 3 cpus: 3 7 11 15 19 23 27 31 35 39 43 47 51 55 59 63
node 3 size: 64507 MB
node 3 free: 33646 MB
node distances:
node   0   1   2   3
 0:  10  20  20  20
 1:  20  10  20  20
 2:  20  20  10  20
 3:  20  20  20  10
```

Physical Memory representation

- **Memory zones for the page allocator – `struct zone`**
  - Defines a class of memory
    - `ZONE_DMA` – low 16MB for legacy HW (ISA buses)
    - `ZONE_DMA32` – low 4GB for 32b restricted devices
    - `ZONE_NORMAL` – memory usable by the kernel directly
    - `ZONE_HIGHMEM` – memory for userspace on 32b systems – has to be mapped to be used from the kernel
    - `ZONE_MOVABLE` – allocations which can be migrated – mostly user memory, page cache
    - `ZONE_DEVICE` – special zone to describe device memory – non-volatile memory DAX, non-coherent device memory HMM
  - Free pages maintained in `zone::free_area`
  - Watermarks to limit access to free pages `zone::watermark[]`

Virtual Memory representation

- **48b (128TB) view of contiguous memory which is translated to the physical memory by page tables**
- **Support for future 52b (4PB) physical address space in 5-level pte (57b of virtual)**
  - Explicit opt in to use in userspace by addr hint to mmap
- **Kernel vs. User space view**
  - Virtual address space is split to kernel and userspace
    - Kernel part is static and doesn’t change with context switches
    - 32b - Lowmem (1GB for direct usage) vs. Highmem (3GB)
      - Only low mem can be accessed directly, highmem has to be mapped temporarily
      - Only 896MB usable – 128MB reserved for vmalloc and kmap

        ```
        0000000 – BFFFFFFF user space
        C000000 – F7xxxxxx kernel (direct mapping)
        F7xxxxx – FF7FE0000 vmalloc
        FF80000 – FFC000000 kmap
        ```
    - 64b – negative address space kernel, positive userspace

      ```
      0000000000000000 – 00007FFFFFFFFFFF – user space
      FFFF880000000000 – FFFFC7FFFFFFFFFF – direct mapping
      FFFFC90000000000 – FFFFE8FFFFFFFFFF – vmalloc
      ```
  - Kernel space is configured to use direct 1:1 mapping
    - Translation is a simple arithmetic operation (`__va(), __pa()`)

Virtual Memory representation

- **Page table walkers use unified 5 page table levels**
  - `pgd_t, p4d, pud_t, pmd_t` and `pte_t`
  - `pgd_alloc, pgd_none, pgd_index, pgd_offset etc...`
  - Architectures with a different pte topology emulate 5 levels (e.g. `include/asm-generic/5level-fixup.h`)
- **Simple page table walk**

```
pgd = pgd_offset(mm, addr) /* mm of the process or init_mm */
P4d = p4d_offset(pgd, addr)
pud = pud_offset(p4d, addr)
pmd = pmd_offset(pud, addr)
pte = pte_offset_map_lock(mm, pmd, addr, &ptl)
```

- **Once we have pte – `vm_normal_page()`**
  - `pte_pfn()+ pfn_to_page` with some special casing for special mappings

Address space descriptor

- **Each process has its address space descriptor `struct mm_struct`**
- **Keeps track of all the maped memory**
  - `mm_struct::mmap` – linked list of all mapped areas (VMA)
  - `mm_struct::mm_rb` – RedBlack tree for quick VMA lookups - `find_vma`
- **Reference counted**
  - `mm_count – mmgrab(), mmdrop()`
    - Number of mm_struct users – last reference will free the data structure
  - `mm_users – mmget(), mmget_not_zero(), mmput()`
    - Number of users of the address space – last user will unmap the whole address space
- **Links to the top page table entry – mm_struct::pgd**
- **Counters – number of page table entries, locked memory, high_rss etc…**
