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/](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
- 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 \(\rightarrow\) struct page translation
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

- Memory ranges exported by BIOS/EFI firmware
  - E820 for x86 systems
    
    ```markdown
    [ 0.0000000] e820: BIOS-provided physical RAM map:
    [ 0.0000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009dbff] usable
    [ 0.0000000] BIOS-e820: [mem 0x000000000009dc00-0x000000000009ffff] reserved
    [ 0.0000000] BIOS-e820: [mem 0x00000000000f0000-0x00000000000fffff] reserved
    [ 0.0000000] BIOS-e820: [mem 0x0000000000100000-0x0000000bf61ffff] usable
    [ 0.0000000] BIOS-e820: [mem 0x00000000000bf620000-0x00000000000bf63bfff] ACPI data
    [ 0.0000000] BIOS-e820: [mem 0x00000000000bf63c000-0x00000000000bf63cfff] usable
    [ 0.0000000] BIOS-e820: [mem 0x00000000000bf63d000-0x00000000000cfffffff] reserved
    [ 0.0000000] BIOS-e820: [mem 0x00000000000fec0000-0x00000000000fee0000] ACPI data
    [ 0.0000000] BIOS-e820: [mem 0x00000000000fee0000-0x00000000000ffffff] reserved
    [ 0.0000000] BIOS-e820: [mem 0x00000000000ffffff0-0x0000000000000000] reserved
    [ 0.0000000] BIOS-e820: [mem 0x0000000000000000-0x0000000000000000] reserved
    ```

- 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
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_to_page()` – *physical page frame number to struct page translation*
- `page_owner` – tracks stack of the allocation request – very useful for debugging
struct page {
    long unsigned int          flags;                /*     0     8 */
    union {
        struct address_space * mapping;          /*           8 */
        void *             s_mem;                /*           8 */
        atomic_t           compound_mapcount;    /*           4 */
    };                                               /*     8     8 */
    union {
        long unsigned int  index;                /*           8 */
        void *             freelist;             /*           8 */
    };                                               /*    16     8 */
    union {
        long unsigned int  counters;             /*           8 */
        struct {
            union {
                atomic_t _mapcount;      /*           4 */
                unsigned int active;     /*           4 */
                struct {
                    unsigned int inuse:16; /*    24:16  4 */
                    unsigned int objects:15; /*    24: 1  4 */
                    unsigned int frozen:1; /*    24: 0  4 */
                };                       /*           4 */
            int units;               /*           4 */
        };                          /*    24     4 */
        int         _refcount;            /*    28     4 */
    };                                           /*           8 */
    union {
        struct list_head   lru;                  /*          16 */
        struct dev_pagemap * pgmap;              /*           8 */
        struct {
            struct page * next;              /*    32     8 */
            int        pages;                /*    40     4 */
            int        pobjects;             /*    44     4 */
        };                                       /*          16 */
        struct callback_head callback_head;      /*          16 */
        struct {
            long unsigned int compound_head; /*    32     8 */
            unsigned int compound_dtor;      /*    40     4 */
            unsigned int compound_order;     /*    44     4 */
        };                                       /*          16 */
        struct {
            long unsigned int __pad;         /*    32     8 */
