File management
Virtual file system: interface, data structures

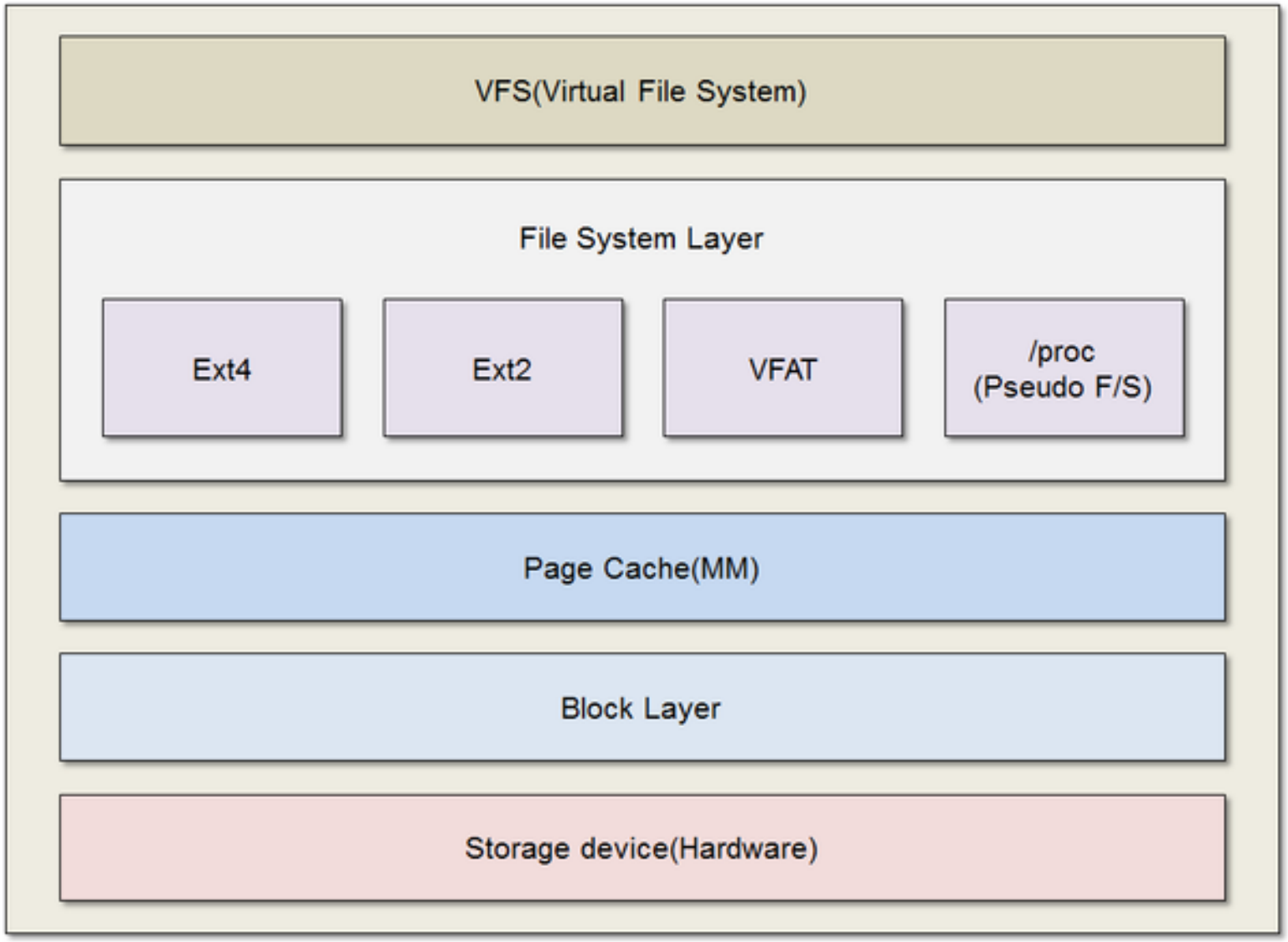
Table of contents

- Virtual File System (VFS)
- File system interface
  - Creating and opening a file
  - Reading from a file
  - Writing to a file
  - Closing a file
- VFS data structures
  - Process data structures with file information
  - Structure fs_struct
  - Structure files_struct
  - Structure file
  - Inode
  - Superblock
- Pipe vs FIFO

Motto

Everything is a file

Virtual File System (VFS)



VFS (source: Tizen Wiki)

Maintenance of filesystems

Trust in and maintenance of filesystems, Jonathan Corbet, November 2023

The Linux kernel supports a wide variety of filesystems, many of which are no longer in heavy use — or, perhaps, any use at all. The kernel code implementing the less-popular filesystems tends to be relatively unpopular as well, receiving little in the way of maintenance. Keeping old filesystems alive does place a burden on kernel developers, so it is not surprising that there is pressure to remove the least popular ones.

At the **2023 Kernel Maintainers Summit**, the developers talked about these filesystems and what can be done about them.

User space (in the form of desktop environments in particular) has a strong urge to automatically mount filesystems, even those that are unmaintained, insecure, and untrustworthy. This automounting exposes the system to security threats and is always a bad idea; maybe there needs to be a way for the kernel to indicate to user space that some filesystems are not suitable for mounting in this way.

That, **Kroah-Hartman** said, requires coming up with a list of good and bad filesystems. **Hellwig** said that there would need to be at least three levels: "no trust", "generally maintained but don't mount untrusted images", and "well maintained". **Torvalds** said that this information could be given to the kernel when a filesystem is registered, and a warning printed if an untrusted filesystem is mounted.

Torvalds repeated that it is possible to deprecate old code when there is a good reason to do so.

Virtual File System (VFS)

Linux can support many different (**formats** of) **file systems**. (*How many?*)

The virtual file system uses a **unified interface** when accessing data in various formats, so that from the user level adding support for the new data format is relatively simple.

This concept allows implementing support for data formats:

- saved on **physical media** (**Ext2**, **Ext3**, **Ext4** from Linux, **VFAT**, **NTFS** from Microsoft),
- available via **network** (like **NFS**),
- **dynamically created** at the user's request (like **/proc**).

This unified interface is fully compatible with the **Unix-specific file model**, making it easier to implement **native Linux file systems**.

File System Interface

In Linux, processes refer to files using a well-defined set of **system functions**:

- functions that support existing files: **open()**, **read()**, **write()**, **lseek()** and **close()**,
- functions for creating new files: **creat()**,
- functions used when implementing pipes: **pipe()** i **dup()**.

The first step that the process must take to access the data of an existing file is to call the **open()** function.

If successful, it passes the **file descriptor** to the process, which it can use to perform other operations on the file, such as reading (**read()**) and writing (**write()**).

The **read()** and **write()** functions provide **sequential** access to data.

When **direct** (non-sequential) access is needed, the process can use the **lseek()**, function, which allows changing the **current position** in the file.

When a process stops using a file or a pipe, it can close it by calling the **close()** function with the appropriate descriptor to free the position in the descriptor table.

File System Interface

Related processes (such as ancestor and descendant) can communicate with each other using **communication channels** (***pipes***).

They are created using the **pipe()** function, which passes **two** file descriptors: for **reading** and for **writing**.

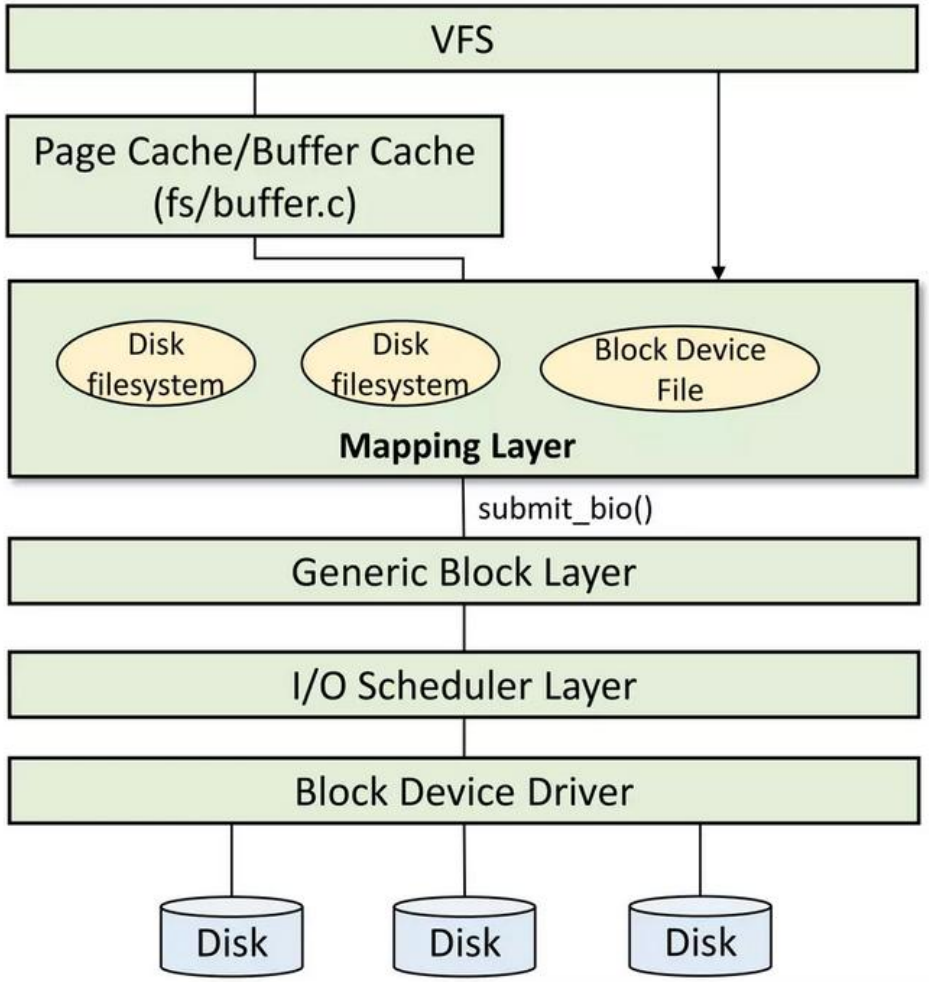
The child process (created using the **fork()** function) **inherits** these descriptors from the parent process, so data can flow from one process to another.

