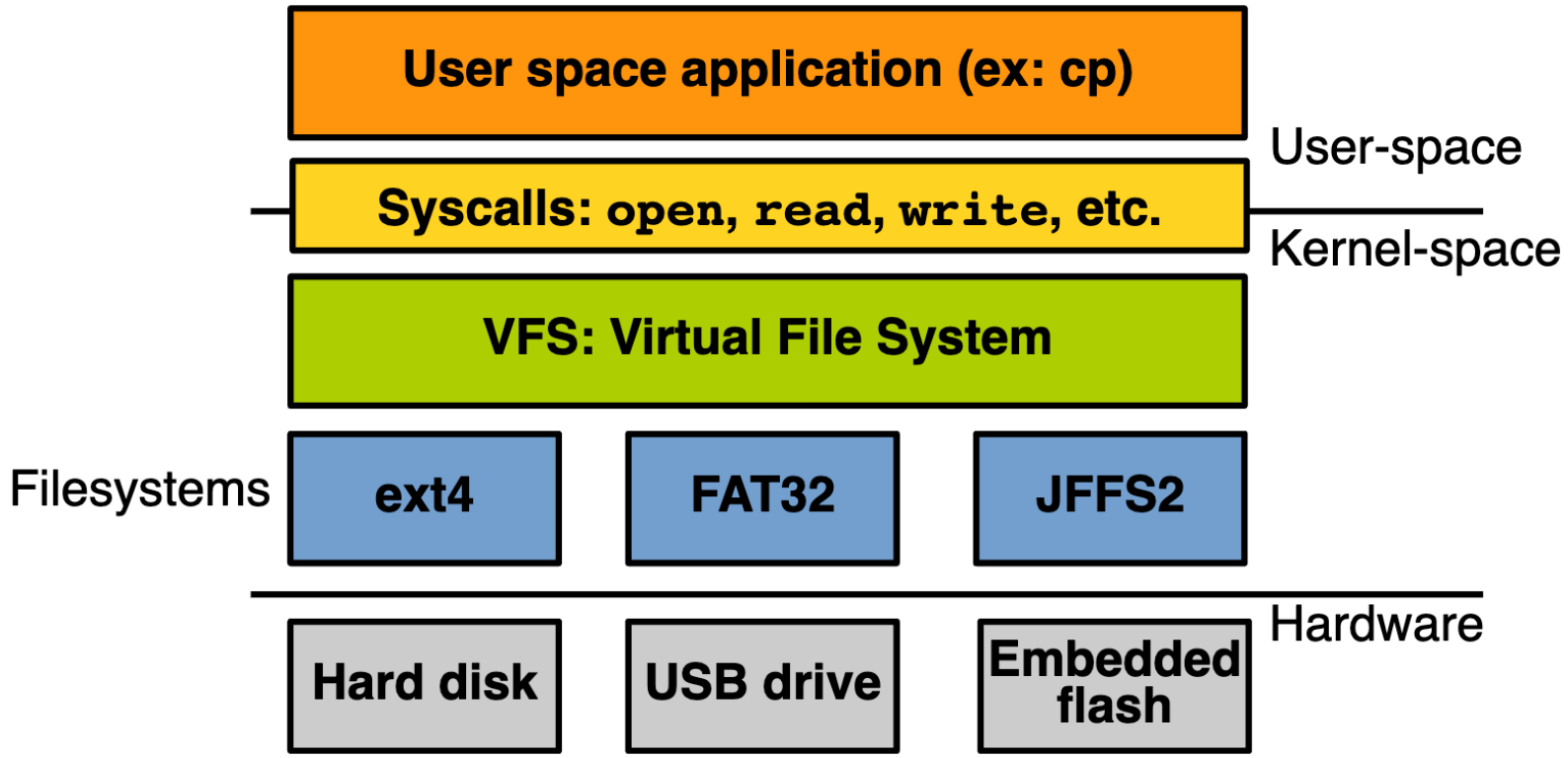
Recap: The Virtual File System (VFS)



Recap: VFS data structures

superblock object: represents a specific mounted filesystem

inode object: represents a specific file

dentry: contains file/directory name and hierarchical links defining the filesystem directory tree