            pgtable_t  pmd_huge_pte;         /*    40     8 */
        };                                       /*          16 */
    };                                               /*    32    16 */
    union {
        long unsigned int  private;              /*           8 */
        spinlock_t *       ptl;                  /*           8 */
        struct kmem_cache * slab_cache;          /*           8 */
    };                                               /*    48     8 */
    struct mem_cgroup *        mem_cgroup;           /*    56     8 */
    /* size: 64, cachelines: 1, members: 7 */
}
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
    ```c
    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`
struct pglist_data {
  struct zone node_zones[4];  /* 0  6400 */
  struct zonelist node_zonelists[2];  /* 6400  8224 */
  int nr_zones;  /* 14624  4 */
  spinlock_t node_size_lock;  /* 14632  56 */
  long unsigned int node_start_pfn;  /* 14688  8 */
  long unsigned int node_present_pages;  /* 14696  8 */
  long unsigned int node_spanned_pages;  /* 14704  8 */
  int node_id;  /* 14712  4 */
  wait_queue_head_t kswapd_wait;  /* 14720  72 */
  wait_queue_head_t pfmemalloc_wait;  /* 14792  72 */
  struct task_struct * kswapd;  /* 14864  8 */
  int kswapd_order;  /* 14872  4 */
  enum zone_type kswapd_classzone_idx;  /* 14876  4 */
  int kswapd_failures;  /* 14880  4 */
  int kcompactd_max_order;  /* 14884  4 */
  enum zone_type kcompactd_classzone_idx;  /* 14888  4 */
  wait_queue_head_t kcompactd_wait;  /* 14896  72 */
  struct task_struct * kcompactd;  /* 14968  8 */
  long unsigned int totalreserve_pages;  /* 14976  8 */
  long unsigned int min_unmapped_pages;  /* 14984  8 */
  long unsigned int min_slab_pages;  /* 14992  8 */
  struct zone_padding _pad1_;  /* 15040  0 */
  spinlock_t lru_lock;  /* 15040  56 */
  spinlock_t split_queue_lock;  /* 15096  56 */
  struct list_head split_queue;  /* 15152  16 */
  long unsigned int split_queue_len;  /* 15168  8 */
  struct lruvec lruvec;  /* 15176  136 */
  unsigned int inactive_ratio;  /* 15312  4 */
  long unsigned int flags;  /* 15320  8 */
  struct zone_padding _pad2_;  /* 15360  0 */
  struct per_cpu_nodestat * per_cpu_nodestats;  /* 15360  8 */
  atomic_long_t vm_stat[25];  /* 15368  200 */
  /* size: 15616, cachelines: 244, members: 32 */
ACPI: SRAT: Node 0 PXM 0 [mem 0x00000000-0xbfffffff]
ACPI: SRAT: Node 0 PXM 0 [mem 0x100000000-0x103fffffff]
ACPI: SRAT: Node 1 PXM 1 [mem 0x1040000000-0x203fffffff]
ACPI: SRAT: Node 2 PXM 2 [mem 0x2040000000-0x303fffffff]
ACPI: SRAT: Node 3 PXM 3 [mem 0x3040000000-0x403fffffff]
NUMA: Node 0 [mem 0x00000000-0xbfffffff] + [mem 0x100000000-0x103fffffff] -> [mem 0x00000000-0x103fffffff]
NODE_DATA(0) allocated [mem 0x103ffde000-0x103fffffff]
NODE_DATA(1) allocated [mem 0x203ffde000-0x203fffffff]
NODE_DATA(2) allocated [mem 0x303ffde000-0x303fffffff]
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 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[]`
struct zone {
    long unsigned int          watermark[3];         /*     0    24 */
    long unsigned int          nr_reserved_highatomic; /*     24     8 */
    long int                   lowmem_reserve[4];    /*     32     32 */
    int                        node;                 /*     64     4 */
    struct pglist_data *       zone_pgdat;           /*     72     8 */
    struct per_cpu_pageset *   pageset;              /*     80     8 */
    long unsigned int          zone_start_pfn;       /*     88     8 */
    long unsigned int          managed_pages;        /*     96     8 */
    long unsigned int          spanned_pages;        /*    104     8 */
    long unsigned int          present_pages;        /*    112     8 */
    const char  *              name;                 /*    120     8 */