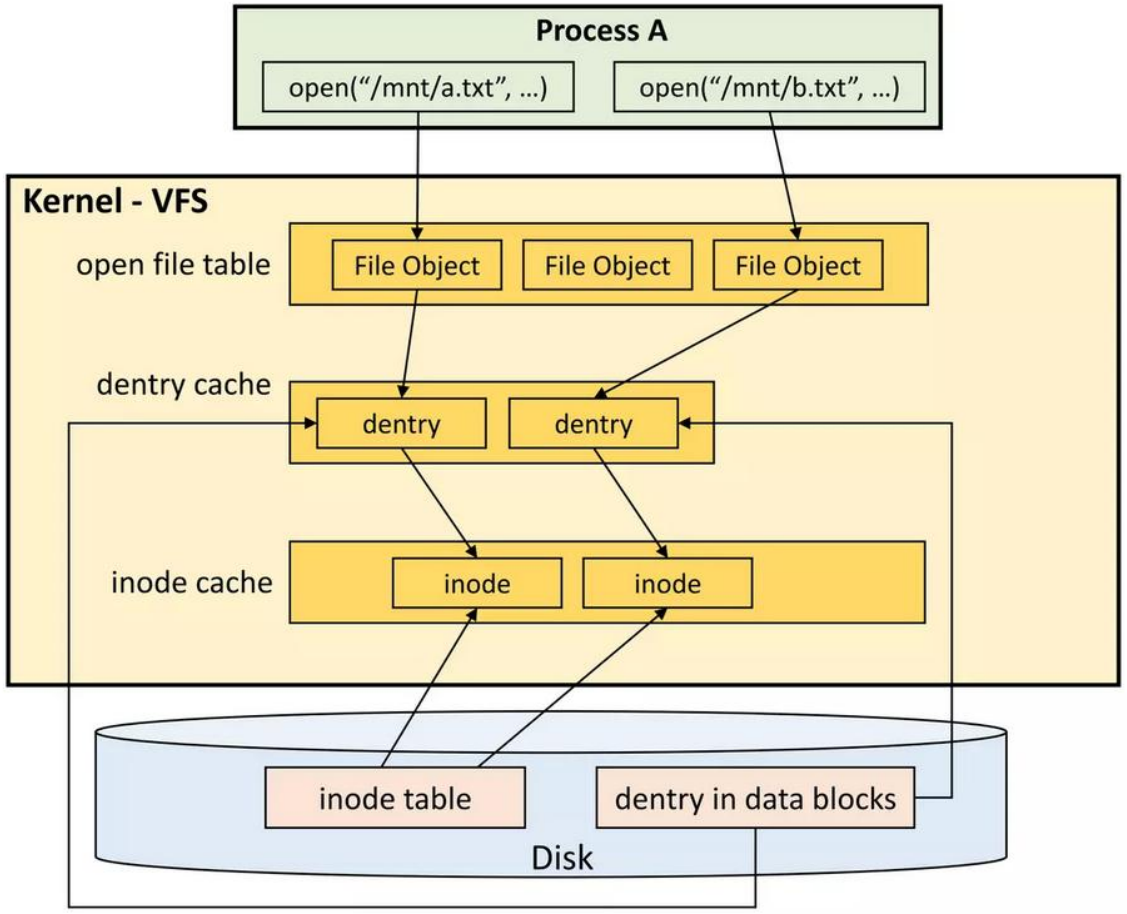
The **pipe()** function, together with the **dup()** function, which copies a given descriptor to the first free position in the descriptor table, enables implementation of pipelines, such as in the command interpreter, that connect the **standard output** of one process to the **standard input** of another.

These system functions have their equivalents in the **library of standard input/output functions**, which create a **high-level interface** between the process and the kernel, enabling the use of facilities such as **buffering** or **formatted output**.

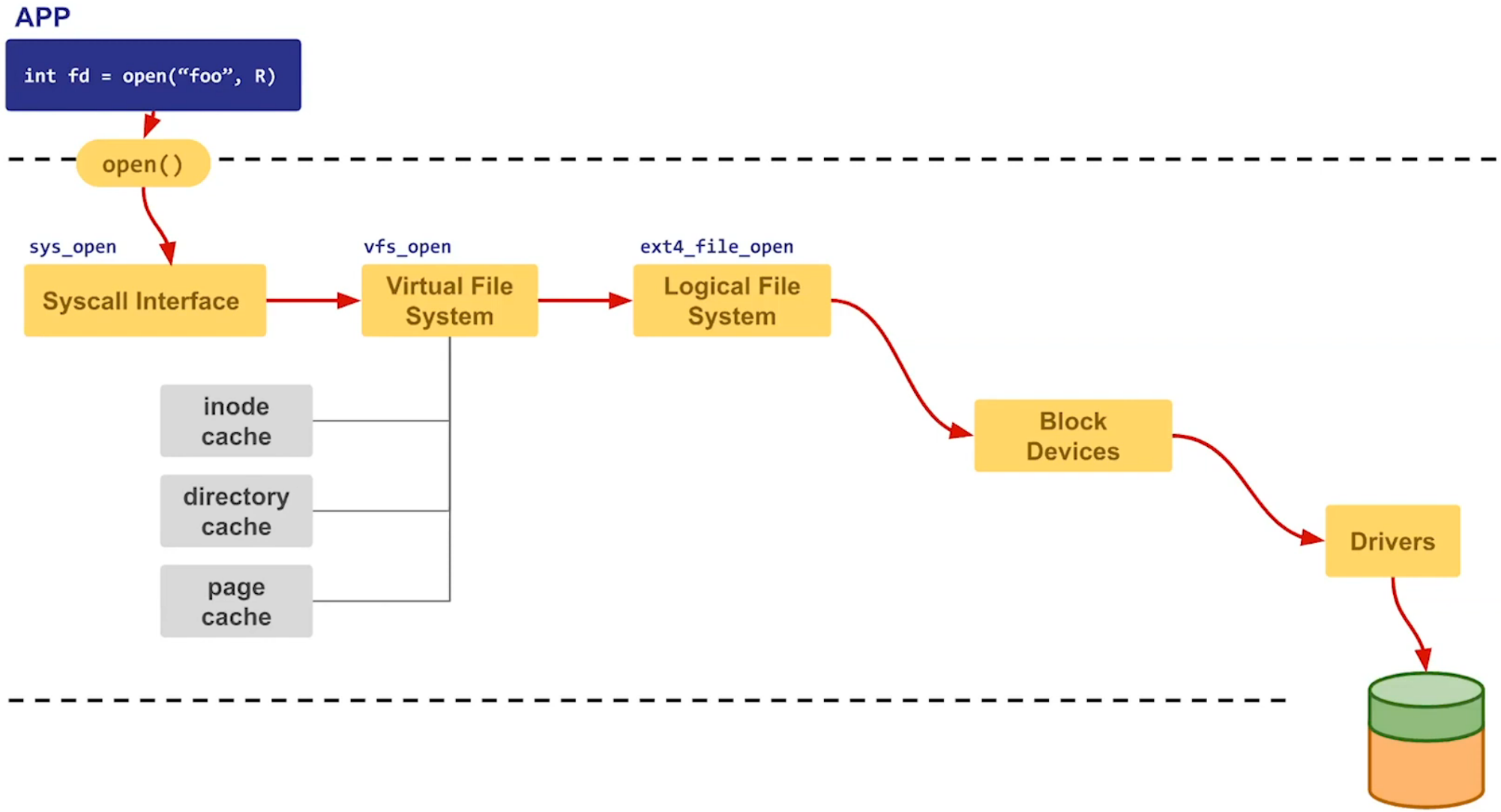
Each **filesystem driver** exports a table of **supported operations** to VFS and when a system call is issued, VFS performs some preliminary actions, finds a file system specific function in that table and calls it.

Kernel components affected by an IO request
Object relationship in VFS





(source: Adrian Huang, Virtual File System, 2022)



SDC2020: Tracing and Visualizing File System Internals with eBPF Superpowers

Suchakrapani Sharma, Staff Scientist, ShiftLeft Inc Hani Nemati, Software Engineer, Microsoft

SKIP

Creating and opening a file

The **creat()** function creates a file with a given name. It is only maintained for compatibility with previous versions of Linux.

```
SYSCALL_DEFINE2 creat(creat, const char __user *pathname, umode_t mode)
{
    int flags = O_CREAT | O_WRONLY | O_TRUNC;
    return do_sys_open(pathname, flags, mode);
}
```

The **open()** system function opens the file. The result is a natural number called the **file descriptor**. It is used in the process as a **file identifier**; two files opened by the process have **different descriptors**. One file can be opened **twice** by a process and then has **two different descriptors**.

```
asmlinkage long sys_open(const char __user *filename, int flags, umode_t mode);
```

The macro definition from the **/fs/open.c** defines **sys_open()** as **do_sys_open()**.

```
SYSCALL_DEFINE3(open,  const char __user *, filename, int, flags, umode_t, mode)
{
    return do_sys_open(filename, flags, mode);
}
```

Creating and opening a file

The file descriptor is an index in the **file descriptor table** (also called the **open file descriptor table**).

The first step is to load the **pathname of the file** from the process address space into the kernel space.

Then free space in the **file descriptor table** is located. The descriptor with the **smallest free number** is selected, it will be released if the file opening operation fails.

The **pathname** of the file is examined step by step to reach the corresponding **inode** of the file to be opened.

If a file is to be **created**, the kernel checks if the file already exists and sends an error if necessary. The error also appears if the calling process does not have **access rights**.

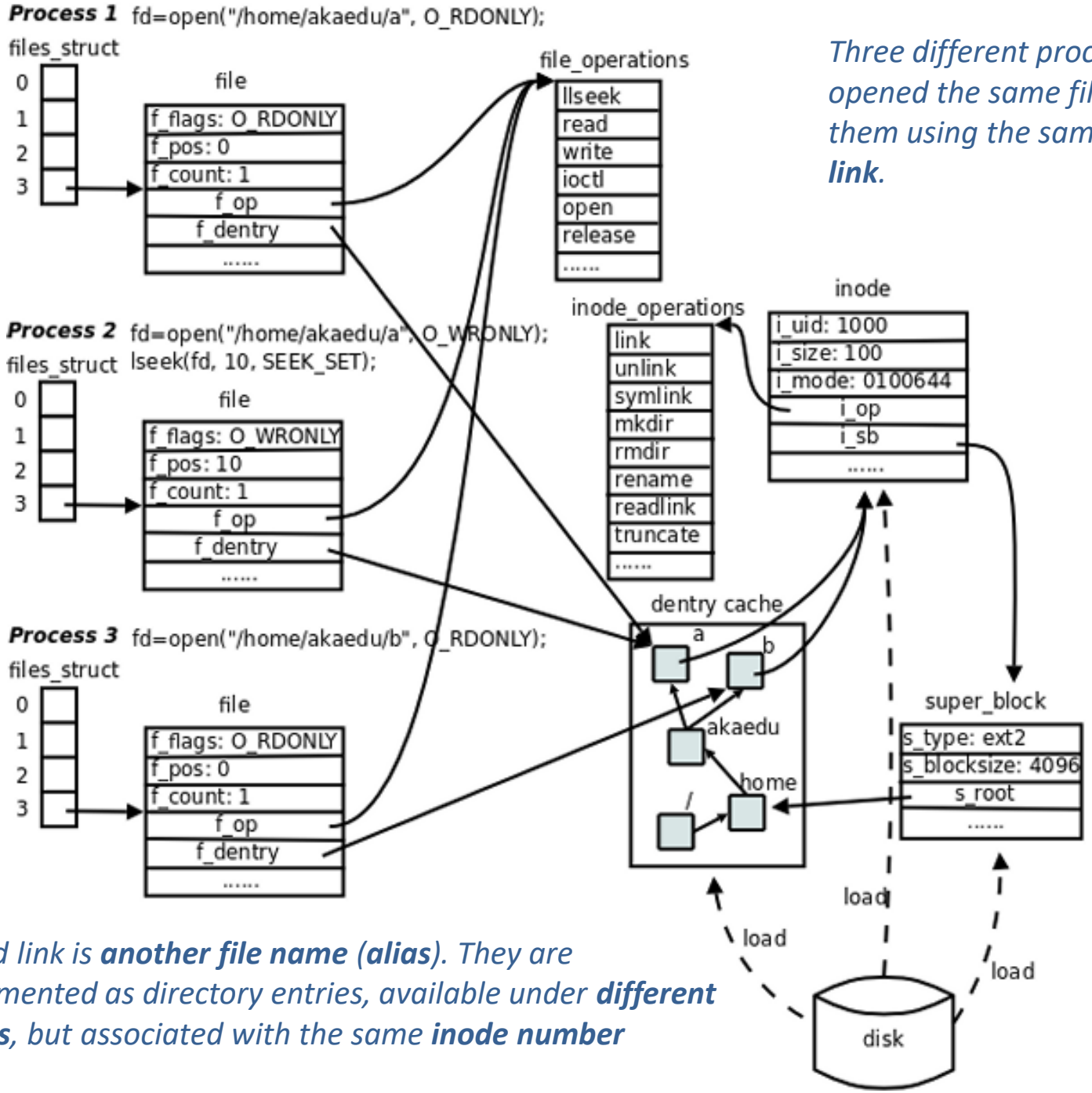
If the rights are correct, a **new free inode** is taken.