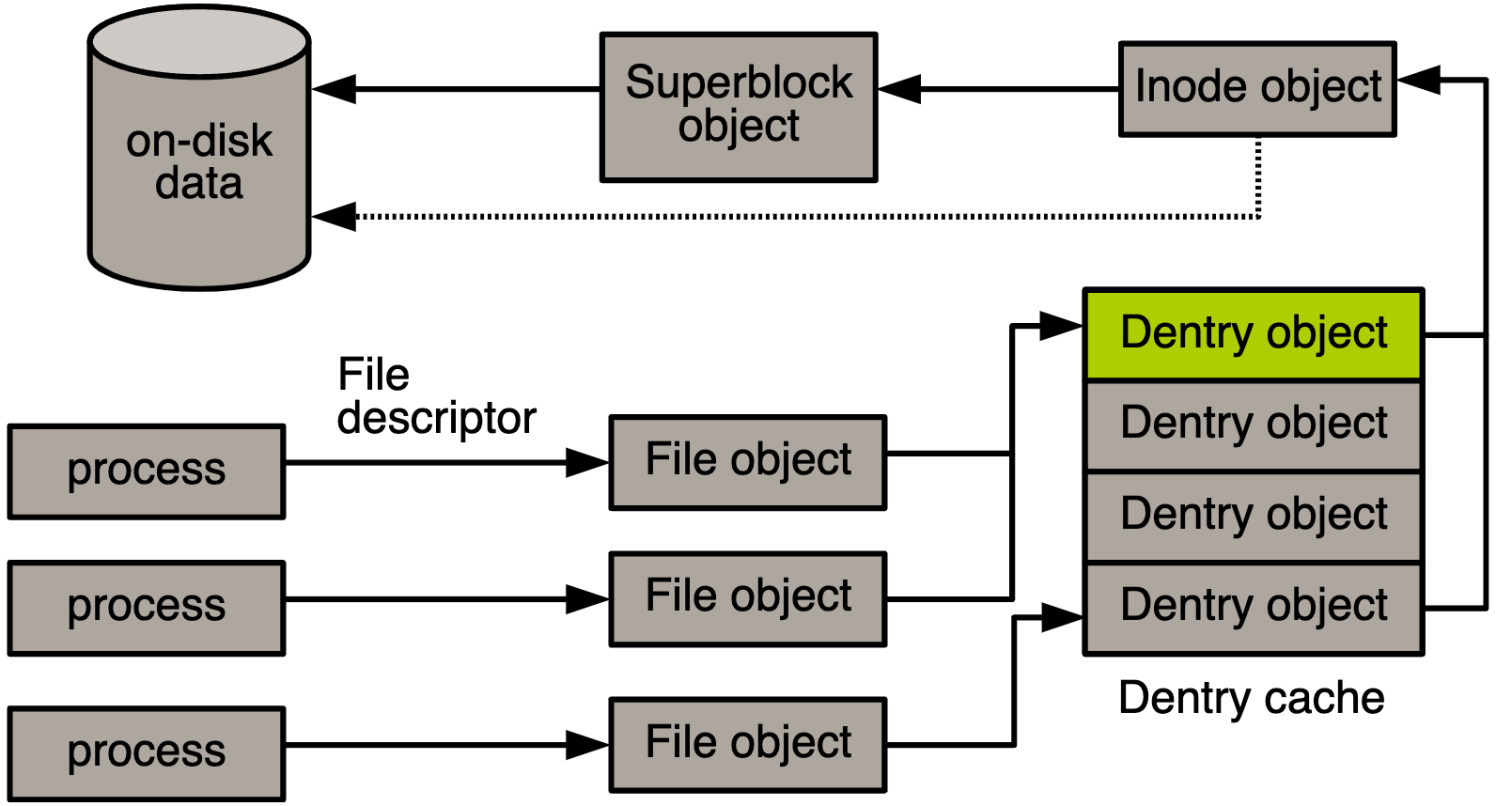
file: represents an opened file associated with a process

file_system_type: contains information about a file system type (`ext4`)

Associated **operations** ("bottom" VFS interface):

- `super_operations`, `inode_operations`, `dentry_operations`, `file_operations`

dentry (or directory entry)



dentry

Associated with a file or a directory to:

- Store the file/directory name
- Store its location in the directory
- Perform directory specific operations, e.g., pathname lookup

`/home/lkp/test.txt`

- One dentry associated with each of: ‘/’, ‘home’, ‘lkp’, and ‘test.txt’

Constructed on the fly as files and directories are accessed

- Cache of disk representation

dentry

A dentry can be used, unused or negative

Used: corresponds to a valid inode (pointed by `d_inode`) with one or more users (`d_count`)

- Cannot be discarded to free memory

Unused: valid inode, but no current users

- Kept in RAM for caching; can be discarded

Negative: does not point to a valid inode

- e.g., `open()` on a file that does not exists
- Kept around for caching; can be discarded

dentry

```
struct dentry {
    atomic_t            d_count;    /* usage count */
    unsigned int        d_flags;    /* dentry flags */
    spinlock_t          d_lock;     /* per-dentry lock */
    int                 d_mounted; /* indicate if it is a mount point */
    struct inode        *d_inode;  /** associated inode **/
    struct hlist_node d_hash;      /** list of hash table entries **/
    struct dentry       *d_parent; /** parent dentry **/
    struct qstr         d_name;     /* dentry name */
    struct list_head    d_lru;      /* unused list */
    struct list_head            d_subdirs;      /** sub-directories **/
    struct list_head            d_alias;        /** list of dentries
                                                  ** pointing to the same inode **/
    unsigned long               d_time;         /* last time validity was checked */
    struct dentry_operations *d_op;             /** operations **/
    struct super_block          *d_sb;          /** superblock **/
    void                        *d_fsdata;      /* filesystem private data */
    unsigned char               d_iname[DNAME_INLINE_LEN_MIN]; /* short name */
    /* ... */
};
```

ER SCIENCE

dentry cache

Linked list of used dentries linked by the `i_dentry` field of their inode

- One inode can have multiple links, thus multiple dentries

Linked list of LRU sorted unused and negative dentries

- LRU: quick reclamation from the tail of the list

Hash table + hash function to quickly resolve a path into the corresponding dentry present in the dcache

dentry operations

```
/* include/linux/dcache.h */
struct dentry_operations {
    int (*d_revalidate)(struct dentry *, unsigned int);
    int (*d_weak_revalidate)(struct dentry *, unsigned int);
    int (*d_hash)(const struct dentry *, struct qstr *);
    int (*d_compare)(const struct dentry *,
            unsigned int, const char *, const struct qstr *);
    int (*d_delete)(const struct dentry *);
    int (*d_init)(struct dentry *);
    void (*d_release)(struct dentry *);
    void (*d_prune)(struct dentry *);
    void (*d_iput)(struct dentry *, struct inode *);
    char *(*d_dname)(struct dentry *, char *, int);
    struct vfsmount *(*d_automount)(struct path *);
    int (*d_manage)(const struct path *, bool);
    struct dentry *(*d_real)(struct dentry *, const struct inode *,
                unsigned int);
} ____cacheline_aligned;
```