- **mmap_sem – RW lock to serialize address space operations**
  - And more abusers unfortunately

Address space descriptor

- **Mapped memory range `struct vm_area_struct`**
- **Created for mmap, brk, special mappings (VDSO)**
- **vm_flags**
  - Access protection – `VM_READ, VM_WRITE, VM_EXEC`
  - Mlock status – `VM_LOCKED`
  - Special mappnig – `VM_IO, VM_PFNMAP, VM_MIXEDMAP`
- **Link to the mapped object – vm_file or anon_vma**
- **Memory policy for the area**
- **Set of “virtual functions” - vm_ops**
  - How to handle page fault – `fault()`
  - Notify the backing store that a read only page will become writable – `page_mkwrite()` – FS can refuse due to `ENOSPACE` and process will get `SIGBUS`
  - Other hooks for special device mappings

On demand paging

- **HW (onx86) will trigger #PF exception when the pte is not mapped or the current protection doesn't allow requested operation (e.g. Write on ReadOnly pte).**
- **`do_page_fault` – main entry – arch specific**
  - A lot of special casing – e.g. faults from kernel, fixups, errata workarounds etc
  - Take `mmap_sem` in read mode
  - `find_vma` – no VMA → `SEGV`
  - Expand stack VMAs – `VM_GROWS_{UP,DOWN}`
  - handle_mm_fault – arch independent page fault handling
    - Wrong access SEGV
  - `__handle_mm_fault` → pte_walk, handle large pages (PUD, PMD) or handle_pte_fault for base pages
    - `do_anonymous_page` – allocates a new page, setups page table, reverse mapping, adds page the LRU list
    - `do_fault` – relies on `vm_ops→fault()` - many filesystems rely on filemap_fault
    - `do_swap_page` – swapped out page – swap it in
    - `do_numa_page` – used by numa balancing
    - `do_wp_page` – break CoW page – allocate new anonymous page for private mappings
  - Parallel page faults are handled by rechecking pte against the saved one under the page table lock (`pte_same()`)

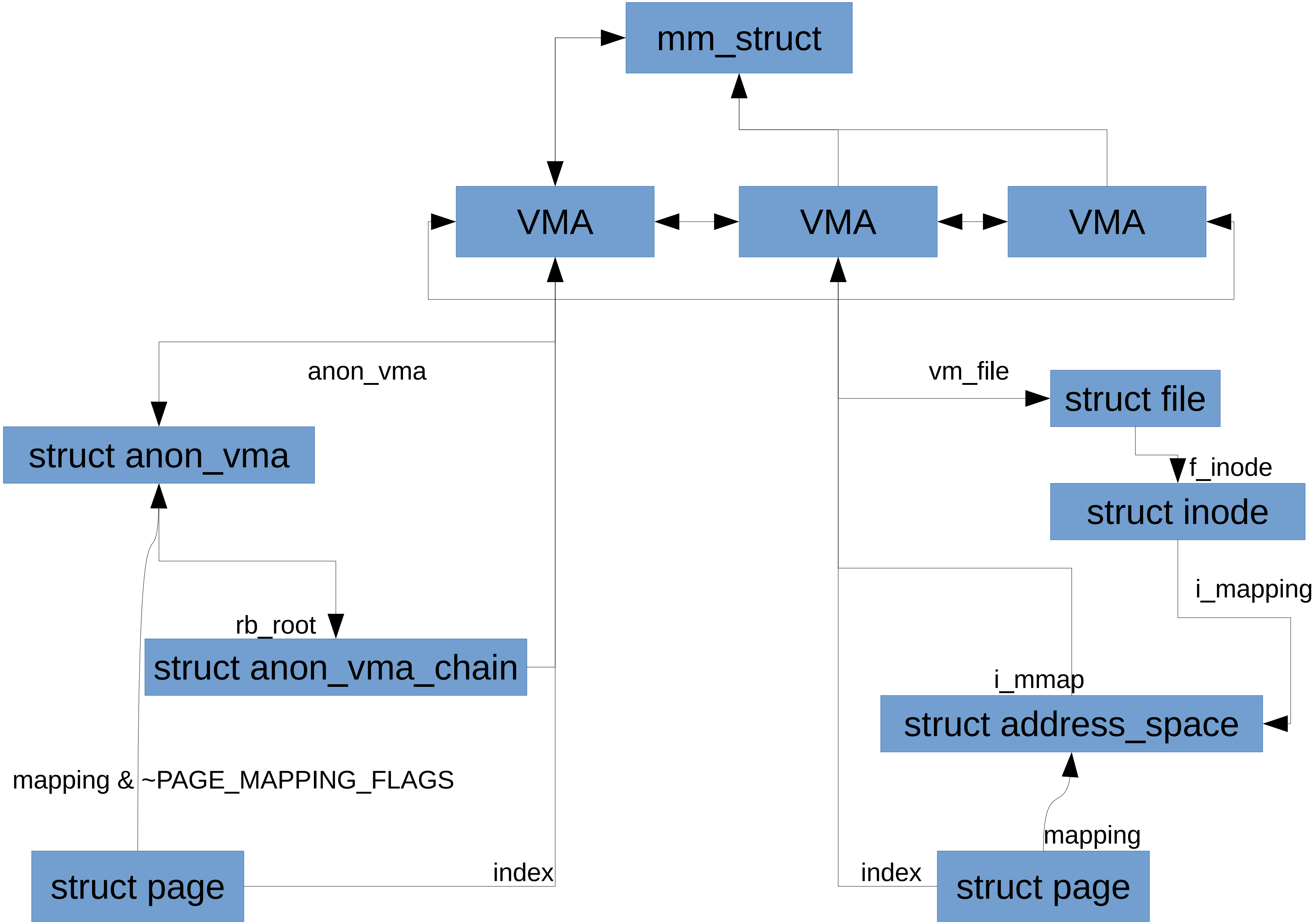
page → VMA mappings

- **How to get from struct page to all mappings? (`mm/rmap.c`)**
  - `rmap_walk` – `rmap_walk_control` defines callback to call for each mapping
- **page::mapping, page::index**
  - Anonymous pages – page::mapping has the lowest bit set
    - `anon_vma = page→mapping & ~PAGE_MAPPING_FLAGS`
      - Address space of all anonymous pages – hierarchical tree of interval trees

```
INTERVAL_TREE_DEFINE(struct anon_vma_chain, rb, unsigned long, rb_subtree_last,
                     avc_start_pgoff, avc_last_pgoff,
                     static inline, __anon_vma_interval_tree)
```

      - More on https://lwn.net/Articles/383162/
    - `pgoff = page→index`
    - `anon_vma_interval_tree_foreach` – iterates over all VMAs which contain pgoff
  - File backed pages
    - Mapping points to struct address_space – one per each inode/block device
    - `mapping→i_mmap` contains interval tree of all VMAs
    - `vma_interval_tree_foreach` iterates over all VMAs which contain pgoff

Address space – gluing it together



Page cache management

- **address_space::page_tree - radix_tree of pages belonging to the inode – move to xarray in the recent past**
- **filemap_fault**
  - find_get_page
    - Returns an existing page from the radix tree or allocates a new one `__page_cache_alloc()` and inserted to the radix tree `__add_to_page_cache_locked()` and LRU list
    - Page is locked and `!PageUptodate()` if newly allocated
  - `do_async_mmap_readahead()` – triggers asynchronous read from the backing storage (including readahead).
  - `do_sync_mmap_readahead()` – synchronous read
