Page cache and Page fault

Dongyoon Lee – Lecture notes from Dr. Min @ VT
Summary of last lectures

- Tools: building, exploring, and debugging Linux kernel
- Core kernel infrastructure
- Process management & scheduling
- Interrupt & interrupt handler
- Kernel synchronization
- Memory management
- Virtual file system
Today: page cache and page fault

• Introduction to cache
• Page cache in Linux
• Cache eviction
• Interaction with memory management
• Flusher daemon
Latency numbers

L1 cache reference ............................... 0.5 ns
Branch mispredict ............................... 5 ns
L2 cache reference ............................... 7 ns
Mutex lock/unlock ............................... 25 ns
Main memory reference ........................... 100 ns
Compress 1K bytes with Zippy .................. 3,000 ns = 3 µs
Send 2K bytes over 1 Gbps network ........... 20,000 ns = 20 µs
SSD random read ................................. 150,000 ns = 150 µs
Read 1 MB sequentially from memory ......... 250,000 ns = 250 µs
Round trip within same datacenter ............ 500,000 ns = 0.5 ms
Read 1 MB sequentially from SSD* ............ 1,000,000 ns = 1 ms
Disk seek ........................................... 10,000,000 ns = 10 ms
Read 1 MB sequentially from disk ............. 20,000,000 ns = 20 ms
Send packet CA->Netherlands->CA .......... 150,000,000 ns = 150 ms

• Source: Latency numbers every programmer should know
### Humanized version (x 1,000,000,000)

<table>
<thead>
<tr>
<th>Activity</th>
<th>Time</th>
<th>Human Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>0.5 s</td>
<td>One heart beat (0.5 s)</td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>5 s</td>
<td>Yawn</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7 s</td>
<td>Long yawn</td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>25 s</td>
<td>Making a coffee</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100 s</td>
<td>Brushing your teeth</td>
</tr>
<tr>
<td>Compress 1K bytes with Zippy</td>
<td>50 min</td>
<td>One episode of a TV show (including ad breaks)</td>
</tr>
<tr>
<td>Send 2K bytes over 1 Gbps network</td>
<td>5.5 hr</td>
<td>From lunch to end of work day</td>
</tr>
<tr>
<td>SSD random read</td>
<td>1.7 days</td>
<td>A normal weekend</td>
</tr>
<tr>
<td>Read 1 MB sequentially from memory</td>
<td>2.9 days</td>
<td>A long weekend</td>
</tr>
<tr>
<td>Round trip within same datacenter</td>
<td>5.8 days</td>
<td>A medium vacation</td>
</tr>
<tr>
<td>Read 1 MB sequentially from SSD</td>
<td>11.6 days</td>
<td>Waiting for almost 2 weeks for a delivery</td>
</tr>
<tr>
<td>Disk seek</td>
<td>16.5 weeks</td>
<td>A semester in university</td>
</tr>
<tr>
<td>Read 1 MB sequentially from disk</td>
<td>7.8 months</td>
<td>Almost producing a new human being</td>
</tr>
<tr>
<td>The above 2 together</td>
<td>1 year</td>
<td>Average time it takes</td>
</tr>
<tr>
<td>Send packet CA-&gt;Netherlands-&gt;CA</td>
<td>4.8 years</td>
<td>to complete a bachelor's degree</td>
</tr>
</tbody>
</table>
Why caching

- Disk access is several orders of magnitude slower than memory access
- Data accessed once will, with a high likelihood, find itself accessed again in the near future → temporal locality
Page cache (or buffer cache)

- Physical pages in RAM holding disk content (blocks)
  - Disk is called a *backing store*
  - Works for regular files, memory mapped files, and block device files
- Dynamic size
  - Grows to consume free memory unused by kernel and processes
  - Shrinks to relieve memory pressure
Page cache (or buffer cache)

- Buffered IO operations (without `O_DIRECT`), the page cache of a file is first checked.
- **Cache hit**: if data is in the page cache, copy from/to user memory.
- **Cache miss**: otherwise, VFS asks the concrete file system (e.g., ext4) to read data from disk.
  - Read/write operations populate the page cache.
Write caching policies

- **No-write**: does not cache write operations
- **Write-through**: write operations immediately go through to disk
  - Keeping the cache coherent
  - No need to invalidate cached data → simple
- **Write-back**: write operations update page cache but disk is not immediately updated → **Linux page cache policy**
  - Pages written are marked *dirty* using a tag in radix tree
  - Periodically, write dirty pages to disk → *writeback*
  - Page cache absorbs temporal locality to reduce disk access
Write caching policies

No-write

Application → Cache → Disk

Write-through

Application → Cache → Disk

Write-back

Application → Cache → (later) → Disk
Cache eviction

• When data should be removed from the cache?
  • Need more free memory (memory pressure)