dentry operations

```
int d_hash(struct dentry *dentry, struct qstr *name)
```

- Create a hash value for a dentry to insert in the dcache

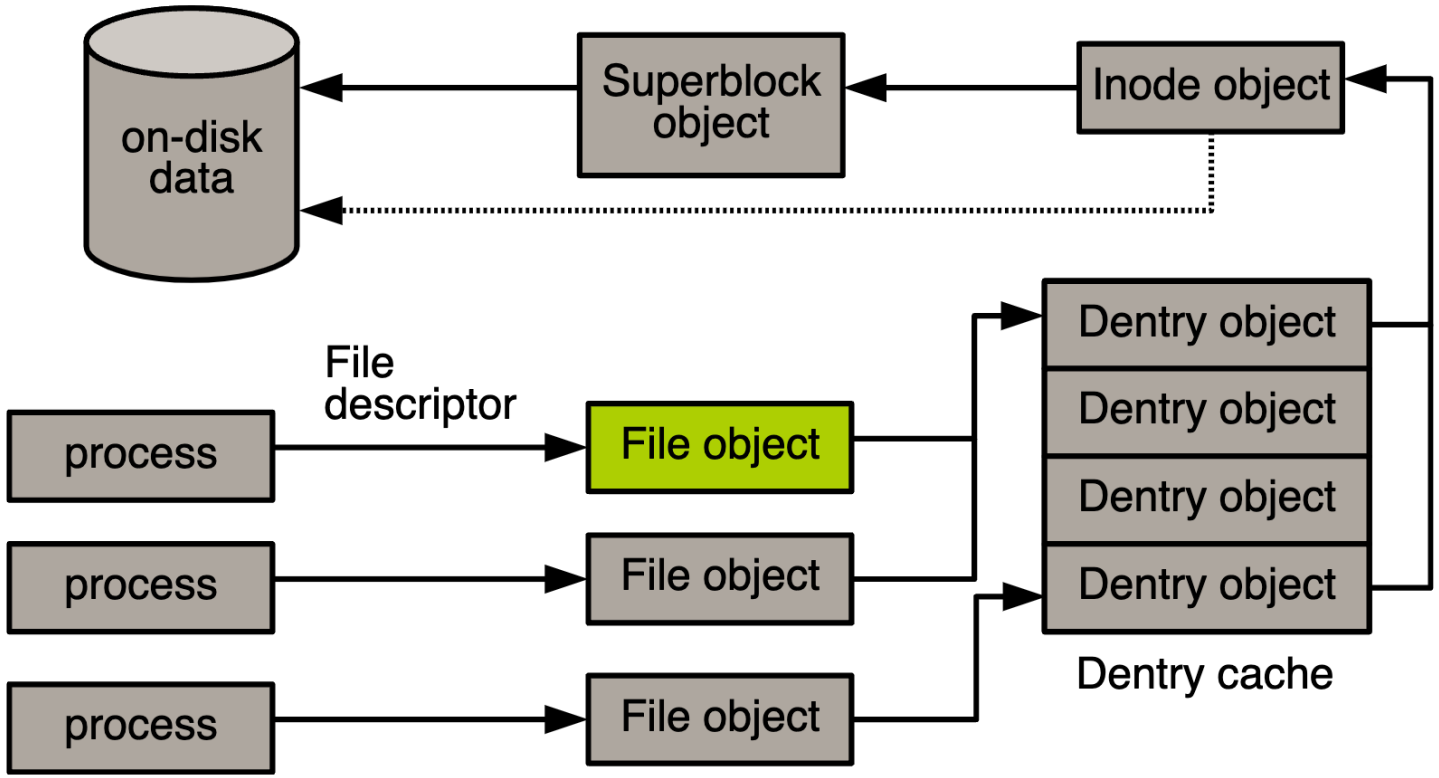
```
int d_compare(struct dentry *dentry, struct qstr *name1,
struct qstr *name2)
```

- Compare two filenames, requires dcache_lock

```
int d_delete (struct dentry *dentry)
```

- Called by VFS when d_count reaches zero, requires dcache_lock and d_lock

File object


on-disk data
Superblock object
Inode object
File descriptor
process
process
process
File object
File object
File object
Dentry object
Dentry object
Dentry object
Dentry object
Dentry cache

File object

The file object

- Represents a file opened by a process
- Created on `open()` and destroyed on `close()`

2 processes opening the same file:

- Two file objects, pointing to the same unique `dentry`, that points itself on a unique `inode`

No corresponding on-disk data structure

File object

```
/* include/linux/fs.h */
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 */
    loff_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
                                             ** == inode->i_mapping **/

    /* ... */
};
```

File operations

```c
/* include/linux/fs.h */
struct file_operations {
    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 *);
    /* ... */
};
```

File operations

```
loff_t llseek(struct file *file, loff_t offset, int origin)
```

- Update file offset

```
ssize_t read(struct file *file, char *buf, size_t count,
loff_t *offset)
```

- Read operation

```
ssize_t aio_read(struct kiocb *iocb, char *buf, size_t count,
loff_t offset)
```

- Asynchronous read

File operations

```
ssize_t write(struct file *file, const char *buf, size_t count,
loff_t *offset)
```

- Write operation

```
ssize_t aio_write(struct kiocb *iocb, const char *buf, size_t
count, loff_t offset)
```

- Asynchronous write

```
int readdir(struct file *file, void *dirent, filldir_t filldir)
```

- Read the next directory in a directory listing

File operations

```
int ioctl(struct inode *inode, struct file *file, unsigned
int cmd, unsigned long arg)
```

- Sends a command and arguments to a device
- Unlocked/compat versions

```
int mmap(struct file *file, struct vm_area_struct *vma)
```

- Maps a file into an address space

```
int open(struct inode *inode, struct file *file)
```

- Opens a file

Trace sys_read system call in VFS

fs/read_write.c

```
621 SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
622 {
623         return ksys_read(fd, buf, count);
624 }
```

```
602 ssize_t ksys_read(unsigned int fd, char __user *buf, size_t count)
603 {
604         struct fd f = fdget_pos(fd);
605         ssize_t ret = -EBADF;
606
607         if (f.file) {
608                 loff_t pos, *ppos = file_ppos(f.file);
609                 if (ppos) {
610                         pos = *ppos;
611                         ppos = &pos;
612                 }
613                 ret = vfs_read(f.file, buf, count, ppos);
```

```
450 ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
```

```
467         if (file->f_op->read)
468                 ret = file->f_op->read(file, buf, count, pos);
469         else if (file->f_op->read_iter)
470                 ret = new_sync_read(file, buf, count, pos);
```