  - `read_pages` – `mapping→a_ops→readpages()` – to do the actual read from the (fs usually use `mpage_readpages()`)
  - Once we have the content – `SetPageUptodate() + PageUnlock()`

Page allocator

- **__alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order, struct zonelist *zonelist, nodemask_t *nodemask) to get a struct page**
- **__get_free_pages() to get a directly usable pointer – use with care!**
- **gfp_mask – bitmask for the allocation mode**
  - Request specific zones – `__GFP_DMA, __GFP_DMA32, __GFP_HIGHMEM, __GFP_MOVABLE`
  - `GFP_NOWAIT, GFP_ATOMIC` – non sleeping allocations, no direct reclaim
  - `GFP_KERNEL` – standard kernel allocations
  - `GFP_USER, GFP_USER_MOVABLE` – allocations for userspace
  - `GFP_NOFAIL` – non-failing allocations
  - `GFP_NOFS, GFP_NOIO` – do not recurse to fs perform any IO from the reclaim
- **Order – size of the allocation 2^{order} contiguous pages**
  - `PAGE_ALLOC_COSTLY_ORDER` (3) – small allocations are special – trigger OOM killer rather than fail
- **Zonelists – list of zones to allocate from**
  - Start with zones of a local or requested node - `node_zonelist()`
  - `build_zonelists()` - `numa_zonelist_order` kernel boot parameter – node order, zone order
- **Nodemask to filter only allowed nodes defined by memory policy**
  - Note that there is also cpuset API to overrule memory policies
  - Funny things will happen if those two disagree

Page allocator

- **Slow path quite complex**
  - Wake up kswapd/kcompactd
  - Triggers direct memory reclaim/compaction when needed
  - Triggers the OOM killer when no progress was made during the reclaim
- **Core of the page allocator – `get_page_from_freelist()`**
  - Checks watermarks to not allow memory depletion
  - Per-cpu allocation for order-0 – no locking, batch refill, freeing - `rmqueue_pcplist()`
  - `__rmqueue()` for other orders
- **Based on buddy allocator**
  - Physically contiguous pages are grouped in 2^N chunks
  - 2 2^{N-1} blocks are merged to 2^N when page is freed - `__free_one_page()`
  - A larger block is split up when appropriate is not available - `__rmqueue_smallest()` vs `__rmqueue_fallback()`

```
$ cat /proc/buddyinfo
Node 0, zone      DMA      1      0      1      0      1      1      1      0      1      1      3
Node 0, zone    DMA32      7      4      3      5      3      5      7      5      4      2    538
Node 0, zone   Normal    438    445   3397   1588    877    367    177     74     36      7    312
```

Memory reclaim

- **Background reclaim**
  - Kernel thread per NUMA node
  - Starts when free memory hit **low** watermark on all zones eligible for the allocation – `pgdat_balanced()`
  - Reclaims until **high** watermark is hit
  - The main logic implemented in `balance_pgdat()`
- **Direct reclaim**
  - All eligible zones hit the **min** watermark
  - Tries to free `SWAP_CLUSTER_MAX` pages
  - The main logic implemented in `try_to_free_pages()`
- **Node reclaim – former zone reclaim**
  - Enforce direct reclaim on the requested node first
  - Used to be enabled on NUMA machines with large numa distances in the past
  - Has to be enabled explicitly - `/proc/sys/vm/zone_reclaim_mode`
- **OOM killer**
  - Last despair attempt to free memory by killing the task with the largest memory consumption
  - `oom_reaper` – kernel thread to unmap memory of the oom victim
  - Very tricky to get right

Memory reclaim

- **Reclaimable pages are sitting on LRU lists – `struct lruvec`**

```
enum lru_list {
        LRU_INACTIVE_ANON = LRU_BASE,