When finding a new inode, a lock is placed on the **inode of the home directory** so that a new inode can be attached to it.

A position is created and filled in the **file table** (also called the **open file table**) associated with the given file system. From there, the **inode** of the file can be reached. It also sets the **current position** in the file.

Now the part of the file opening function is invoked, which depends on the **particular file system**.

*Each of the processes reaches through the private **file descriptor** to own **file** structure. Two **dentry** structures are used, one for each hard link. Both **dentry** structures point to the same **inode** that identifies the same **superblock** and with it the same **physical file** on the disk.*



*Three different processes opened the same file, two of them using the same **hard link**.*

*A hard link is **another file name** (**alias**). They are implemented as directory entries, available under **different names**, but associated with the same **inode number***

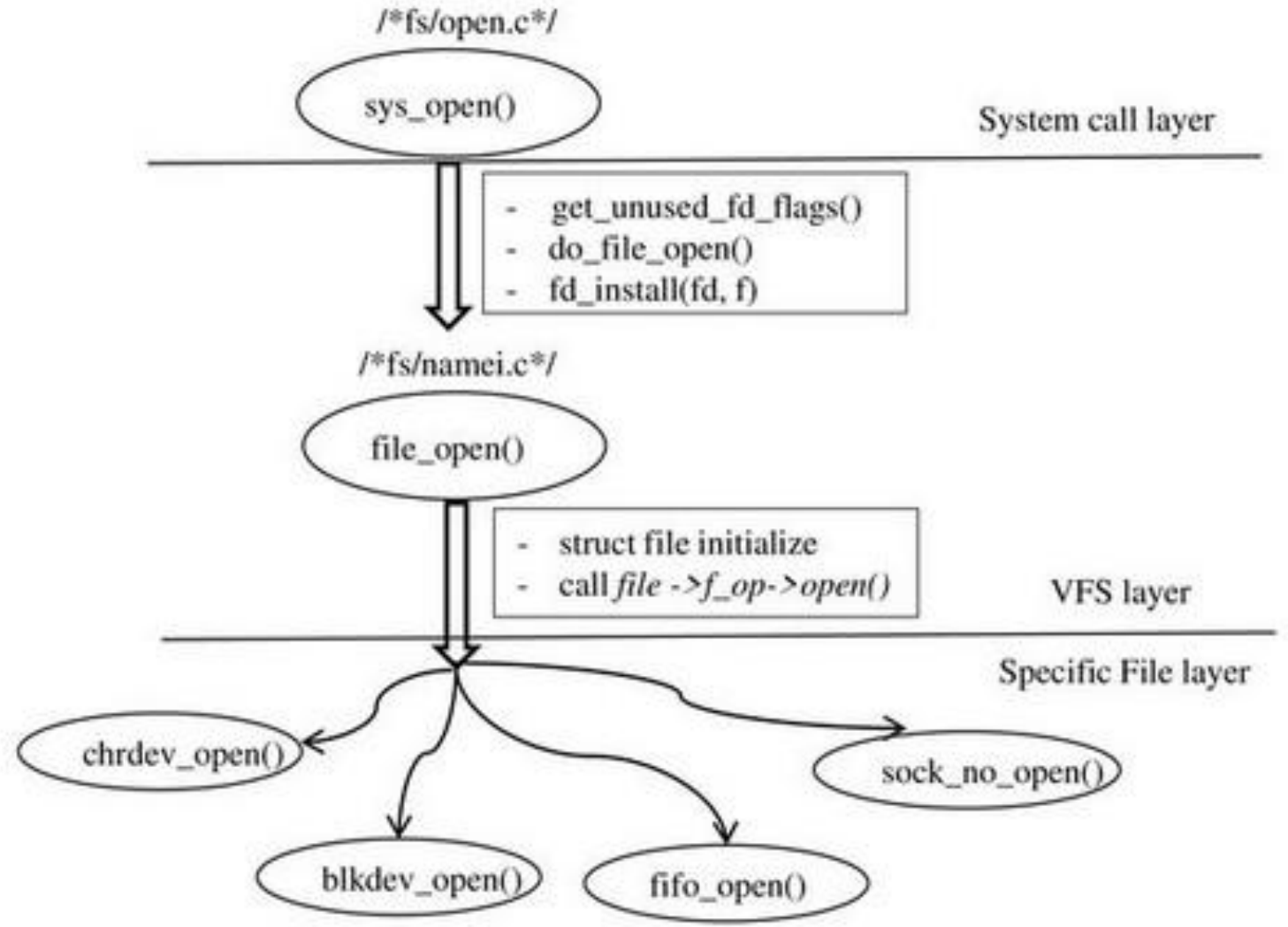
VFS – operacje (źródło: Y. Zhang)

SKIP

```
const struct file_operations ext2_file_operations = {
        .llseek         = generic_file_llseek,
        .read_iter      = ext2_file_read_iter,
        .write_iter     = ext2_file_write_iter,
        .unlocked_ioctl = ext2_ioctl,
#ifdef CONFIG_COMPAT
        .compat_ioctl   = ext2_compat_ioctl,
#endif
        .mmap           = ext2_file_mmap,
        .open           = ext2_file_open,
        .release        = ext2_release_file,
        .fsync          = ext2_fsync,
        .get_unmapped_area = thp_get_unmapped_area,
        .splice_read    = filemap_splice_read,
        .splice_write   = iter_file_splice_write,
};
```

```
const struct file_operations ext4_file_operations = {
        .llseek         = ext4_llseek,
        .read_iter      = ext4_file_read_iter,
        .write_iter     = ext4_file_write_iter,
        .iopoll         = iocb_bio_iopoll,
        .unlocked_ioctl = ext4_ioctl,
#ifdef CONFIG_COMPAT
        .compat_ioctl   = ext4_compat_ioctl,
#endif
        .mmap           = ext4_file_mmap,
        .open           = ext4_file_open,
        .release        = ext4_release_file,
        .fsync          = ext4_sync_file,
        .get_unmapped_area = thp_get_unmapped_area,
        .splice_read    = ext4_file_splice_read,
        .splice_write   = iter_file_splice_write,
        .fallocate      = ext4_fallocate,
        .fop_flags      = FOP_MMAP_SYNC | FOP_BUFFER_RASYNC |
                          FOP_DIO_PARALLEL_WRITE,
};
```

Creating and opening a file



VFS – operations (source: Kyu Ho Park)

SKIP

Reading from a file

The **read()** system function is used to read data from an open file.

```
asmlinkage long sys_read(unsigned int fd, char __user *buf, size_t count);
```

Call chain (*shortened*):

- The macro definition from the **/fs/read_write.c** file defines **sys_read()** as **vfs_read()**.

```
SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
```

- Inside **vfs_read** there is a file system specific function call:

```
if (file->f_op->read)
    return file->f_op->read(file, buf, count, pos);
else if (file->f_op->read_iter) /* three other calls along the way, that's
                                   why change of parameters */
    return file->f_op->read_iter(&kiocb, &iter);
```

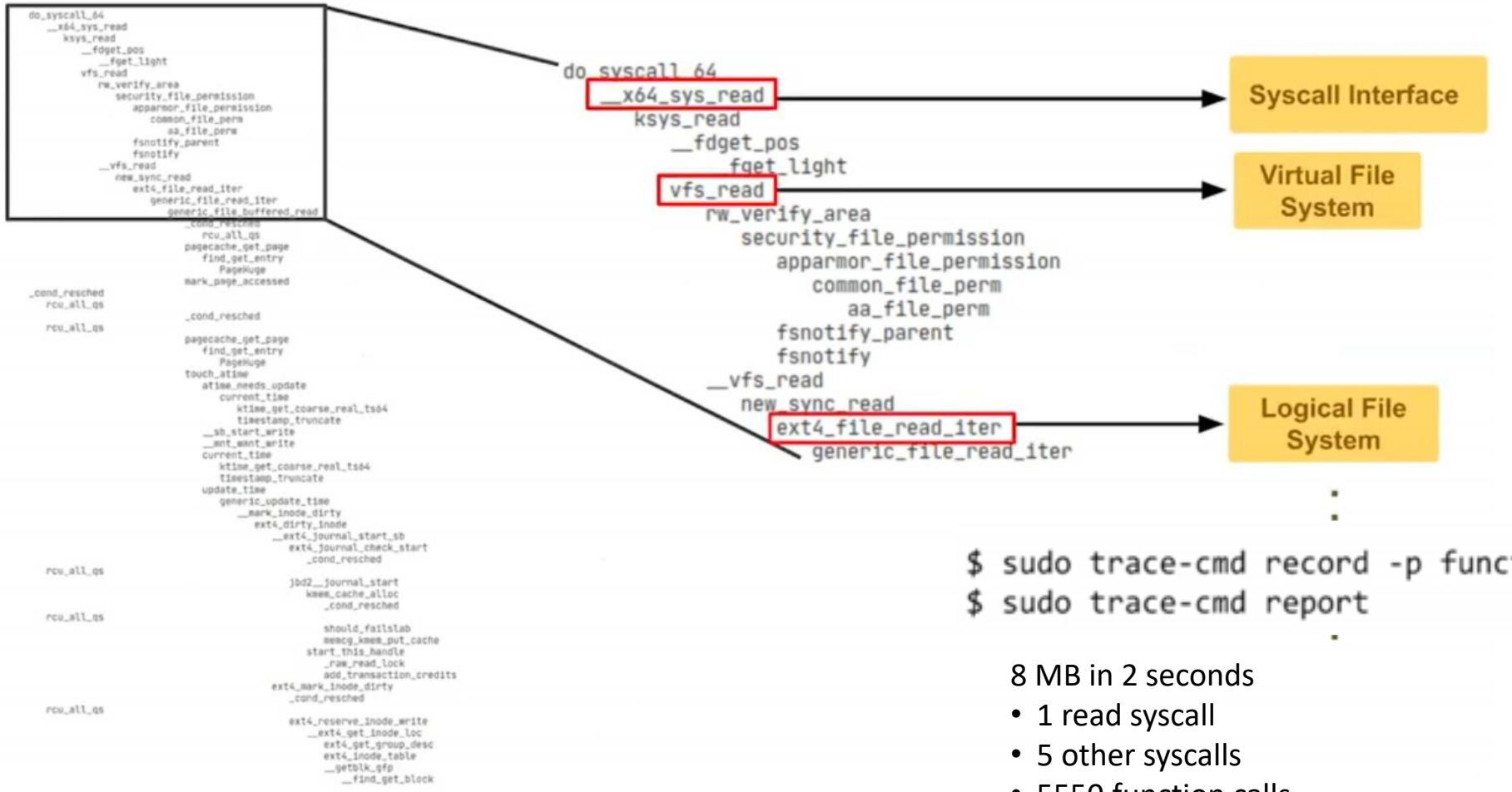
- For **Ext*** file systems, the **read()** function is not defined, and the **read_iter()** function is defined as **ext2_file_read_iter()** (**ext4_file_read_iter()**)
- Both functions call a function from the VFS level, **generic_file_read_iter()**, which does most of the work.