• Which data should be removed from the cache?
  • Ideally, evict cache pages that will not be accessed in the future
  • Eviction policy: deciding what to evict
Eviction policy: LRU

• **Least recently used (LRU) policy**
  • Keep track of when each page is accessed
  • Evict the pages with the oldest timestamp

• Failure cases of LRU policy
  • Many files are accessed once and then never again
  • LRU puts them at the top of LRU list → not optimal
The two-list strategy

- **Active list**
  - Pages in the active list is considered *hot*
  - Not available for eviction

- **Inactive list**
  - Pages in the inactive list is considered *cold*
  - Available for eviction
The two-list strategy

- Newly accessed pages are added to inactive list
- If a page in an inactive list is accessed again, it is promoted to an active list
  - When a page is moved to an inactive list, its access permission in a page table is removed.
- If an active list becomes much larger than an inactive list, items from the active list’s head are moved back to the inactive list.
- *When a page is added to inactive list, its access permission in the page table is disabled to track its access.*
The two-list strategy

Two-List or LRU/2 strategy

Active list: hot pages, not available for eviction

Inactive list

Evict from inactive list only
The two-list strategy

Two-List or LRU/2 strategy

Active list

Inactive list

Accessed pages not in the list are added to the inactive list
The two-list strategy

Two-List or LRU/2 strategy

Active list

Inactive list

Inactive page accessed are added to the active list
The two-list strategy

Lists are balanced and active pages are evicted in the inactive list
The Linux page cache (or buffer cache)

/* linux/include/linux/fs.h */
struct inode {
    const struct inode_operations   *i_op;
    struct super_block              *i_sb;
    struct address_space            *i_mapping;
    unsigned long                   i_ino;
};

struct address_space {
    struct inode                    *host;      /* owner: inode, block_device */
    struct radix_tree_root          page_tree;  /* radix tree of all pages */
    spinlock_t                      tree_lock;  /* and lock protecting it */
};

/* Insert an item into the radix tree at position @index. */
int radix_tree_insert(struct radix_tree_root *root,
                      unsigned long index, void *item);

/* linux/mm/shmem.c */
static int shmem_add_to_page_cache(struct page *page,
                                     struct address_space *mapping, pgoff_t index, void *expected)
{
    error = radix_tree_insert(&mapping->page_tree, index, page);
}
The Linux page cache (or buffer cache)

$> sudo cat /proc/1/maps
7fe87b1f1000-7fe87b21d000 r-xp 00000000 fd:00 1975147 /usr/lib64/libseccomp.so
7fe87b21d000-7fe87b41c000 ---p 0002c000 fd:00 1975147 /usr/lib64/libseccomp.so
7fe87b41c000-7fe87b431000 r--p 0002b000 fd:00 1975147 /usr/lib64/libseccomp.so
7fe87b431000-7fe87b432000 rw-p 00040000 fd:00 1975147 /usr/lib64/libseccomp.so
7fe87b432000-7fe87b439000 r-xp 00000000 fd:00 1975989 /usr/lib64/librt-2.26.so
7fe87b439000-7fe87b638000 ---p 00007000 fd:00 1975989 /usr/lib64/librt-2.26.so
7fe87b638000-7fe87b639000 r--p 00006000 fd:00 1975989 /usr/lib64/librt-2.26.so
7fe87b639000-7fe87b63a000 rw-p 00007000 fd:00 1975989 /usr/lib64/librt-2.26.so

• Q: the number of **vm_area_struct**
• Q: the number of **inode**
• Q: the number of **address_space**
address_space

- An entity present in the page cache
  - an address_space = a file = accessing a page cache of a file
  - an address_space = one or more vm_area_struct
/* linux/include/linux/fs.h */
struct address_space {
    struct inode *host;    /* owning inode */
    struct radix_tree_root page_tree; /* radix tree of all pages */
    spinlock_t tree_lock; /* page tree lock */
    unsigned int i_mmap_writable; /* VM_SHARED (writable) */
    struct rb_root i_mmap; /* list of all mappings */
    unsigned long nrpages; /* total number of pages */
    pgoff_t writeback_index; /* writeback start offset */
    struct address_space_operations a_ops; /* operations table */
    unsigned long flags; /* error flags */
    gfp_t gfp_mask; /* gfp mask for allocation */
    struct backing_dev_info backing_dev_info; /* read-ahead info */
    spinlock_t private_lock; /* private lock */
    struct list_head private_list; /* private list */
    struct address_space assoc_mapping; /* associated buffers */
}
address_space