        LRU_ACTIVE_ANON = LRU_BASE + LRU_ACTIVE,
        LRU_INACTIVE_FILE = LRU_BASE + LRU_FILE,
        LRU_ACTIVE_FILE = LRU_BASE + LRU_FILE + LRU_ACTIVE,
        LRU_UNEVICTABLE,
        NR_LRU_LISTS
};
```

- **Used to have LRU lists per zones, now we have one per node**
  - Actually per memory cgroup – more on that later
- **Pages are added to the list when allocated**
- **Anonymous pages start on the active list**
- **File pages start on the inactive list**
  - Pages freed recently are put to the active list - `workingset_refault()`
- **Promotion from inactive to active list based on pte references – `page_check_references()`**
  - Used once heuristic for file pages
  - Executable pages protection
- **Active list is shrunk when it grows too large – `inactive_list_is_low()`**

Memory reclaim

- **Each reclaim pass has a priority – starting from `DEF_PRIORITY` (12)**
  - Size of the window to scan LRU lists – `lruvec_lru_size() >> priority`
- **`get_scan_count()` - keeps balance between file and anonymous lru lists**
  - Highly biased to reclaim file pages
  - `/proc/sys/vm/swappiness`
  - Considers recently scanned and rotated pages for each LRU
- **`isolate_lru_pages()` - removes pages from the LRU list in a batch for further inspection**
  - Reduces the lock contention
  - Skip over ineligible pages – e.g. highmem pages for `GFP_KERNEL` request
- **`shrink_page_list()` - core of the reclaim**
  - Referenced pages are promoted to the active list
  - Anonymous pages are added to the swap cache and scheduled for swapout - `add_to_swap()`
  - Dirty file pages are written out – `pageout()` - only in kswapd context
  - Mapped pages are unmapped – `try_to_unmap_one()`
    - Anonymous ptes will point to swap entries, `MADV_FREE` pages are dropped
    - Dirty pte states is moved to struct page – `set_page_dirty()`
  - `__remove_mapping()`
    - Dirty pages are not removed – protection for races
    - Remove from the page cache (including swap cache) – records the eviction time for file cache `workingset_eviction()`

Memory reclaim

- **Many types of SLAB allocations are reclaimable**
  - Dentry, inode cache etc…
  - Register their shrinkers
    - Not restricted to slab objects only

```
struct shrinker {
        unsigned long (*count_objects)(struct shrinker *,
                                       struct shrink_control *sc);
        unsigned long (*scan_objects)(struct shrinker *,
                                      struct shrink_control *sc);
        int seeks;      /* seeks to recreate an obj */
        long batch;     /* reclaim batch size, 0 = default */
        unsigned long flags;
        /* These are for internal use */
        struct list_head list;
        /* objs pending delete, per node */
        atomic_long_t *nr_deferred;
};
```

- **shrink_slab()**
  - Invokes shrinkers – count_objects() to see how many are freeable, scan_objects will reclaim and age
  - Can be really inefficient because it is object rather than page based – internal fragmentation

Huge pages in Linux

- **Kernel mapping of physical memory**
  - Uses 1GB or 2MB huge pages when possible
  - Direct mapping, ioremap() for device memory ranges
- **Explicit hugepage usage – HugeTLBfs**
  - Pre-allocated in pools, accessible by several interfaces
  - Private or shared, no splitting, no swapping
  - Multiple sizes supported; page table sharing support
- **Transparent hugepage usage – THP**
  - Allocated implicitly, possible to prefer or disallow by hints