Reading from a file

The kernel in the **loop** performs the following actions until the **reading** is **complete**.

- Searches in the **page cache** for the **page** that contains data from the file. It may happen that the kernel does not find such a page in the main memory – even if the file was **read-ahead**, the corresponding page may have been removed from memory. The kernel reserves the **free page frame** in memory and instructs to fill the frame with data from the corresponding block on the disk.
- If the frame is filled with current data, or if its content is currently being updated, the kernel initiates loading the appropriate number of pages **in advance**. It waits until all previously initiated I/O operations on the given frame have been completed and will be blocked.
- If the unlocked frame contains outdated data, then the kernel synchronously fills it with data from the corresponding block on the disk.
- **Copies** data from the appropriate frame fragment to the **process address space**.

Tracing a read() with ftrace




```
$ sudo trace-cmd record -p function -P 2535
$ sudo trace-cmd report
```

8 MB in 2 seconds

- 1 read syscall
- 5 other syscalls
- 5550 function calls

SDC2020: Tracing and Visualizing File System Internals with eBPF Superpowers

Suchakrapani Sharma, Staff Scientist, ShiftLeft Inc Hani Nemati, Software Engineer, Microsoft

SKIP

Writing to a file

The **write()** system call is used to write data to an open file.

```
asmlinkage long sys_write(unsigned int fd, const char __user *buf, size_t count)
```

The file system **independent part** (**vfs_write()**) checks if the process has the **permission to write** to the file, or whether the process has the **right to access the memory area**, to which the address was passed as call parameter.

Now the function appropriate to the specific file system is being called; in case of **Ext*** it is **generic_file_write_iter()**.

The **inode is locked** to exclude **simultaneous writes** to the file.

Writing to a file, like reading, is done through the **page cache**. When writing full pages, the task is simpler. However, if only a part of the page is updated, first the entire page has to be loaded into the page cache, modified, and then marked for writing.

SKIP

Closing a file

The **close()** system function is used to close an open file.

```
asmlinkage long sys_close(unsigned int fd)
```

The **sys_close()** function (calls **__close_fd()**) releases the descriptor so that it can be reassigned.

If the **reference count of entries in the file table** associated with this descriptor is greater than 1, then the count decreases and the operation ends.

If the **reference count is 1**, then the position in the file table is released and the function is called for the corresponding **inode** from the **inode table**.

In case other processes refer to this **inode**, its **reference count** is reduced.

Otherwise, the function **frees** the **inode** from memory and, if necessary, **updates** its **contents** on the disk.

Asynchronous I/O – io_uring

Io_uring: started with a simple motivation: as devices get extremely fast, interrupt-driven work is no longer as efficient as polling for completions — a common theme that underlies the architecture of **performance-oriented I/O systems**.

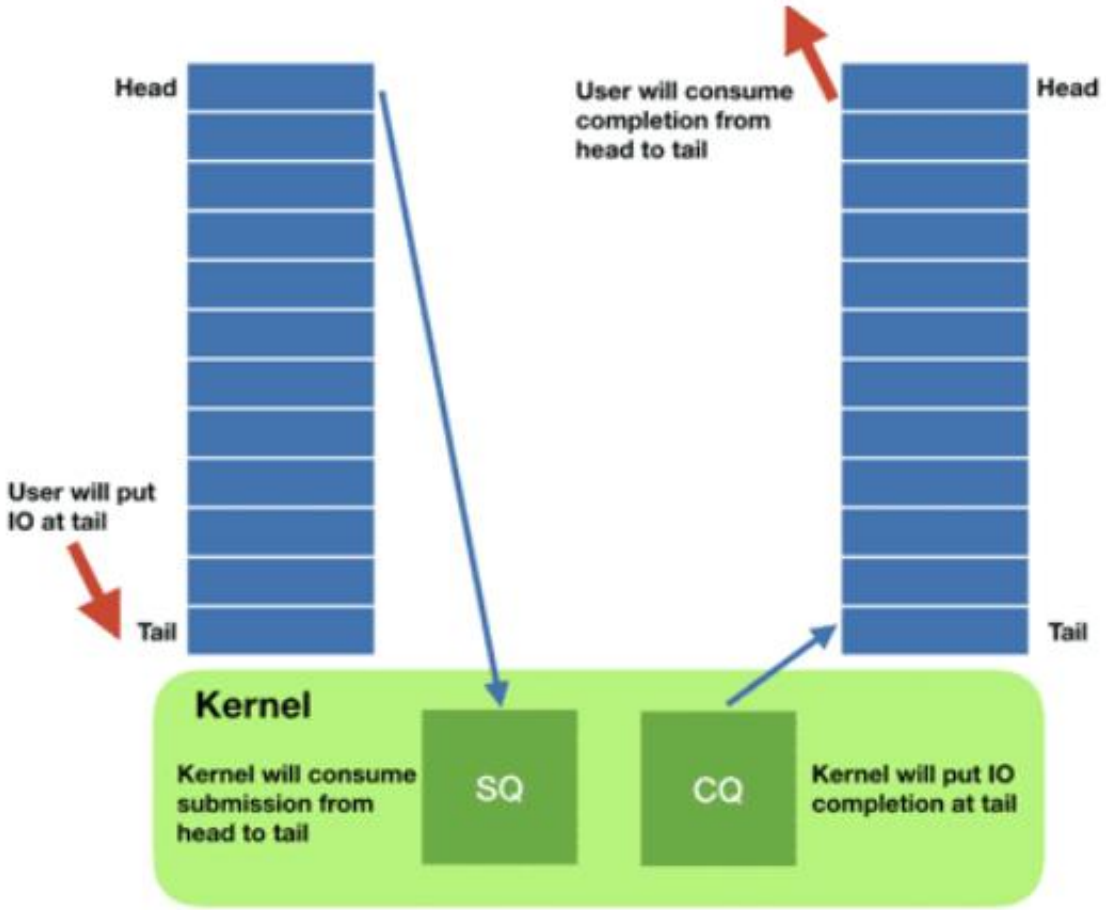
Fundamentally, **ring** based **communication channel**.

Built around a **ring buffer** in memory **shared** between **user space and the kernel**; that allows the submission of operations (and collecting the results) **without the need to call into the kernel** in many cases.

Instances of **SQ** and **CQ** live in a shared memory single-producer-single-consumer ring buffer between the kernel and the application.

Showed up in 5.1 release in May 2019. Author **Jens Axboe**.

liburing to the rescue.



How io_uring and eBPF Will Revolutionize Programming in Linux

Asynchronous I/O – io_uring

Io_uring is primarily a way of performing operations **asynchronously**. User space can queue operations in a ring buffer; the kernel consumes that buffer, executes the operations asynchronously, then puts the results into another ring buffer (the "completion ring") as each operation completes.

Initially, only basic I/O operations were supported, but the list of operations has grown over the years.

It works with any kind of I/O: cached files, direct-access files, blocking sockets.

It is flexible and extensible: new opcodes are added at high rate.

Some of the operations that io_uring supports: read, write, send, recv, accept, openat, stat, and even way more specialized ones like fallocate.

Faster IO through io_uring, Kernel Recepies, 2019, Jens Axboe

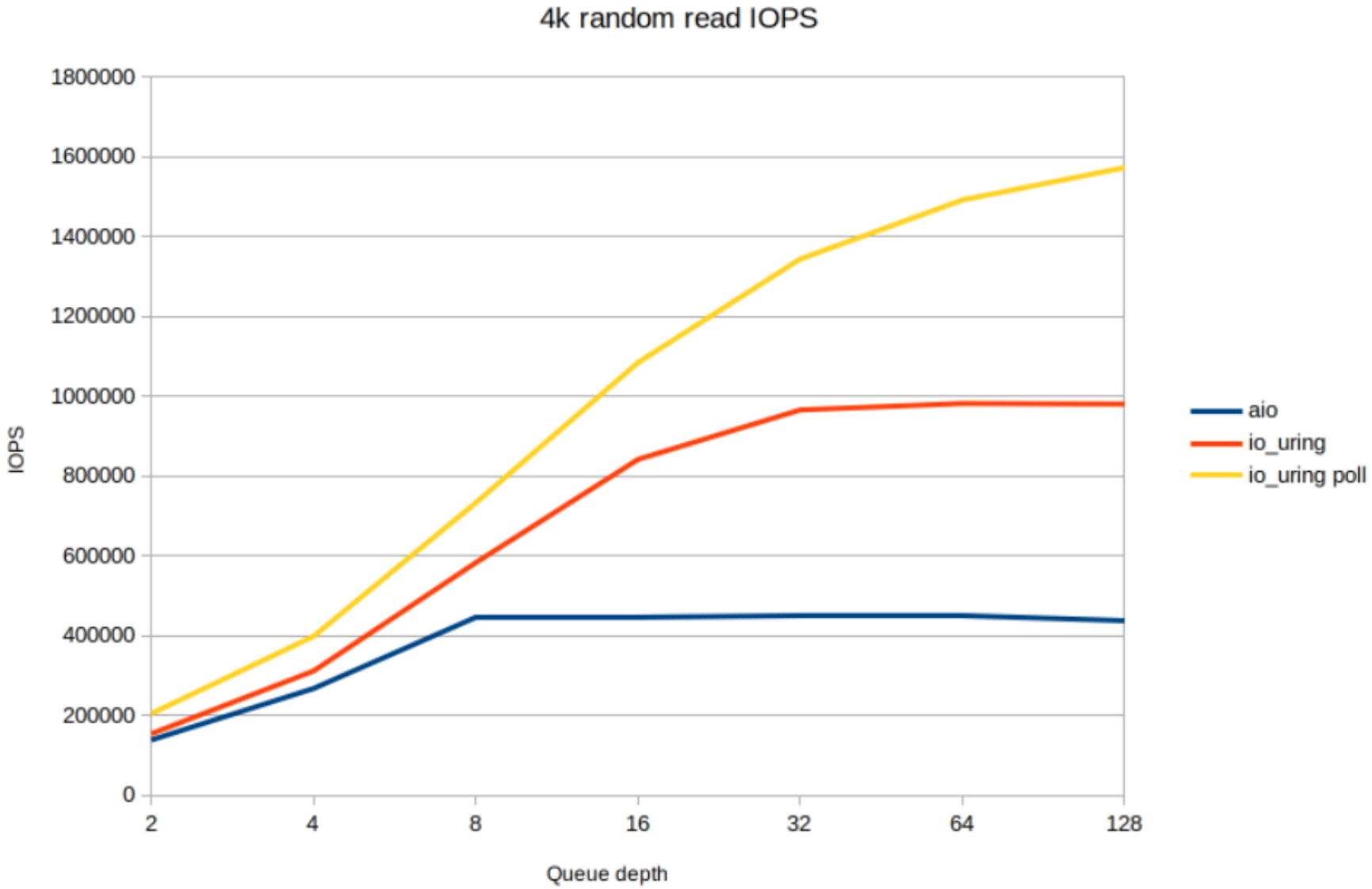
Efficient IO with io_uring, October 2019 (definitive guide).

Ringing in a new asynchronous I/O API, Jonathan Corbet, January 2020

The rapid growth of io_uring, Jonathan Corbet, January 2020

How io_uring and eBPF Will Revolutionize Programming in Linux, G. Costa, May 2020.

Asynchronous I/O – io_uring



Faster IO through io_uring, Kernel Recepies, 2019, Jen Axboe

Process creation in io_uring

Process creation in io_uring, Jonathan Corbet, December 20, 2024

io_uring can be thought of as a sort of **alternative system-call interface** for Linux that is inherently **asynchronous**.