- i_mmap: all shared and private mappings concerning this address space
- nrpages: total number of pages in the address space
- host: points to the inode of the corresponding file
- a_ops: address space operations
address_space_operations

/* linux/include/linux/fs.h */
struct address_space_operations {
    int (*writepage)(struct page *page, struct writeback_control *wbc);
    int (*readpage)(struct file *, struct page *);
    int (*writepages)(struct address_space *, struct writeback_control *);
    int (*set_page_dirty)(struct page *page);
    int (*readpages)(struct file *filp, struct address_space *mapping,
                     struct list_head *pages, unsigned nr_pages);
    int (*write_begin)(struct file *, struct address_space *mapping,
                       loff_t pos, unsigned len, unsigned flags,
                       struct page **pagep, void **fsdata);
    int (*write_end)(struct file *, struct address_space *mapping,
                     loff_t pos, unsigned len, unsigned copied,
                     struct page *page, void *fsdata);

    /* ... */
};
Page read operation

- read() function from the file_operations
  - generic_file_buffered_read()
- Search the data in the page cache
  - page = find_get_page(mapping, index)
- Adding the page to the page cache
  - page = __page_cache_alloc(gfp_mask);
- Then, read data from disk
  - mapping->a_ops->readpage(filp, page)
Page write operation

- When a page is modified in the page cache, mark it as dirty
  - `SetPageDirty(page)`
- Default write path: in `mm/filemap.c`

```c
/* search the page cache for the desired page. If the page is not present, 
an entry is allocated and added: */
page = __grab_cache_page(mapping, index, &cached_page, &lru_pvec);
/* Set up the write request: */
status = a_ops->write_begin(file, mapping, pos, bytes, flags, &page, &fsdata);
/* Copy data from user-space into a kernel buffer: */
copied = iov_iter_copy_from_user_atomic(page, i, offset, bytes);
/* write data to disk: */
status = a_ops->write_end(file, mapping, pos, bytes, copied, page, fsdata);
```
Interaction with memory management

• **file, file_operations**
  - How to access the contents of a file

• **address_space, address_space_operations**
  - How to access the page cache of a file

• **vm_area_struct, vm_operations_struct**
  - How to handle page fault of a virtual memory region

• Page table in x86 processor
struct file {
    struct path f_path;            /* contains the dentry */
    struct file_operations *f_op;   /* operations */
    spinlock_t f_lock;             /* lock */
    atomic_t f_count;             /* usage count */
    unsigned int f_flags;          /* open flags */
    mode_t f_mode;                /* file access mode */
    logg_t f_pos;                 /* file offset */
    struct fown_struct f_owner;   /* owner data for signals */
    const struct cred *f_cred;    /* file credentials */
    struct file_ra_state f_ra;    /* read-ahead state */
    u64 f_version;                /* version number */
    void *private_data;           /* private data */
    struct list_head f_ep_link;   /* list of epoll links */
    spinlock_t f_ep_lock;         /* epoll lock */
    struct address_space *f_mapping; /* page cache mapping */
/* ... */
};
/* linux/include/linux/fs.h */
struct file_operations {
  struct module *owner;
  loff_t (*llseek) (struct file *, loff_t, int);
  ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
  ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
  ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
  ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
  int (*iterate) (struct file *, struct dir_context *);
  int (*iterate_shared) (struct file *, struct dir_context *);
  unsigned int (*poll) (struct file *, struct poll_table_struct *);
  /* ... */
};
struct address_space {
    struct inode                *host;            /* owning inode */
    struct radix_tree_root      page_tree;        /* radix tree of all pages */
    spinlock_t                  tree_lock;        /* page tree lock */
    unsigned int                i_mmap_writable;  /* VM_SHARED (writable)
                                      * mapping count */
    struct rb_root              i_mmap;           /* list of all mappings */
    unsigned long               nrpages;          /* total number of pages */
    pgoff_t                     writeback_index;  /* writeback start offset */
    struct address_space_operations a_ops;        /* operations table */
    unsigned long               flags;            /* error flags */
    gfp_t                       gfp_mask;         /* gfp mask for allocation */
    struct backing_dev_info     backing_dev_info; /* read-ahead info */
    spinlock_t                  private_lock;     /* private lock */
    struct list_head            private_list;     /* private list */
    struct address_space        assoc_mapping;    /* associated buffers */
    /* ... */
}
/* linux/include/linux/fs.h */
/ * linux/include/linux/fs.h */
struct address_space_operations {
    int (*writepage)(struct page *page, struct writeback_control *wbc);
    int (*readpage)(struct file *, struct page *);
    int (*writepages)(struct address_space *, struct writeback_control *);
    int (*set_page_dirty)(struct page *page);
    int (*readpages)(struct file *filp, struct address_space *mapping, 
        struct list_head *pages, unsigned nr_pages);
    int (*write_begin)(struct file *, struct address_space *mapping, 
        loff_t pos, unsigned len, unsigned flags, 
        struct page **pagep, void **fsdata);
    int (*write_end)(struct file *, struct address_space *mapping, 
        loff_t pos, unsigned len, unsigned copied, 
        struct page *page, void *fsdata);
    /* ... */
};