    long unsigned int          nr_isolate_pageblock; /*    128     8 */
    seqlock_t                  span_seqlock;         /*    136     96 */
    int                        initialized;          /*    232     4 */
    struct zone_padding        _pad1_;               /*    256     0 */
    struct free_area           free_area[11];        /*    256   968 */
    long unsigned int          flags;                /*   1224     8 */
    spinlock_t                 lock;                 /*   1232    56 */
    struct zone_padding        _pad2_;               /*   1344     0 */
    long unsigned int          percpu_drift_mark;    /*   1344     8 */
    long unsigned int          compact_cached_free_pfn; /*   1352     8 */
    long unsigned int          compact_cached_migrate_pfn[2]; /*   1360     16 */
    unsigned int               compact_considered;   /*   1376     4 */
    unsigned int               compact_defer_shift;  /*   1380     4 */
    int                        compact_order_failed; /*   1384     4 */
    bool                       compact_blockskip_flush; /*   1388     1 */
    bool                       contiguous;           /*   1389     1 */
    struct zone_padding        _pad3_;               /*   1408     0 */
    atomic_long_t              vm_stat[20];          /*   1408  160 */
/* size: 1600, cachelines: 25, members: 29 */
Virtual Memory representation

- 47b (64TB) view of contiguous memory which is translated to the physical memory by page tables
- 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 – BFFFFFFFF user space
        - C000000 – F7xxxxxx kernel (direct mapping)
        - F7xxxx – FF7FE0000 vmalloc
        - FF80000 – FFC000000 kmap
  - 64b – negative address space kernel, positive userspace
    - 0000000000000000 – 00007FFFFFFFFF – user space
    - FFFFFFFF800000000000 – FFFFFFFFC7FFFFFFFFF – direct mapping
    - FFFFFFFFC900000000000 – FFFFFFFFE8FFFFFFFFF – 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 4 page table levels**
  - `pgd_t`, `pud_t`, `pmd_t` and `pte_t`
  - `pgd_alloc`, `pgd_none`, `pgd_index`, `pgd_offset` etc...
  - Architectures with a different pte topology emulate 4 levels (e.g. include/asm-generic/4level-fixup.h)

• **Simple page table walk**
  ```c
  pgd = pgd_offset(mm, addr) /* mm of the process or init_mm */
  pud = pud_offset(pgd, 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

• **5-level page tables under development**
  - Increase of the address space to 52b address space
  - P4d additional level under pgd
Address space descriptor

- Each process has its address space descriptor `struct mm_struct`
- Keeps track of all the mapped 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
struct mm_struct {
    struct vm_area_struct * mmap; /* 0 8 */
    struct rb_root mm_rb; /* 8 8 */
    u32 vmacache_seqnum; /* 16 4 */
    long unsigned int (*get_unmapped_area)(struct file *, long unsigned int, long unsigned int, long unsigned int, long unsigned int); /* 24 8 */
    long unsigned int mmap_base; /* 32 8 */
    long unsigned int mmap_legacy_base; /* 40 8 */
    long unsigned int mmap_compat_base; /* 48 8 */
    long unsigned int mmap_compat_legacy_base; /* 56 8 */
    long unsigned int task_size; /* 64 8 */
    long unsigned int highest_vm_end; /* 72 8 */
    pgd_t * pgd; /* 80 8 */
    atomic_t mm_users; /* 88 4 */
    atomic_t mm_count; /* 92 4 */
    atomic_long_t nr_ptes; /* 96 8 */
    atomic_long_t nr_pmds; /* 104 8 */
    int map_count; /* 112 4 */
    spinlock_t page_table_lock; /* 120 56 */
    struct rw_semaphore mmap_sem; /* 176 128 */
    struct list_head mmlist; /* 304 16 */
    long unsigned int hiwater_rss; /* 320 8 */
    long unsigned int hiwater_vm; /* 328 8 */
    long unsigned int total_vm; /* 336 8 */