An important io_uring feature, for the purposes of implementing something like posix_spawn(), is the ability to **create chains of linked operations**. When the kernel encounters a chain, it will only initiate the first operation; the next operation in the chain will only run after the first completes.

The failure of an operation in a chain will normally cause all remaining operations to be canceled, but a "hard link" between two operations will cause execution to continue regardless of the success of the first of the two.

Linking operations in this way essentially allows simple programs to be loaded into the kernel for asynchronous execution; these programs can run in parallel with any other io_uring operations that have been submitted.

But … security

New Linux Rootkit, Bruce Schneier, April 24, 2025

- The company has released a working rootkit called “Curing” that uses **io_uring**, a feature built into the Linux kernel, to stealthily perform malicious activities without being caught by many of the detection solutions currently on the market.
- This was just waiting to happen, if it hasn’t already happened earlier. In June 2023, Google reported on its Security blog:
  - in the past year, there has been a clear trend: 60% of the submissions exploited the io_uring component of the Linux kernel[…]. Furthermore, io_uring vulnerabilities were used in all the submissions which bypassed our mitigations.
  - To protect our users, we decided to limit the usage of io_uring in Google products.
- LWN waved a big red flag four years ago, so the only surprise here is that the rootkit is issued in public this much later, and apparently still works.

  Auditing io_uring, Jonathan Corbet, June 3, 2021

VFS data structures

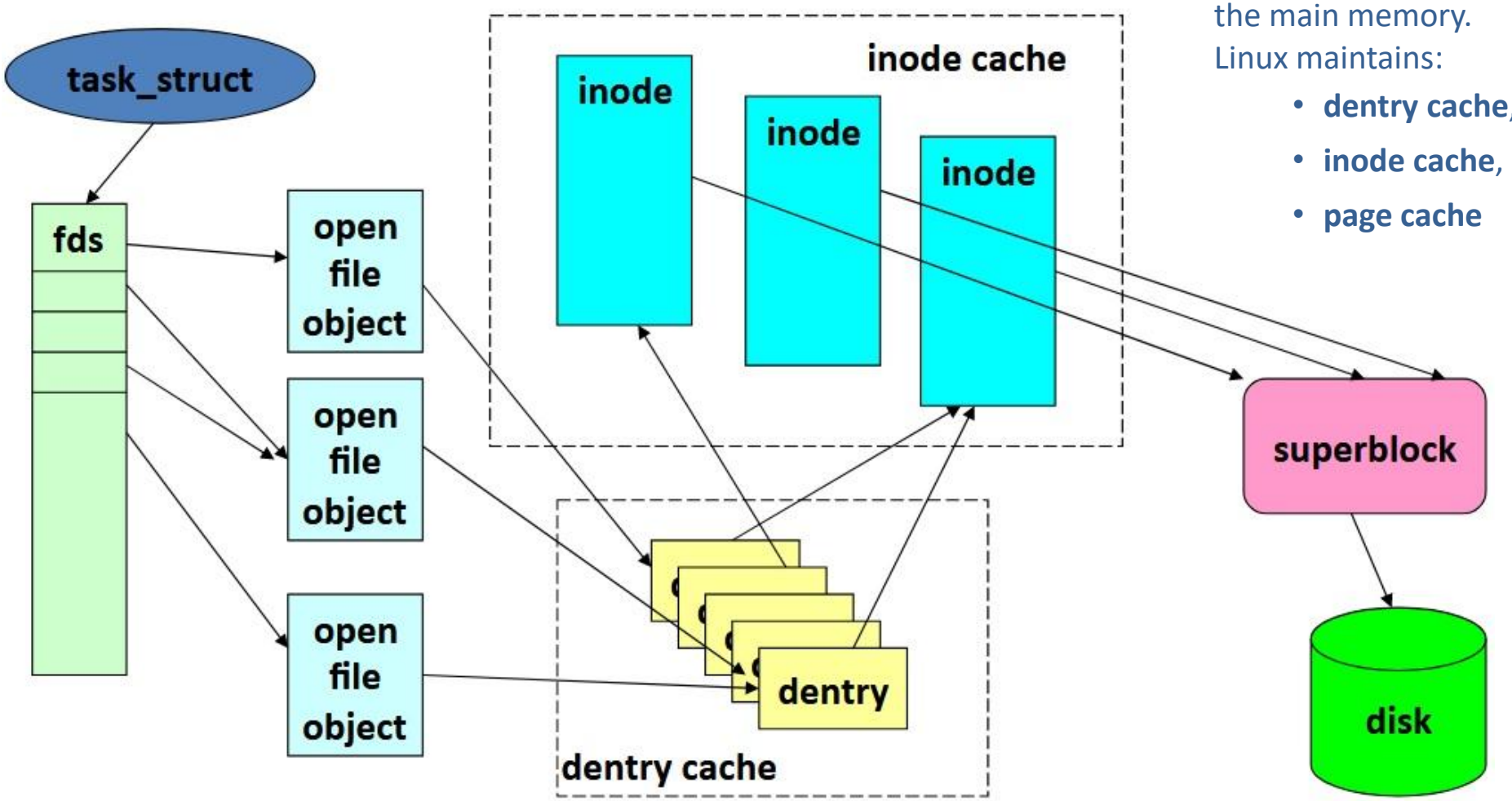
VFS presents an **object-oriented** approach to the file system and provides a **uniform** interface.

1. **Structure files_struct:** Stores process-local information about open file descriptors and file handling. The object exists only in the main memory.
2. **Structure fs_struct:** Stores process-local filesystem information. The object exists only in the main memory.
3. **Structure file:** Stores information about the relationship of the process to the open file. The object exists only in the main memory.
4. **Structure dentry:** Stores information about the relationship between directory entry and file. Each disk file system stores this information on the disk in its own way.
5. **Inode: File descriptor** containing all basic information about the file. For disk file systems, this object has its equivalent in the form of a **file control block** stored on disk. Each inode has an **inode number** associated with it that uniquely identifies the file in the file system.
6. **Structure address_space:** The main structure of the **page cache**. Maps all pages of one file (that is, the image of the file's address space in memory) to disk blocks (that is, the image of the file's address space on the disk).
7. **Superblock:** Stores mounted file system information. For disk file systems, the object has its equivalent in the form of a **file system control block** stored on disk.

VFS data structures

For efficiency reasons, some of the structures stored on the disk are temporarily held in caches in the main memory.
Linux maintains:

- **dentry cache**,
- **inode cache**,
- **page cache**



VFS data structures are described in [Documentation/filesystems/vfs.rst]

VFS – objects (source: Kaustubh Joshi)

Process data structures with file information – fs_struct

Fields in the **process descriptor** (**task_struct**) describing the relationship of the process with the files.

```
struct fs_struct *fs;          /* file system information */
struct files_struct *files;    /* information about open files */
```

The main fields are a link to a description of the **current process directory** (**pwd**) and the **root** of the file system (**root**), which create a process's **context** in the file system.

The **mnt** field is an object that describes the **mounted file system**.

```
struct path {
    struct vfsmount *mnt;
    struct dentry *dentry;
} __randomize_layout;

struct fs_struct {
    int users;          /* user count */
    spinlock_t lock;  /* per-structure lock */
    seqcount_t seq; /* Reader/writer consistent mechanism */
                        /* without starving writers */
    int umask;
    int in_exec;       /* currently executing a file */
    struct path root, pwd;
} __randomize_layout;
```

Process data structures with file information – fs_struct

The **users** field specifies the number of processes sharing the same **fs_struct** structure (this number increases when calling **do_fork()** with the **CLONE_FS** flag set).

The **copy_fs()** function called inside **do_fork()** illustrates how this field is handled.

In /kernel/fork.c

```
static int copy_fs(unsigned long clone_flags, struct task_struct * tsk)
{
    struct fs_struct *fs = current->fs;
    if (clone_flags & CLONE_FS) {/* tsk->fs is already what we want */
        spin_lock(&fs->lock);
        if (fs->in_exec) {
            spin_unlock(&fs->lock);
            return -EAGAIN;
        }
        fs->users++;
        spin_unlock(&fs->lock);
        return 0;
    }
    tsk->fs = copy_fs_struct(fs);
    if (!tsk->fs)
        return -ENOMEM;
    return 0;
}
```

Process data structures with file information – files_struct

Contains an array indexed by natural numbers that match the **descriptors of open files**. The entry value in this array is a link to the **global** (within the file system) **open file table**.

The **count** field – the number of processes sharing the same **files_struct** structure (increases when the function **do_fork()** is called with the **CLONE_FILES** flag set).

The **files_struct** contains both an **instance** of the **fdtable**, and a **pointer** to this structure, because synchronization is provided using the **RCU** mechanism, so that reading does not require a lock.

- The **max_fds** field specifies the current maximum number of file objects and file descriptors that the process can handle (up to **rlimit**).
- **fd** is an array of pointers to **file** structures. The file descriptor passed to the user process is the index in this table. The current size of the array is defined by **max_fds**.
- **open_fds** is a pointer to a bit array. The table contains one bit for each descriptor; 1 means the descriptor is in use, 0 means it is free. The **next_fd** field suggests where to start looking for the next free descriptor.
- **close_on_exec** is also a pointer to a bit array, it indicates these files that should be closed on the **exec()** system call.

```
#define BITS_PER_LONG   32  // or 64
#define NR_OPEN_DEFAULT   BITS_PER_LONG

struct fdtable {
   unsigned int  max_fds;
   struct file __rcu  **fd;     /* current fd table*/
   unsigned long  *close_on_exec;
   unsigned long  *open_fds;
   struct rcu_head  rcu;
};
struct files_struct {
   atomic_t  count;
   struct fdtable __rcu  *fdt;
   struct fdtable  fdtab;
   spinlock_t  file_lock;     // slightly simplified
   int  next_fd;
   unsigned long  close_on_exec_init[1];
   unsigned long  open_fds_init[1];
   struct file __rcu  * fd_array[NR_OPEN_DEFAULT];
};
```

The **fdtable** i **files_struc**t structures **do not duplicate** information – elements of the **files_struct** structures are real instances of certain data structures, while the elements of the **fdtable** structure are pointers to them.

The **fd**, **open_fds** and **close_on_exec** fields are initialized to point to the appropriate structures in **files_struct**.