• **Q: what is the difference between** file->read() **and asop->readpage()?** See linux/fs/ext4/file.c
struct vm_area_struct {
    struct mm_struct *vm_mm; /* associated address space */
    unsigned long vm_start; /* VMA start, inclusive */
    unsigned long vm_end; /* VMA end, exclusive */
    struct vm_area_struct *vm_next; /* list of VMAs */
    struct vm_area_struct *vm_prev; /* list of VMAs */
    pgprot_t vm_page_prot; /* access permissions */
    unsigned long vm_flags; /* flags */
    struct rb_node vm_rb; /* VMA node in the tree */
    struct list_head anon_vma_chain; /* list of anonymous mappings */
    struct anon_vma *anon_vma; /* anonymous vma object */
    struct vm_operation_struct *vm_ops; /* operations */
    unsigned long vm_pgoff; /* offset within file */
    struct file *vm_file; /* mapped file (can be NULL) */
    void *vm_private_data; /* private data */
}
/* ... */
/* linux/include/linux/mm.h */
struct vm_operations_struct {
    /* called when the area is added to an address space */
    void (*open)(struct vm_area_struct * area);

    /* called when the area is removed from an address space */
    void (*close)(struct vm_area_struct * area);

    /* invoked by the page fault handler when a page that is
     * not present in physical memory is accessed*/
    int (*fault)(struct vm_area_struct *vma, struct vm_fault *vmf);

    /* invoked by the page fault handler when a previously read-only
     * page is made writable */
    int (*page_mkwrite)(struct vm_area_struct *vma, struct vm_fault *vmf);
    /* ... */
};
**vm_area_struct** - page table
Page cache - physical page
Page fault handling

- Entry point: `handle_pte_fault` (mm/memory.c)
- Identify which VMA faulting address falls in
- Identify if VMA has a registered fault handler
- Default fault handlers
  - `do_anonymous_page`: no page and no file
  - `filemap_fault`: page backed by file
  - `do_wp_page`: write protected page (CoW)
  - `do_swap_page`: page backed by swap
File-mapped page fault: \texttt{filemap\_fault}

- PTE entry does not exist (---)
- BUT VMA is marked as accessible (e.g., rwx) and has an associated file (\texttt{vm\_file})
- Page fault handler notices differences
  - In \texttt{filemap\_fault}
  - Look up a page cache of the file
  - If cache hit, map the page in the cache
  - Otherwise, \texttt{mapping->a\_ops->readpage(file, page)}
Copy on Write: `do_wp_page`

• PTE entry is marked as un-writable (e.g., `r--`)
• But VMA is marked as writable (e.g., `rw-`)
• Page fault handler notices differences
  • In `do_wp_page`
  • Must mean CoW
  • Make a duplicate of physical page
  • Update PTEs and flush TLB entry
Flusher daemon

- Write operation are deferred, data is marked *dirty*
  - RAM data is out-of-sync with the storage media
- Dirty page writeback occurs
  - Free memory is low and the page cache needs to shrink
  - Dirty data grows older than a specific threshold
  - User process calls `sync()` or `fsync()`
- Multiple *flusher threads* are in charge of syncing dirty pages from the page cache to disk
Flusher daemon

- When the free memory goes below a given threshold, the kernel
  \texttt{wakeup\_flusher\_threads()}
  - Wakes up one or several flusher threads performing writeback
    through \texttt{bdi\_writeback\_all}
- Thread write data to disk until
  - \texttt{num\_pages\_to\_write} have been written
  - and the amount of memory drops below the threshold
- percentage of total memory to trigger flusher daemon
  - \texttt{/proc/sys/vm/dirty\_background\_ratio}
Flusher daemon

- At boot time a timer is initialized to wake up a flusher thread calling `wb_writeback()`
- Writes back all data older than a given value
  - `/proc/sys/vm/dirty_expire_interval`
- Timer reinitialized to expire at a given time in the future: now + period
  - `/proc/sys/vm/dirty_writeback_interval`
- Multiple other parameters related to the writeback and the control of the page cache in general are present in `/proc/sys/vm`
  - More info: Documentation/sysctl/vm.txt
Further readings

- Latency numbers every programmer should know
- LWN: Better active/inactive list balancing
- LWN: Flushing out pdflush
- LWN: User-space page fault handling
- W4118 @ Columbia University