  - Anonymous, private (except fork+COW), can be split back to base pages and then swapped out
  - Shmem/tmpfs support – controlled via mount parameter

HugeTLBfs Usage

- **SysV shared memory segment**
  - `shmid = shmget(key, SIZE, SHM_HUGETLB | …);`
`addr = shmat(shmid, NULL, 0);`
  - Since 3.8: alternative flags `SHM_HUGE_2MB`, `SHM_HUGE_1GB`, and `SHM_HUGE_SHIFT`
- **Anonymous `mmap()`**
  - `addr = mmap(NULL, SIZE, PROT_*, MAP_PRIVATE | MAP_ANONYMOUS | MAP_HUGETLB, -1, 0);`
  - Sice 3.8: same alternative flags as `shmget()`
- **Mount a special virtual filesystem**
  - mount -t hugetlbfs none /dev/hugepages -o <pagesize=2M>
  - fd = open("/dev/hugepages/1", O_CREAT | O_RDWR, 0755);
addr = mmap(NULL, SIZE, PROT_*, MAP_SHARED, fd, 0);
- **Use libhugetlbfs library – man libhugetlbfs(7)**
  - get_huge_pages(), get_hugepage_region()...
  - LD_PRELOAD for legacy applications
  - Text, data, malloc(), shared memory backed by hugepages
- **Useful tools: hugeadm, hugectl**

HugeTLBfs Internals

- **Hugetlb pages reserved on mmap()**
  - Reservation system tracks
  - Cheaper mmap(), potentially better NUMA placement
- **Private mappings can fork() + COW fault at any time**
  - Potential copies not reserved – fork() won't fail
  - COW will try to allocate without reserve, but that can fail
    - Child COW alloc fails → SIGBUS
    - Parent COW alloc fails → child's mapping removed, fault → SIGBUS
- **Reservations don't guarantee NUMA placement**
- **Shared page tables**
  - Scenario: many processes mapping the same region of 2MB hugepages
  - Each 1GB large region (fully populated or not) would have 4KB pmd-level page table for each process
  - This page table will be shared when mappings are properly aligned, reducing the memory usage
  - Example: Memory usage of (system running Oracle) by page tables 150GB without vs 1GB with HugeTLB

THP

- **First page fault in each huge-page aligned part of vma (last-level page table does not yet exist)**
  - Read fault → map a shared “THP zero page” first
- **During `mmap()` with `MAP_POPULATE`**
- **Khugepaged merge small pages into THP in the background**
- **If allocating huge page fails, fallback to mapping a page table with a single PTE entry for a base page**
- **COW – alloc+copy whole huge page, fallback to alloc+copy many base pages mapped by PTEs**
- **THP may be mapped by ptes partially – mprotect, unmap**
- **Fault in resp. merging policy fine tuning**
  - madvise(MADV_HUGEPAGE, MADV_NOHUGEPAGE)
  - prctl(PR_SET_THP_DISABLE)
  - Global setting - /sys/kernel/mm/transparent_hugepage/
    - allways, madvise, never for global setting
    - allways, madvise, never, defer, defer+madvise for khugepaged
  - Shmem controlled by mount option
    - allways, advise, never, within_size

THP - khugepaged

- **Kernel thread to scan address space**
- **/sys/kernel/mm/transparent_hugepage/khugepaged/***
- **Merges sparsely populated PMD**
  - max_ptes_none – how much more to allocate
  - max_ptes_swap – how much to swap in
  - pages_to_scan, scan_sleep_millisecs – how much/often to scan
- **Pros**
  - Fault in is latency sensitive while deferred context might try harder
  - VMAs might grow over time (e.g. stack, shmem file)
  - Reduces memory fragmentation
- **Cons**
  - Background interference – e.g. mmap_sem lock contention
  - Jumps in too late for short lived mappings

THP related statistics

`/proc/meminfo`

```
AnonHugePages:   1929216 kB
ShmemHugePages, ShmemPmdMapped
```

`/proc/vmstat`

```
thp_fault_alloc 174171
thp_fault_fallback 61457