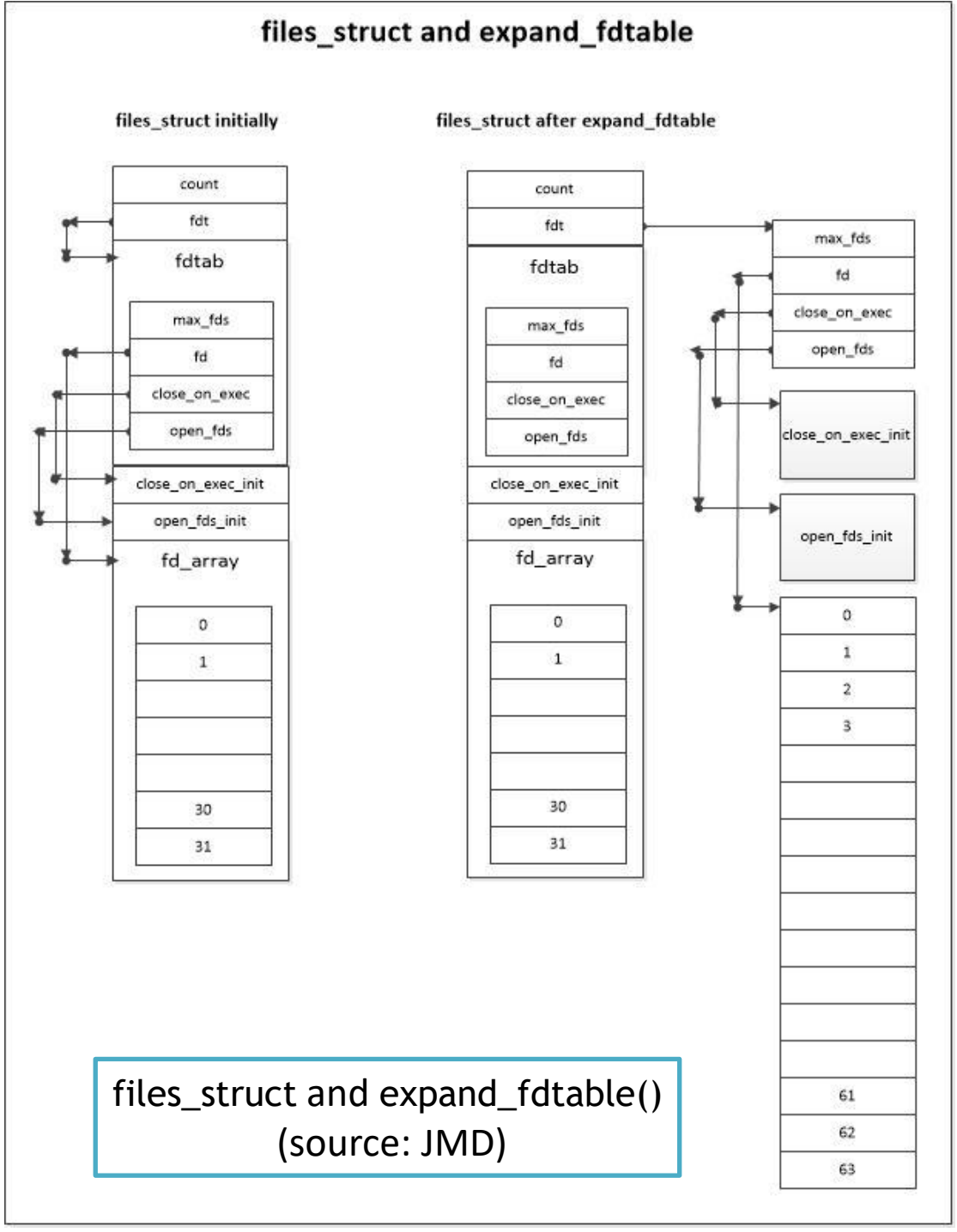
Initially, **fdtable** is placed in **files_struct**.

After expanding **fdtable**, the **files->fdt** points to the new structure.

When a process is created, **fd_array** **[NR_OPEN_DEFAULT]** is preallocated for the first **32** (**64**) files to be open.

When the kernel wants to open a file and there is not enough space in **files_struc**t, it calls the **expand_files()** function.

The function checks if an extension is possible and if so, calls the **expand_fdtable()** function.



files_struct and expand_fdtable() (source: JMD)

Process data structures with file information – files_struct

The **alloc_fdtable()** allocates an array of file descriptors with the maximum number of possible entries and reserves memory for enlarged bitmaps. It then copies the previous contents of the file descriptor table into the new enlarged table.

Switching the pointer **files->fdt** to the new instance is supported by the RCU **rcu_assign_pointer()** function. Next the memory occupied by the old file descriptor table can be freed.

All elements of the file descriptor table have been intentionally placed in a separate **fdtable** structure so that **they can be read atomically without locking, using the RCU mechanism**.

The **fdtable** structure is released using the RCU, and as a result **readers – without locking – can see either the old fdtable or the new one**.

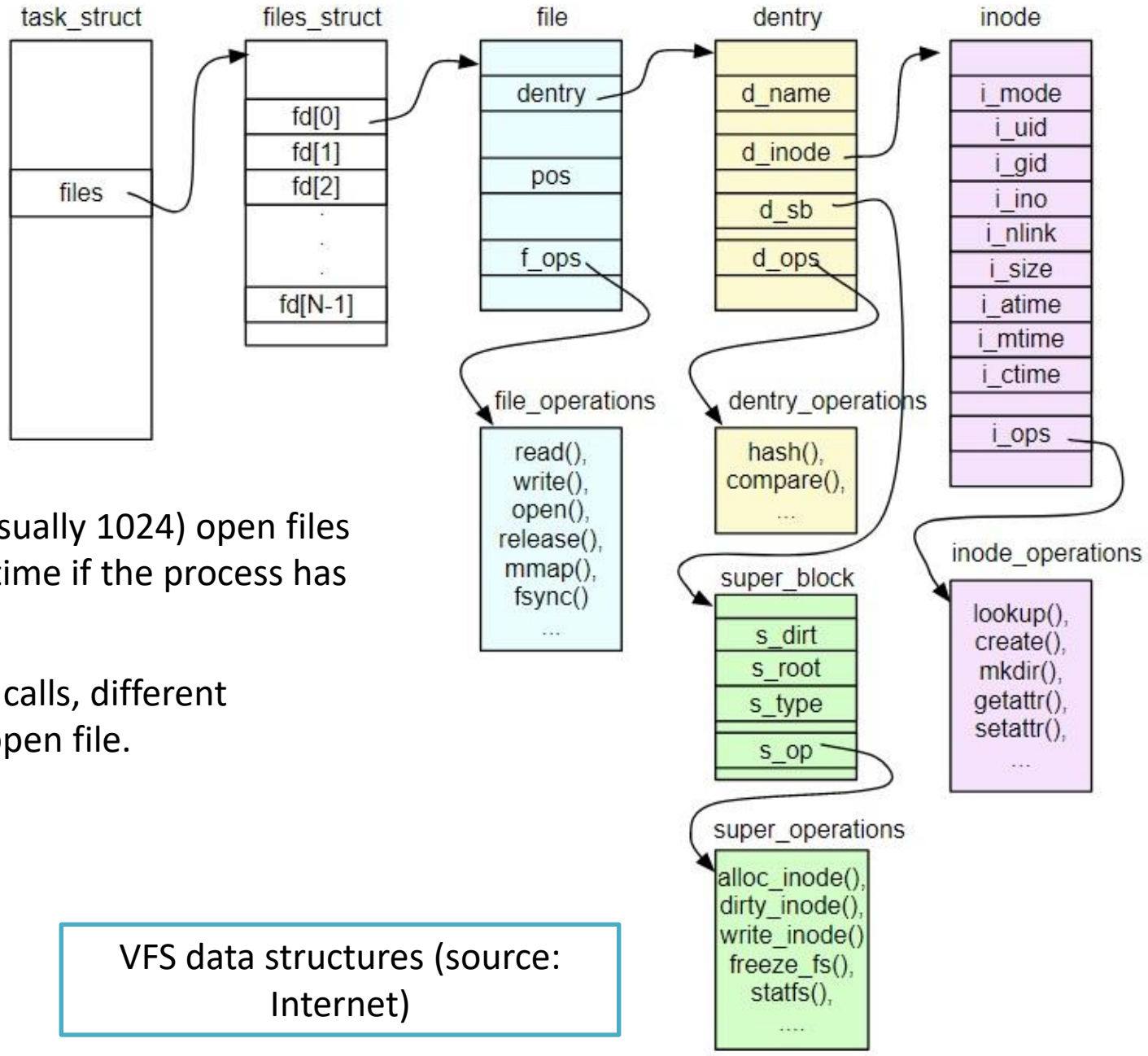
```
static int expand_fdtable
                (struct files_struct *files, int nr)
{
  struct fdtable  *new_fdt,  *cur_fdt;

  spin_unlock(&files->file_lock);
  new_fdt = alloc_fdtable(nr);
  spin_lock(&files->file_lock);
  ...
  cur_fdt = files_fdtable(files);
  copy_fdtable(new_fdt, cur_fdt);
  ...
  rcu_assign_pointer(files->fdt, new_fdt);
  if (cur_fdt != &files->fdtab)
      call_rcu(&cur_fdt->rcu, free_fdtable_rcu);
                ...
  return 1;
}
```

Simplified version

The detailed description is included in Documentation/filesystems/files.rst.

Process data structures with file information



The process can have **NR_OPEN** (usually 1024) open files (this limit can be **increased** at run time if the process has superuser privileges).

Thanks to **dup()** and **fork()** system calls, different descriptors can refer to the same open file.

VFS data structures (source: Internet)

Process data structures with file information – file

The **file** structure describes one element in the open file table.
Each call to **open()** assigns a new position in this table to the file being opened.
Positions can be shared (**f_ref** greater than 1) – this is achieved by using **dup()** or **fork()** .

The **fu_llist** links a structure to one (and only one) of the lists:

- list of all open files in a given file system (identified by its **superblock**) or
- list of unused structures.

The **dentry** field contains a link to the **dentry** structure of this file, which is created when the **path name** of the file is mapped to the **inode** number.

The **f_mode** field stores information about the mode in which the file was opened.

The **f_pos** field contains the current position in the file. In the current version for i386, it is **64** bits long (correlated with the **maximum file size** allowed). This field must be placed in this structure, because many processes can have access to the same file at the same time.

The **f_op** field contains a link to an array of pointers to the methods that can be used on this file. This field obtains the value from **inode→i_fop**.

```
struct llist_node {
    struct llist_node *next;
};

struct path {
    struct vfsmount  *mnt;
    struct dentry  *dentry;
} __randomize_layout;

struct  file  {
    file_ref_t              f_ref;
    spinlock_t              f_lock;
    fmode_t                 f_mode;
    const struct file_operations  *f_op;
    struct address_space    *f_mapping;
    struct inode            *f_inode;
    unsigned int            f_flags;
    struct path             f_path;
    loff_t                  f_pos;
    union {
        struct llist_node    fu_llist;
        .....
    } ;
    ....
```