Summary

Key data structures

- `struct file_system_type`: file system (e.g., ext4)
- `struct super_block`: mounted file system instance (i.e., partition)
- `struct dentry`: path name
- `struct inode`: file metadata
- `struct file`: open file descriptor

Further readings

LKD3: Chapter 13 The Virtual Filesystem

Performance and protection in the ZoFS user-space NVM file system, SOSP'19

CrossFS: A Cross-layered Direct-Access File System, OSDI'20

Page cache and Page fault

Xiaoguang Wang

UIC COMPUTER SCIENCE

Agenda

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
```

Link: Latency numbers every programmer should know

Page cache (or buffer cache)

The Linux kernel implements a disk cache called the **page cache**.

- Minimize disk I/O overhead by storing disk data in physical memory

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

Buffered I/O 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

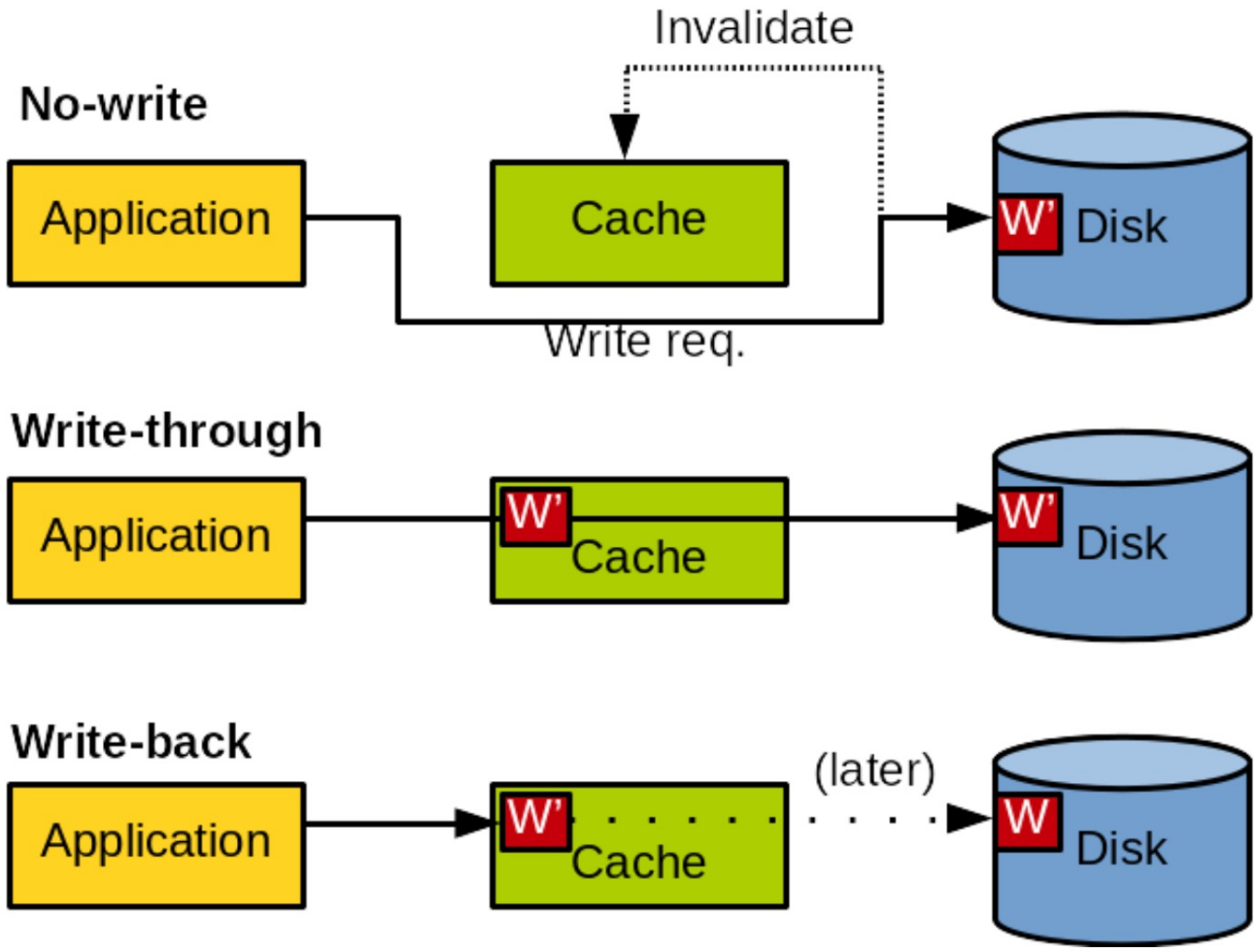
Write-through: write operations immediately go through to disk

- 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



Cache eviction

Smaller memory than disk ➔ Cache eviction

When data should be removed from the cache?

- Need more free memory (memory pressure)

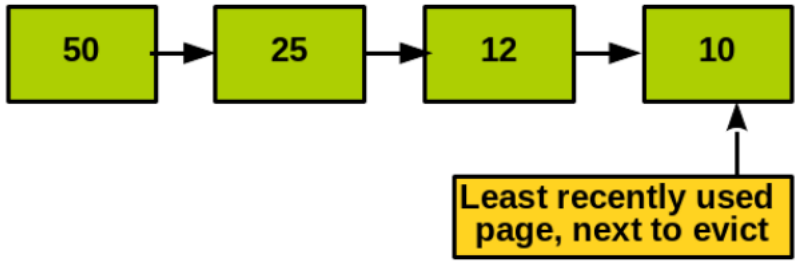
Which data should be removed from the cache?

- 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

← Q: How to solve it?

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

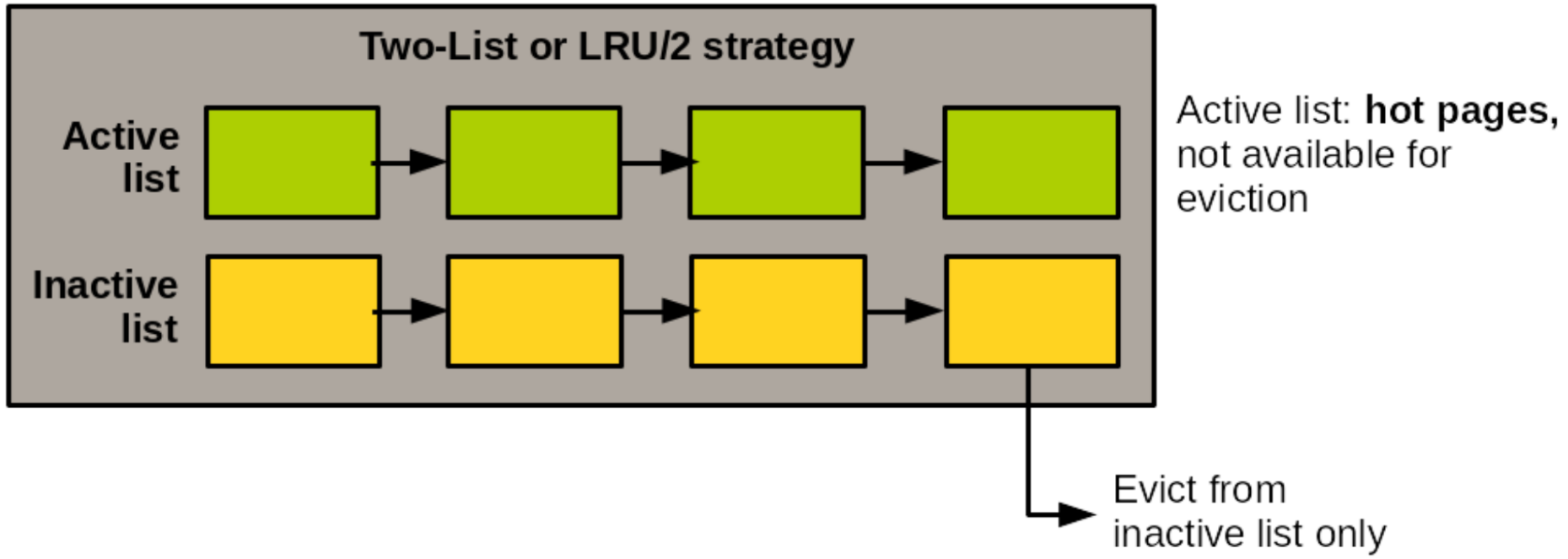
Newly accessed pages are added to *inactive list*