thp_collapse_alloc 35893
thp_collapse_alloc_failed 703
thp_file_alloc 0
thp_file_mapped 0
```

```
thp_split_page 5542
thp_split_page_failed 4
thp_deferred_split_page 199
thp_split_pmd 26504
thp_split_pud 0
thp_zero_page_alloc 1
thp_zero_page_alloc_failed 0
```

`/sys/kernel/mm/transparent_hugepage/khugepaged`

```
full_scans: 751
pages_collapsed: 26272
```

Memory cgroup controller

- **Hierarchical accounting of user memory (page faults) and opt-in kernel allocations `__GFP_ACCOUNT` (e.g. kernel stacks)**
- **Represented by struct mem_cgroup**
  - Page counters for charges
  - Own LRUs – mem_cgroup::nodeinfo – per NUMA node
- **Memory is charged when the page is added to the LRU list or in the page allocator for kernel allocations - `try_charge()`**
  - Charge is propagated up the hierarchy
  - Performs direct reclaim on the memcg which hits hard limit - `mem_cgroup_shrink_node()`
    - `shrink_node_memcg()` - iterates over all lruvecs under given mem_cgroup in a round robin manner
    - Code shared with the standard reclaim – some exceptions, we wait for Dirty pages, swappiness is not ignored even under hard memory pressure etc...
  - Schedules “background” reclaim when high limit is reached – `reclaim_high()` when returning to the user space
- **Low/Min limit protects memory cgroup from reclaim**

Memory cgroup controller

- **Charge fails and marks OOM context when the reclaim fails**
  - Only kills tasks from the memcg hierarchy
- **Memcg OOM killer can be handled from userspace**
  - `echo 1 > memory.oom_control` – disables oom killer, the kernel will notify listener on this file and waits for situation to change - `mem_cgroup_oom_synchronize()`
  - Admin may increase the limit or kill a task manually
- **Only page faults are triggering memcg OOM killer `pagefault_out_of_memory()`**
  - Used to trigger it from the charge path but this could deadlock easily – charge context can hold locks which might be needed for OOM to make a forward progress

Questions?

SUSE®

We adapt. You succeed.

Unpublished Work of SUSE LLC. All Rights Reserved.

This work is an unpublished work and contains confidential, proprietary and trade secret information of SUSE LLC. Access to this work is restricted to SUSE employees who have a need to know to perform tasks within the scope of their assignments. No part of this work may be practiced, performed, copied, distributed, revised, modified, translated, abridged, condensed, expanded, collected, or adapted without the prior written consent of SUSE.
Any use or exploitation of this work without authorization could subject the perpetrator to criminal and civil liability.

General Disclaimer

This document is not to be construed as a promise by any participating company to develop, deliver, or market a product. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. SUSE makes no representations or warranties with respect to the contents of this document, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. The development, release, and timing of features or functionality described for SUSE products remains at the sole discretion of SUSE. Further, SUSE reserves the right to revise this document and to make changes to its content, at any time, without obligation to notify any person or entity of such revisions or changes. All SUSE marks referenced in this presentation are trademarks or registered trademarks of Novell, Inc. in the United States and other countries. All third-party trademarks are the property of their respective owners.