Simplified version

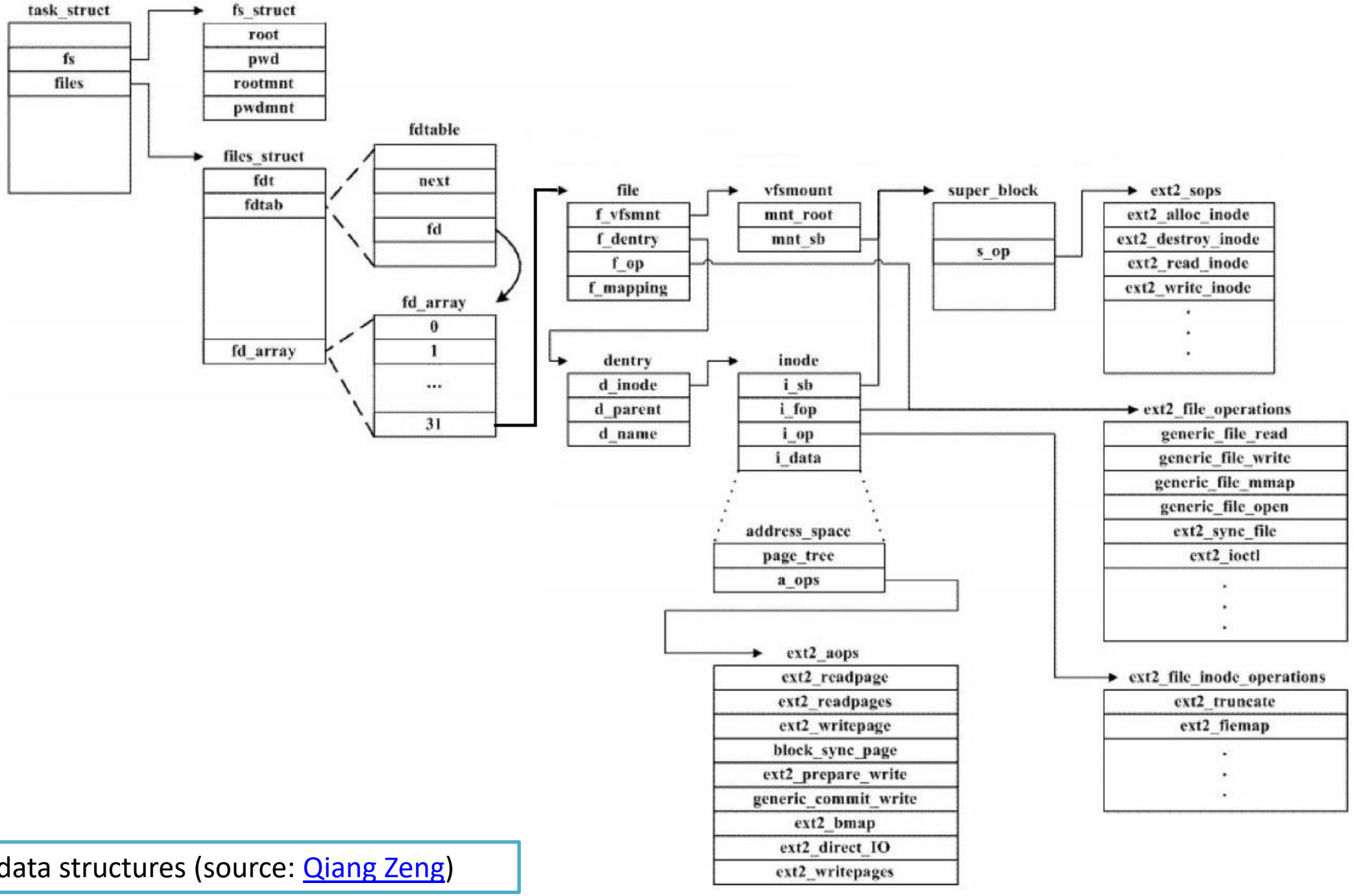
In version 6.12 the size of struct file within the kernel has been reduced from **232** bytes to **184** (3 cachlines).

SKIP

Process data structures with file information – file

```
struct file_operations {
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
    ssize_t (*read_iter) (struct kiocb *, struct iov_iter *);
    ssize_t (*write_iter) (struct kiocb *, struct iov_iter *);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*lock) (struct file *, int, struct file_lock *);
    ....
} __randomize_layout;
```

Type **iov_iter** contains the **address** and the **length** of the user mode buffer that shall receive the data read from the file. Type **kiocb** is used to keep track of the completion status of an ongoing synchronous or asynchronous I/O operation.



VFS data structures (source: Qiang Zeng)

Process data structures with file information – inode

Inode, i.e. the **file descriptor**, is loaded to main memory when opening the file and extended with additional fields (e.g. pointers and inode number). **Size – 568 bytes**.

Type	Name	Description
umode_t	i_mode	File type: regular (IS_REG), directory (IS_DIR), named pipe (IS_FIFO), special character, special block, symbolic link, socket, 0 when free; also file access rights for everyone, the user's group and the user
kuid_t	i_uid	file owner ID
kgid_t	i_gid	file owner group ID
unsigned int	i_flags	flags specifying how to use the inode as well as the file it describes (e.g. S_WRITE – to write ...)
const struct inode_operations	*i_op	set of operations on inodes closely related to the file system used – e.g. EXT2, EXT4
struct super_block	*i_sb	pointer to the superblock of the device to which the inode belongs
struct address_space	*i_mapping	pointer to the address_space object associated with this file

Process data structures with file information – inode

SKIP

```
struct inode_operations {
        struct dentry * (*lookup) (struct inode *,struct dentry *, unsigned int);
        int (*readlink) (struct dentry *, char __user *,int);
        int (*create) (struct inode *,struct dentry *, umode_t, bool);
        int (*link) (struct dentry *,struct inode *,struct dentry *);
        int (*unlink) (struct inode *,struct dentry *);
        int (*symlink) (struct inode *,struct dentry *,const char *);
        int (*mkdir) (struct inode *,struct dentry *,umode_t);
        int (*rmdir) (struct inode *,struct dentry *);
        int (*mknod) (struct inode *,struct dentry *,umode_t,dev_t);
        int (*rename) (struct inode *, struct dentry *,
                        struct inode *, struct dentry *, unsigned int);
        int (*setattr) (struct dentry *, struct iattr *);
        int (*getattr) (const struct path *, struct kstat *, u32, unsigned int);
        ...
} ____cacheline_aligned;
```

Simplified version

Process data structures with file information – inode

Typ	Nazwa	Opis
unsigned long	i_ino	inode number in the disk inode table
dev_t	i_rdev	indicates the actual device (for disk device, the index in the struct block device array)
loff_t	i_size	file size in bytes (64 bits, i.e. 2^{64})
struct timespec	i_atime	time of last access to the file
struct timespec	i_mtime	time the file was last modified
struct timespec	i_ctime	file creation time
unsigned short	i_bytes	number of bytes in the last block of the file
blkcnt_t	i_blocks	number of disk blocks occupied by the file
struct hlist_node	i_hash	Pointers connecting the inode to a list with other located in the hash table under the same number (1)
struct list_head	i_lru	inode LRU list (2)
struct list_head	i_sb_list	pointers maintaining a list of inodes of one superblock (3)

Process data structures with file information – inode

1. Each **inode** is in a **hash table** of bidirectional lists, **inode_hashtable**. It is identified by its **number** and **device number**.

   A hash table allows quick access to an inode in memory.

   Each inode can be at one time only in a **hash table**, only in one **file system** (represented by a superblock), and in one and only one of the lists of **used** inodes, **unused** inodes and **dirty** inodes.

*The memory can contain at most **one copy** of a given disk inode.*

```
/*
 *   Add an inode to the inode hash for this superblock.
 */
void __insert_inode_hash(struct inode *inode, unsigned long hashval)
{
   struct hlist_head *b = inode_hashtable + hash(inode->i_sb, hashval);

   spin_lock(&inode_hash_lock);
   spin_lock(&inode->i_lock);
   hlist_add_head(&inode->i_hash, b);
   spin_unlock(&inode->i_lock);
   spin_unlock(&inode_hash_lock);
}
```

Process data structures with file information – inode

2. Every **unused** and **clean** inode is on the LRU list of its **superblock**

   list_lru_add_obj(&inode->i_sb->s_inode_lru, &inode->i_lru)

   (see **inode_add_lru**(struct inode *inode)

3. In addition, each inode is on a **bidirectional list of all inodes of its superblock** (originating from the same file system). The list is indicated by the **s_inodes** field of the **superblock**, and the **i_sb_list** field of the **inode** is used to create links.

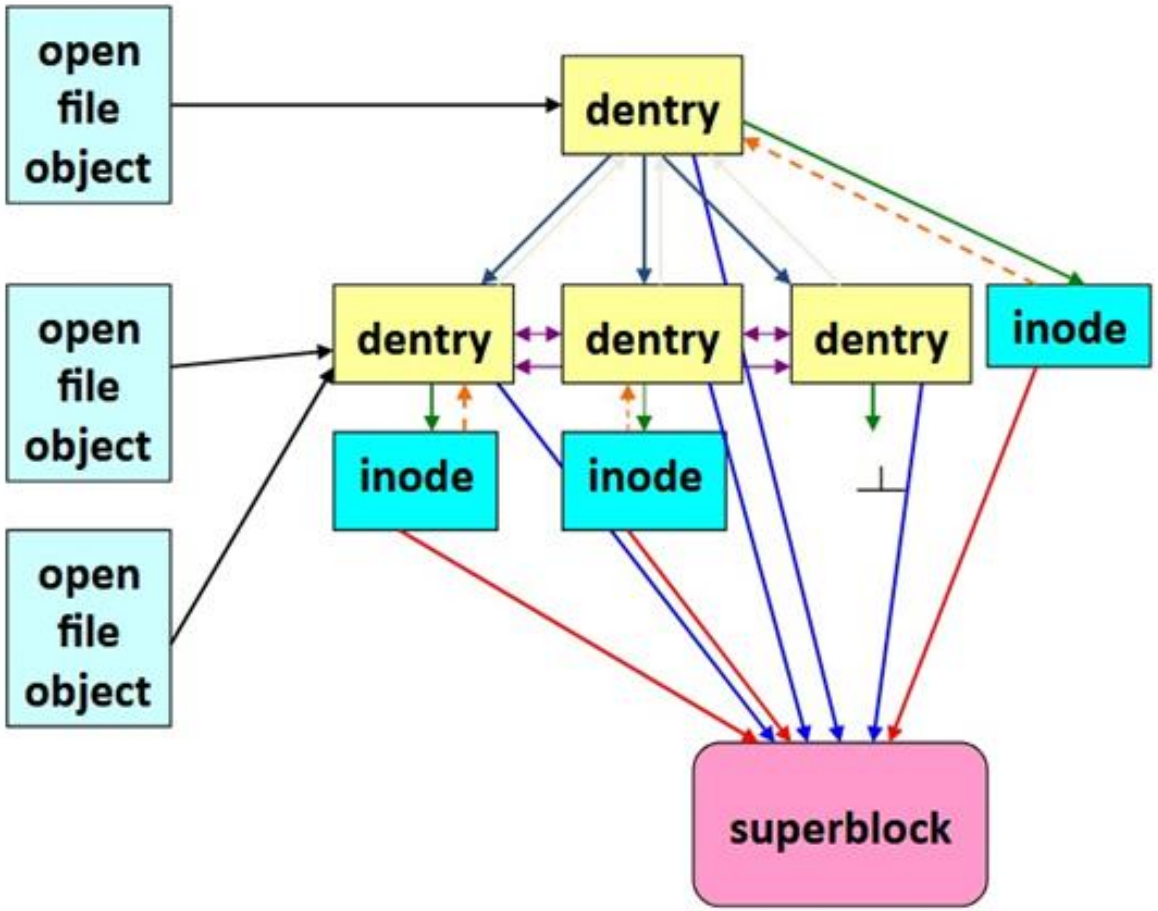
   Used e.g. in the **invalidate_inodes()** function which attempts to free all inodes for a given superblock.

```
/**
 * Add an inode to the superblock list of inodes
 */
void inode_sb_list_add(struct inode *inode)
{
   spin_lock(&inode->i_sb->s_inode_list_lock);
   list_add(&inode->i_sb_list, &inode->i_sb->s_inodes);
   spin_unlock(&inode->i_sb->s_inode_list_lock);
}
```

Process data structures with file information – inode

Typ	Nazwa	Opis
struct hlist_head	i_dentry	list of all dentry structures related to this inode
struct mutex	i_mutex	mutex for locking
const struct file_operations	*i_fop	set of file operations
struct address_space	i_data	the address_space object of this file
struct list_head	i_devices	for a block and character device – a list of inodes that can be used to access this device
struct pipe_inode_info	i_pipe	(union field) used when the inode describes pipe
struct block_device	*i_bdev	(union field) pointer to the block device handler
struct cdev	*i_cdev	(union field) pointer to the character device handler
unsigned long	i_state	If I_DIRTY_SYNC, I_DIRTY_DATASYNC or I_DIRTY_PAGES, the inode is dirty; if I_LOCK, then the inode is in the process of transferring, if I_FREEING is in the process of releasing, I_NEW – newly allocated, unfilled
void	*i_private	private file system or device pointer