If a page in an inactive list is accessed again, it is promoted to an active list

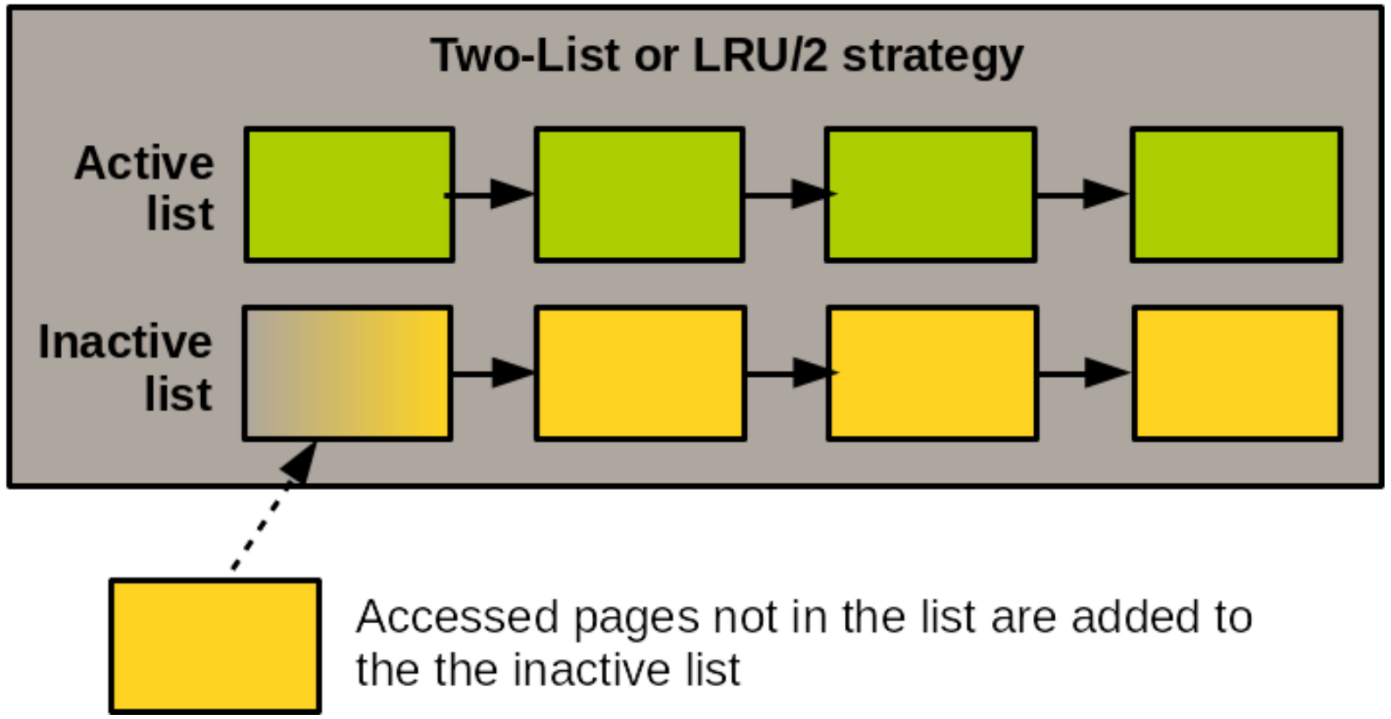
- When a page is added to an inactive list, its access permission in a page table is *disabled to track its access*.

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.

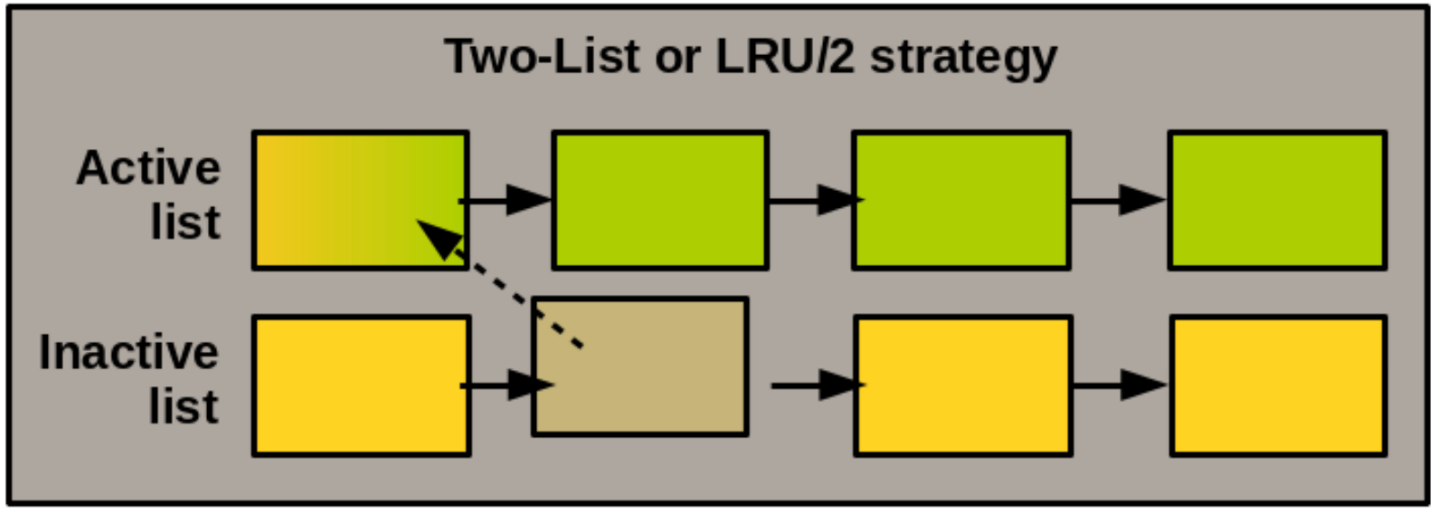
The two-list strategy



The two-list strategy

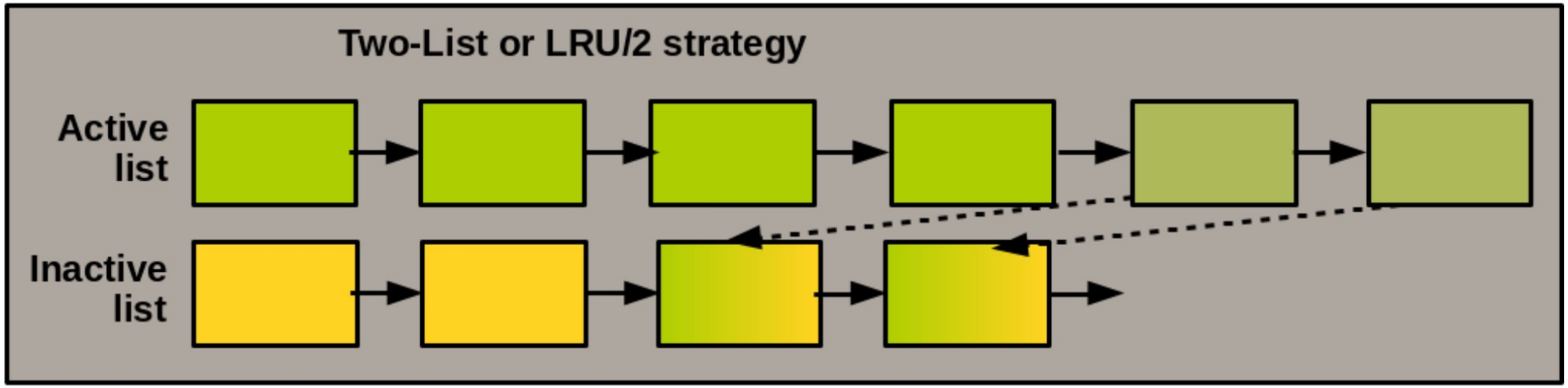


The two-list strategy



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

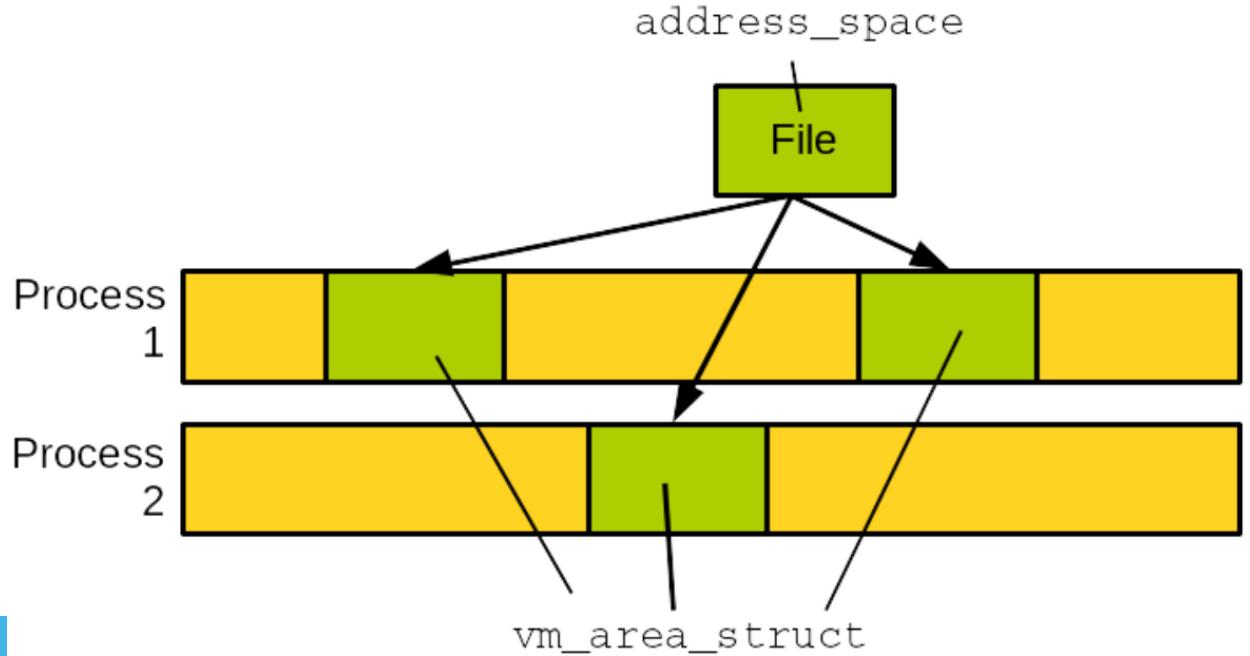
```
/* include/linux/fs.h */
struct inode {
    const struct inode_operations    *i_op;
    struct super_block               *i_sb;
    struct address_space             *i_mapping;
    unsigned long                    i_ino;
};