    long unsigned int locked_vm; /* 344 8 */
    long unsigned int pinned_vm; /* 352 8 */
    long unsigned int data_vm; /* 360 8 */
    long unsigned int exec_vm; /* 368 8 */
    long unsigned int stack_vm; /* 376 8 */
    long unsigned int def_flags; /* 384 8 */
    long unsigned int start_code; /* 392 8 */
    long unsigned int end_code; /* 400 8 */
    long unsigned int start_data; /* 408 8 */
    long unsigned int end_data; /* 416 8 */
    long unsigned int start_brk; /* 424 8 */
    long unsigned int brk; /* 432 8 */
    long unsigned int start_stack; /* 440 8 */
    long unsigned int arg_start; /* 448 8 */
    long unsigned int arg_end; /* 456 8 */
    long unsigned int env_start; /* 464 8 */
    long unsigned int env_end; /* 472 8 */
    long unsigned int saved_auxv[46]; /* 480 368 */
    struct mm_rss_stat rss_stat; /* 848 32 */
    struct linux_binfmt * binfmt; /* 880 8 */
    cpumask_var_t cpu_vm_mask_var; /* 888 8 */
    mm_context_t context; /* 896 16 */
    long unsigned int flags; /* 1064 8 */
    struct core_state * core_state; /* 1072 8 */
    spinlock_t ioctx_lock; /* 1080 56 */
    struct kioctx_table * ioctx_table; /* 1136 8 */
    struct task_struct * owner; /* 1144 8 */
    struct user_namespace * user_ns; /* 1152 8 */
    struct file * exe_file; /* 1160 8 */
    struct mmu_notifier_mm * mmu_notifier_mm; /* 1168 8 */
    bool tlb_flush_pending; /* 1176 1 */
    struct uprobes_state uprobes_state; /* 1184 8 */
    atomic_long_t hugetlb_usage; /* 1192 8 */
    struct work_struct async_put_work; /* 1200 64 */
    /* size: 1264, cachelines: 20, members: 56 */
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 mapping – `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
struct vm_area_struct {
    long unsigned int vm_start; /* 0 8 */
    long unsigned int vm_end;   /* 8 8 */
    struct vm_area_struct * vm_next; /* 16 8 */
    struct vm_area_struct * vm_prev; /* 24 8 */
    struct rb_node vm_rb; /* 32 24 */
    long unsigned int rb_subtree_gap; /* 56 8 */
    struct mm_struct * vm_mm; /* 64 8 */
    pgprot_t vm_page_prot; /* 72 8 */
    long unsigned int vm_flags; /* 80 8 */
    struct {
        struct rb_node rb; /* 88 24 */
        long unsigned int rb_subtree_last; /* 112 8 */
    } shared; /* 88 32 */
    struct list_head anon_vma_chain; /* 120 16 */
    struct anon_vma * anon_vma; /* 136 8 */
    const struct vm_operations_struct * vm_ops; /* 144 8 */
    long unsigned int vm_pgoff; /* 152 8 */
    struct file * vm_file; /* 160 8 */
    void * vm_private_data; /* 168 8 */
    struct mempolicy * vm_policy; /* 176 8 */
    struct vm_userfaultfd_ctx vm_userfaultfd_ctx; /* 184 0 */
} /* size: 184, cachelines: 3, members: 18 */
struct vm_operations_struct {
    void (*open)(struct vm_area_struct *);
    void (*close)(struct vm_area_struct *);
    int (*mremap)(struct vm_area_struct *);
    int (*fault)(struct vm_fault *);
    int (*huge_fault)(struct vm_fault *, enum page_entry_size);
    void (*map_pages)(struct vm_fault *, long unsigned int, long unsigned int);
    int (*page_mkwrite)(struct vm_fault *);
    int (*pfn_mkwrite)(struct vm_fault *);
    int (*access)(struct vm_area_struct *, long unsigned int, void *, int, int);
    const char  * (*name)(struct vm_area_struct *);
    int (*set_policy)(struct vm_area_struct *, struct mempolicy *);
    struct mempolicy *(get_policy)(struct vm_area_struct *, long unsigned int);
    struct page *(find_special_page)(struct vm_area_struct *, long unsigned int);
}
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
struct address_space {
    struct inode * host; /* 0 8 */
    struct radix_tree_root page_tree; /* 8 16 */
    spinlock_t tree_lock; /* 24 56 */
    atomic_t i_mmap_writable; /* 80 4 */
    struct rb_root i_mmap; /* 88 8 */