Process data structures with file information



f_dentry	
d_subdirs	
d_inode	
d_child	
d_parent	
i_sb	
d_sb	
i_dentries	

VFS – objects (source: Kaustubh Joshi)

Process data structures with file information – superblock

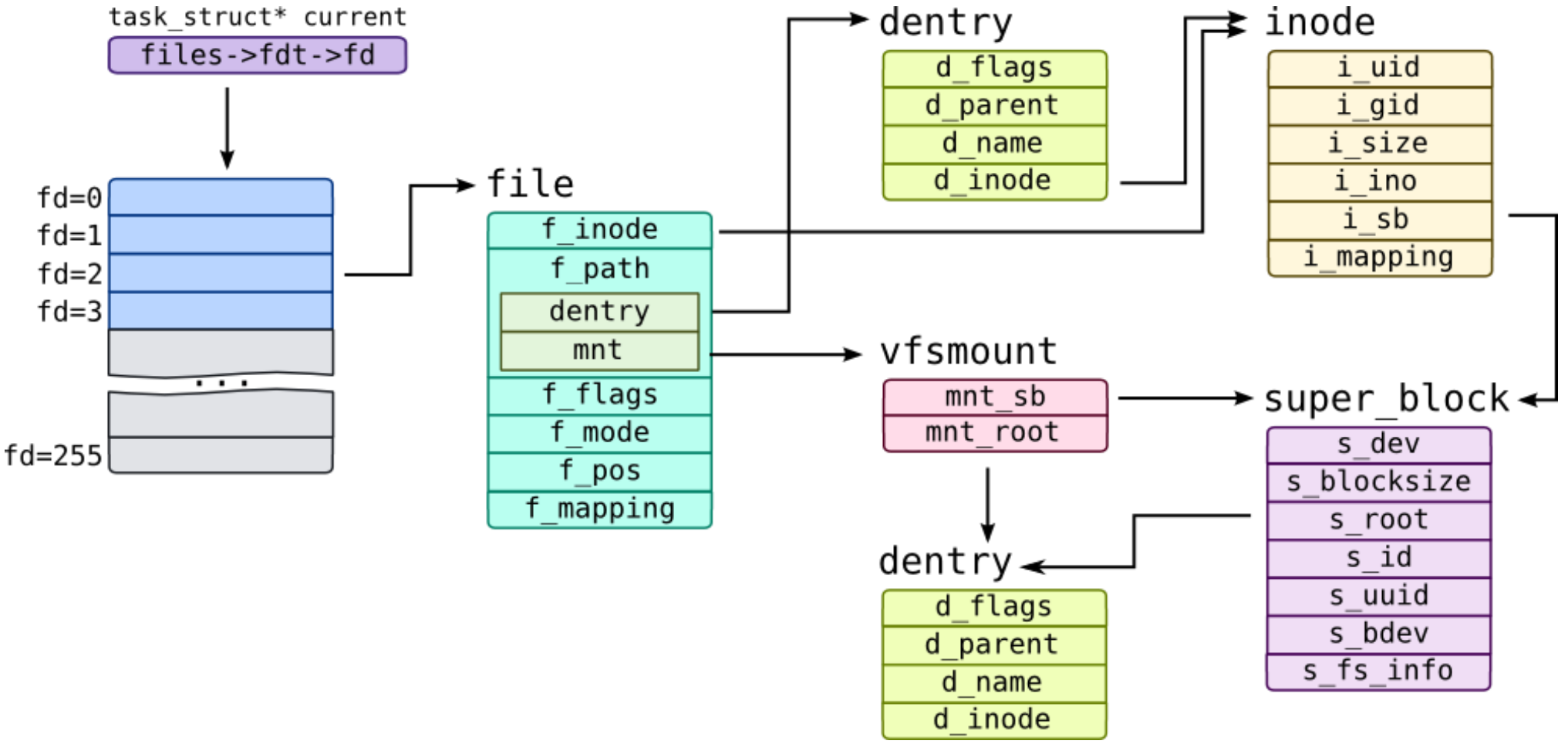
```
struct super_block {                                    Simplified version
    struct list_head    s_list;
    dev_t               s_dev;
    unsigned char       s_blocksize_bits;
    unsigned long       s_blocksize;
    loff_t              s_maxbytes;  /* Max file size */
    struct file_system_type *s_type;
    const struct super_operations *s_op;
    struct dentry       *s_root;
    int                 s_count;
    struct list_lru     s_dentry_lru;
    struct list_lru     s_inode_lru;
    struct list_head    s_inodes;      /* all inodes */
    .......
```

Contains basic information about the mounted file system and corresponds to the **physical disk superblock**.

```
struct super_operations {
    struct inode *(*alloc_inode)
                (struct super_block *sb);
    void (*destroy_inode)(struct inode *);
    void (*free_inode)(struct inode *);
    void (*dirty_inode) (struct inode *, int flags);
    int (*write_inode) (struct inode *, struct writeback_control *wbc);
    int (*drop_inode) (struct inode *);
    void (*put_super) (struct super_block *);
    int (*sync_fs)(struct super_block *sb, int wait);
    int (*statfs) (struct dentry *, struct kstatfs *);
    void (*umount_begin) (struct super_block *);
    ...                                          Simplified version
```

Each modification of the superblock sets the appropriate flag. Linux periodically searches for superblocks and updates disk information.

Process data structures with file information



VFS – objects (source: Sergey Klyaus)

Pipe vs FIFO

Inode of the **pipe** in field **i_pipe** has a pointer to the **pipe_inode_info** object.

Simplified version

```
struct pipe_inode_info {
    struct mutex mutex;          // mutex protecting the whole thing
    wait_queue_head_t rd_wait, wr_wait;   // wait point in case of empty/full pipe
    unsigned int head;           // the point of buffer production
    unsigned int tail;           // the point of buffer consumption
    unsigned int ringsize;       // total number of buffers (should be a power of 2)
    unsigned int readers;        // number of current readers of this pipe
    unsigned int writers;        // number of current writers of this pipe
    unsigned int files;          // number of struct file referring this pipe
    struct page *tmp_page;       // cached released page
    struct pipe_buffer *bufs;    // the circular array of pipe buffers
};
```

Pipe vs FIFO

A **pipe** can have up to **16** buffers, each of them is of the **pipe_buffer** type, together they form a **circular buffer**.

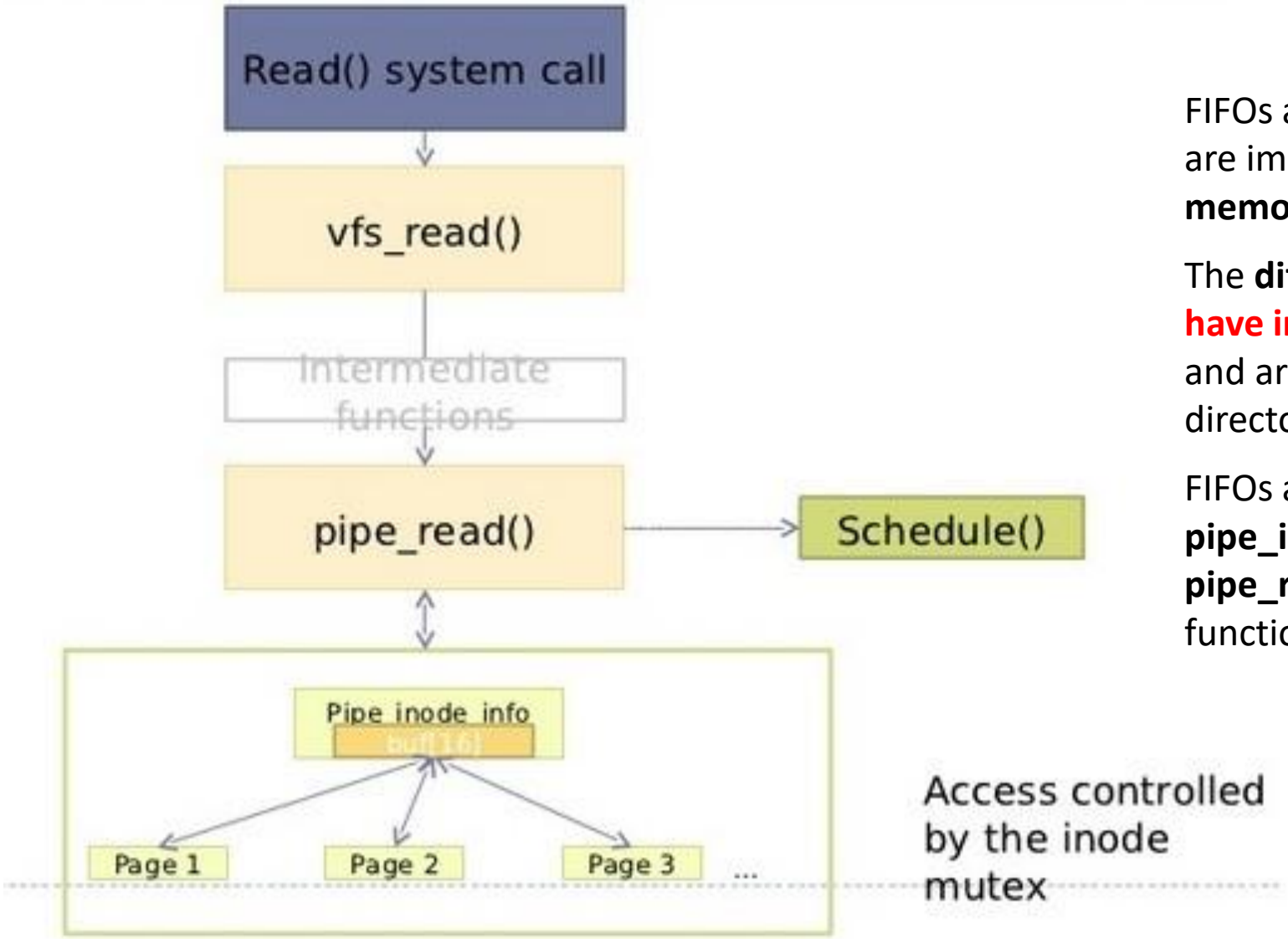
```
#define PIPE_DEF_BUFFERS  16

struct pipe_buffer {
        struct page *page;          // the page containing the data for the pipe buffer
        unsigned int offset, len;   // offset of data inside the page
        const struct pipe_buf_operations *ops; // operations associated with this buffer
        unsigned int flags;         // among others 'is on the LRU list', 'atomically mapped'
        unsigned long private;      // private data owned by the ops
};
```

Pipes are implemented as **VFS objects** and have no representation on disk file systems. Their inodes are part of the special **pipefs** file system, which has **no mount point** in the system directory tree, so it is not visible to users.

The disadvantage of pipes is that an existing link cannot be opened, so it cannot be **shared** by any **two processes** (but only by processes with a common ancestor).

Pipe vs FIFO



FIFOs are **similar** to **pipes** in that they are implemented as **buffers** in **memory**.

The **difference** is that they FIFOs **have inodes** in the **disk file system** and are mounted in the system directory tree (not pipefs).

FIFOs are also implemented using the **pipe_inode_info** structure (and the **pipe_read()** and **pipe_write()** functions).

Pipe (source: Divye Kapoor)