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 */
};
```

address_space

An entity presents the page cache of **a file**

- an `address_space` = a file = accessing a page cache of a file
- an `address_space` = one or more `vm_area_struct`



The Linux page cache

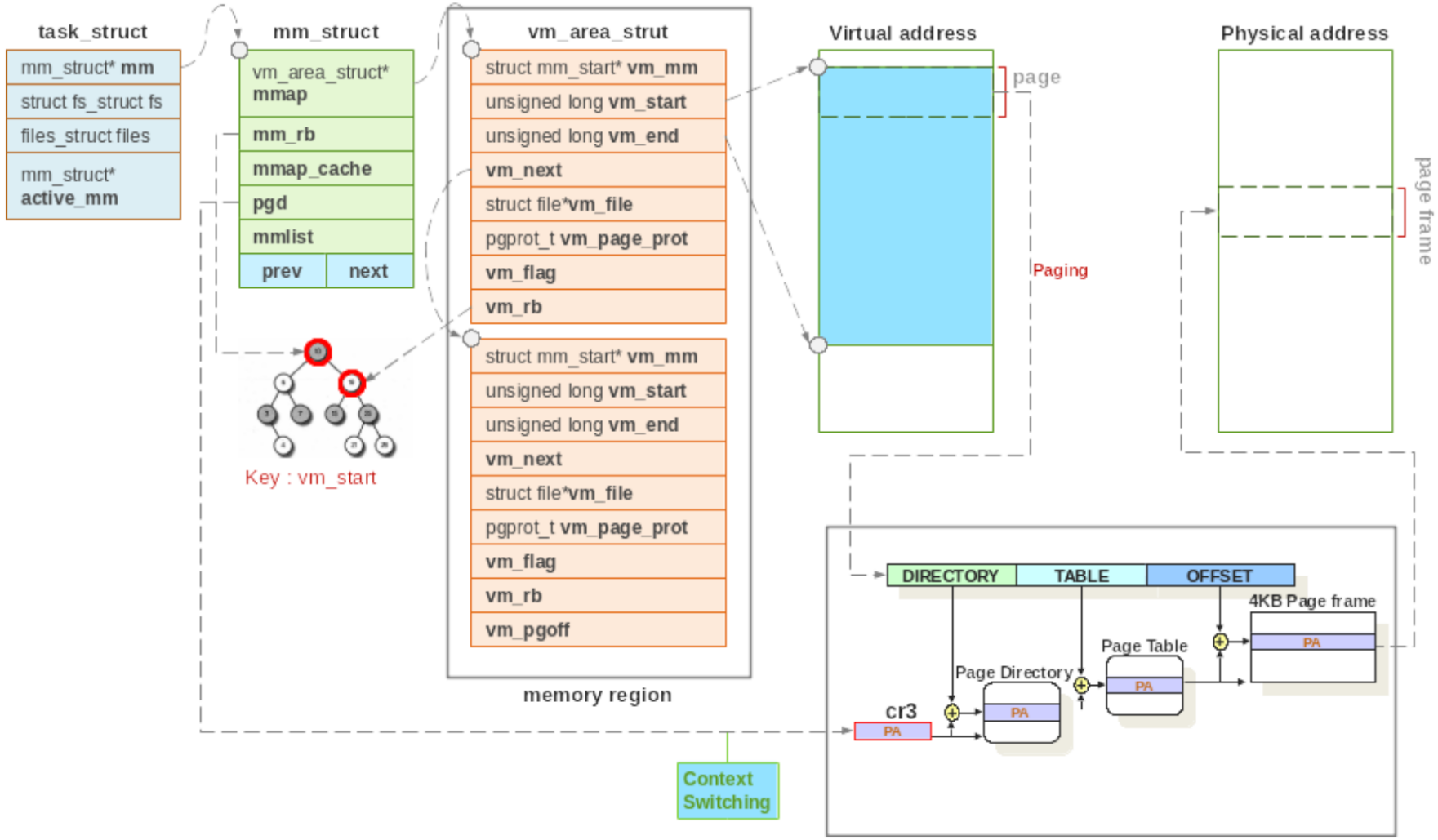
```
$> sudo cat /proc/1/maps
7fe87b1f1000-7fe87b21d000 r-xp 00000000 fd:00 1975147  /usr/lib64/libseccomp.so
7fe87b21d000-7fe87b41c000 ---p 0002c000 fd:00 1975147  /usr/lib64/libseccomp.so
7fe87b41c000-7fe87b431000 r--p 0002b000 fd:00 1975147  /usr/lib64/libseccomp.so
7fe87b431000-7fe87b432000 rw-p 00040000 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`