    struct rw_semaphore i_mmap_rwsem; /* 96 128 */
    long unsigned int nrpages; /* 224 8 */
    long unsigned int nrexceptional; /* 232 8 */
    long unsigned int writeback_index; /* 240 8 */
    const struct address_space_operations * a_ops; /* 248 8 */
    long unsigned int flags; /* 256 8 */
    spinlock_t private_lock; /* 264 56 */
    gfp_t gfp_mask; /* 320 4 */
    struct list_head private_list; /* 328 16 */
    void * private_data; /* 344 8 */

    /* size: 352, cachelines: 6, members: 15 */
Address space – gluing it together

struct anon_vma

struct anon_vma_chain

struct page

mapping & ~PAGE_MAPPING_FLAGS

i_mmap

struct address_space

index

struct inode

struct file

mm_struct

VMA

VMA

VMA

vm_file

f_inode

i_mapping

index

struct page
Page cache management

- **address_space::page_tree** - radix_tree of pages belonging to the inode
- **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)
- 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
  - $2 \times 2^{N-1}$ 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 is below 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 are below 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`
  ```c
  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_unrefault()`
- 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
    ```
    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
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 limit protects memory cgroup from reclaim
struct mem_cgroup {
    struct cgroup_subsys_state css;    /* 0 216 */
    struct mem_cgroup_id id;          /* 216 8 */
    struct page_counter memory;       /* 224 40 */
    struct page_counter swap;         /* 264 40 */
    struct page_counter memsw;        /* 304 40 */
    struct page_counter kmem;         /* 344 40 */
    struct page_counter tcpmem;       /* 384 40 */
    long unsigned int low;            /* 424 8 */
    long unsigned int high;           /* 432 8 */
    struct work_struct high_work;     /* 440 64 */
    long unsigned int soft_limit;     /* 504 8 */
    struct vm_pressure vm_pressure;   /* 512 296 */
    bool use_hierarchy;               /* 808 1 */
    bool oom_lock;                    /* 809 1 */
    int under_oom;                    /* 812 4 */
    int swappiness;                   /* 816 4 */
    int oom_kill_disable;             /* 820 4 */
    struct cgroup_file events_file;   /* 824 8 */
    struct mutex thresholds_lock;     /* 832 128 */
    struct mem_cgroup_thresholds thresholds;  /* 960 16 */
    struct mem_cgroup_thresholds memsw_thresholds; /* 976 16 */
    struct list_head oom_notify;      /* 992 16 */
    long unsigned int move_charge_at_immigrate; /* 1008 8 */
    atomic_t moving_account;          /* 1016 4 */
    spinlock_t move_lock;             /* 1024 56 */
    struct task_struct * move_lock_task; /* 1080 8 */
    long unsigned int move_lock_flags; /* 1088 8 */
    struct mem_cgroup_stat_cpu * stat; /* 1096 8 */
    long unsigned int socket_pressure; /* 1104 8 */
    bool tcpmem_active;               /* 1112 1 */
    int tcpmem_pressure;              /* 1116 4 */
    int kmemcg_id;                    /* 1120 4 */
    enum memcg_kmem_state kmem_state; /* 1124 4 */
    struct list_head kmem_caches;     /* 1128 16 */
    int last_scanned_node;           /* 1144 4 */
    nodemask_t scan_nodes;            /* 1152 8 */
    atomic_t numainfo_events;         /* 1160 4 */
    atomic_t numainfo_updating;       /* 1164 4 */
    struct list_head event_list;      /* 1168 16 */
    spinlock_t event_list_lock;       /* 1184 56 */
    struct mem_cgroup_per_node * nodeinfo[0]; /* 1240 0 */
} /* size: 1240, cachelines: 20, members: 41 */
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?
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.