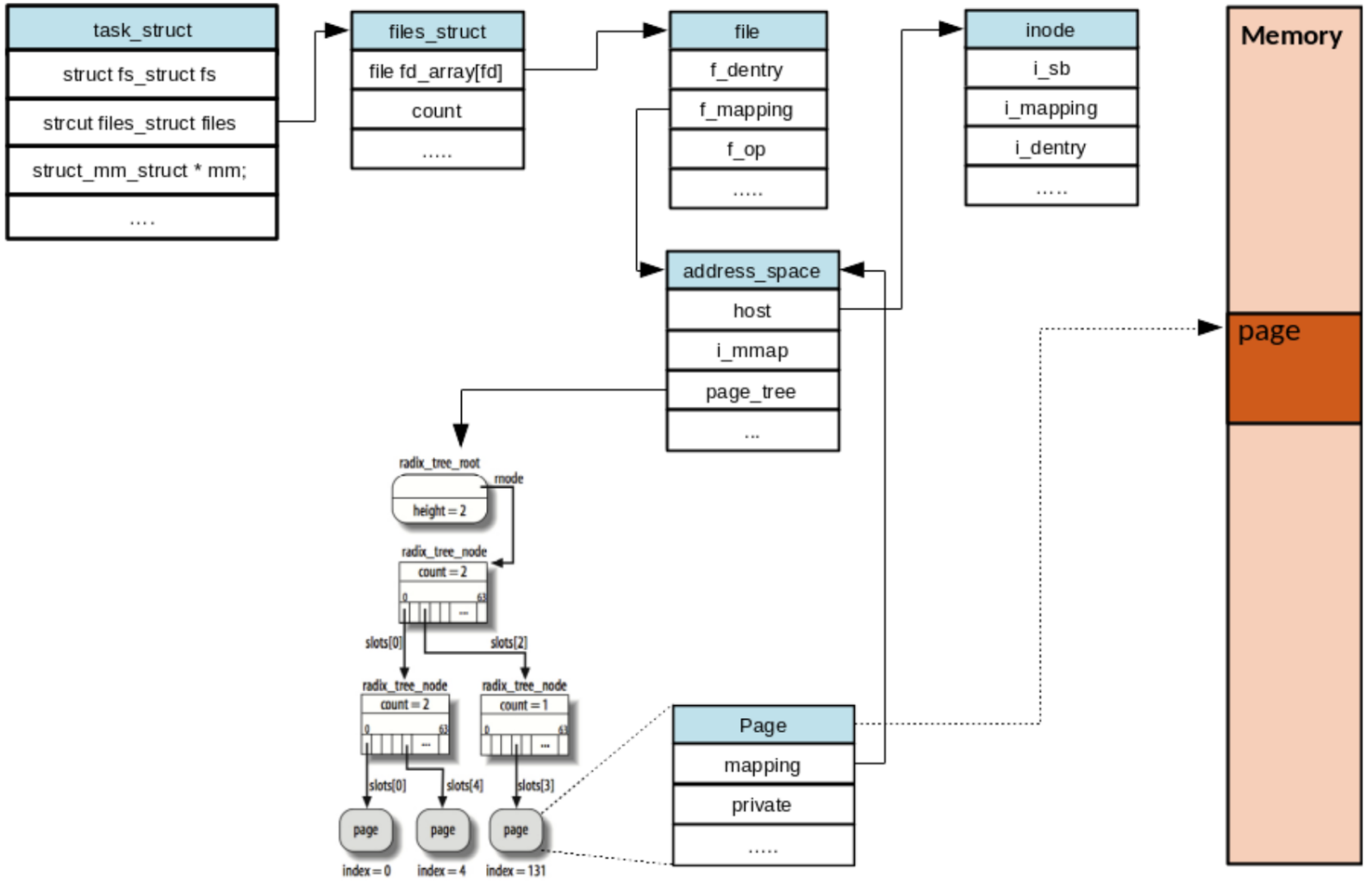
Q: the number of `address_space`

vm_area_struct - page table



PUTER SCIENCE

Page cache - physical page



COMPUTER SCIENCE

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

`filemap_fault`

- PTE entry does not exist (`---`)
- BUT VMA is marked as accessible (e.g., `rwx`) and has an associated file (`vm_file`)

Page fault handler notices differences

- In `filemap_fault`, look up a page cache of the file
- If cache hit, map the page in the cache
- Otherwise, `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 calls `wakeup_flusher_threads()`

- Wakes up one or several flusher threads performing writeback though `bdi_writeback_all`

Thread write data to disk until

- `num_pages_to_write` have been written
- and the amount of memory drops below the threshold

Percentage of total dirty memory pages to trigger flusher daemon

- `/proc/sys/vm/dirty_background_ratio`

What happens in the kernel?

```
00 int main(int argc, char *argv[])
01 {
02         char buff[8192];
03         char *addr;
04         int fd;
05         int i;
06
07         fd = open ("test-file.dat", O_CREAT | O_RDWR | O_TRUNC);
08         for (i = 0; i < 10; ++i)
09                 write(fd, buff, sizeof(buff));
10         addr = mmap(NULL, sizeof(buff), PROT_READ | PROT_WRITE,
                       MAP_PRIVATE, fd, 0);
11         memcpy(buff, addr, sizeof(buff));
12         memset(addr, 1, sizeof(buff));
13         munmap(addr, sizeof(buff));
14         close(fd);
15
16         return 0;
17 }
```

Further readings

LKD3: Chapter 16: *The Page Cache and Page Writeback*

Better active/inactive list balancing

MGLRU

LWN: User-space page fault handling

Midterm

Bring your **laptop**

- It will be on blackboard

45 minutes

- multiple choice quizzes
- short answers

Open-book but no discussion or posting questions via online services (ChatGPT, Stack Overflow, Chegg, Facebook, Twitter, etc).

Midterm scopes

Kernel code exploration, kernel debugging, etc.

Isolation, system calls and Linux kernel data structures

Process management and process scheduling

Interrupt handling: top half, bottom half

Kernel synchronization

Timer and time management

Virtualization

Memory, address space

VFS, ~~Filesystem and block I/O~~