Linux Kernel Implementation of Pipes, FIFOs and other Filesystems

Divye Kapoor
B.Tech (IDD) CSI – IV
061305

Userspace v/s Kernel Space

Conventional Operating Systems generally segregate virtual memory into kernel space and user space

- New research OS's (Singularity etc.) maintain a single virtual address space for all processes and depend on language VMs for maintaining process isolation.

Kernel space is strictly reserved for running the kernel, kernel extensions (modules) and device drivers

Memory is generally not swapped out

User space is where generally all user processes run

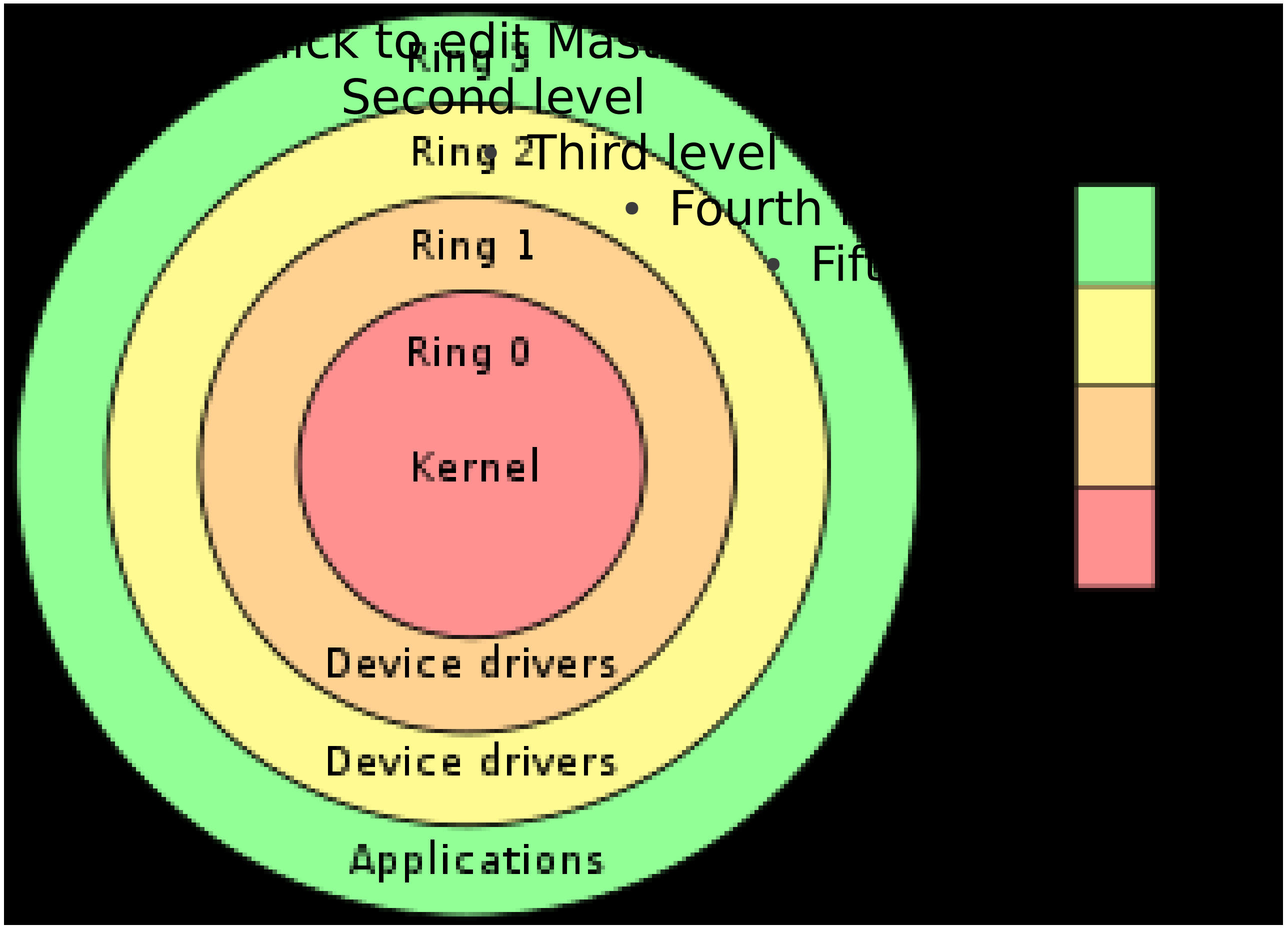
Each process has its own virtual address space and memory may be swapped out.

There is hardware support for providing the distinction in privilege levels in user space and kernel space

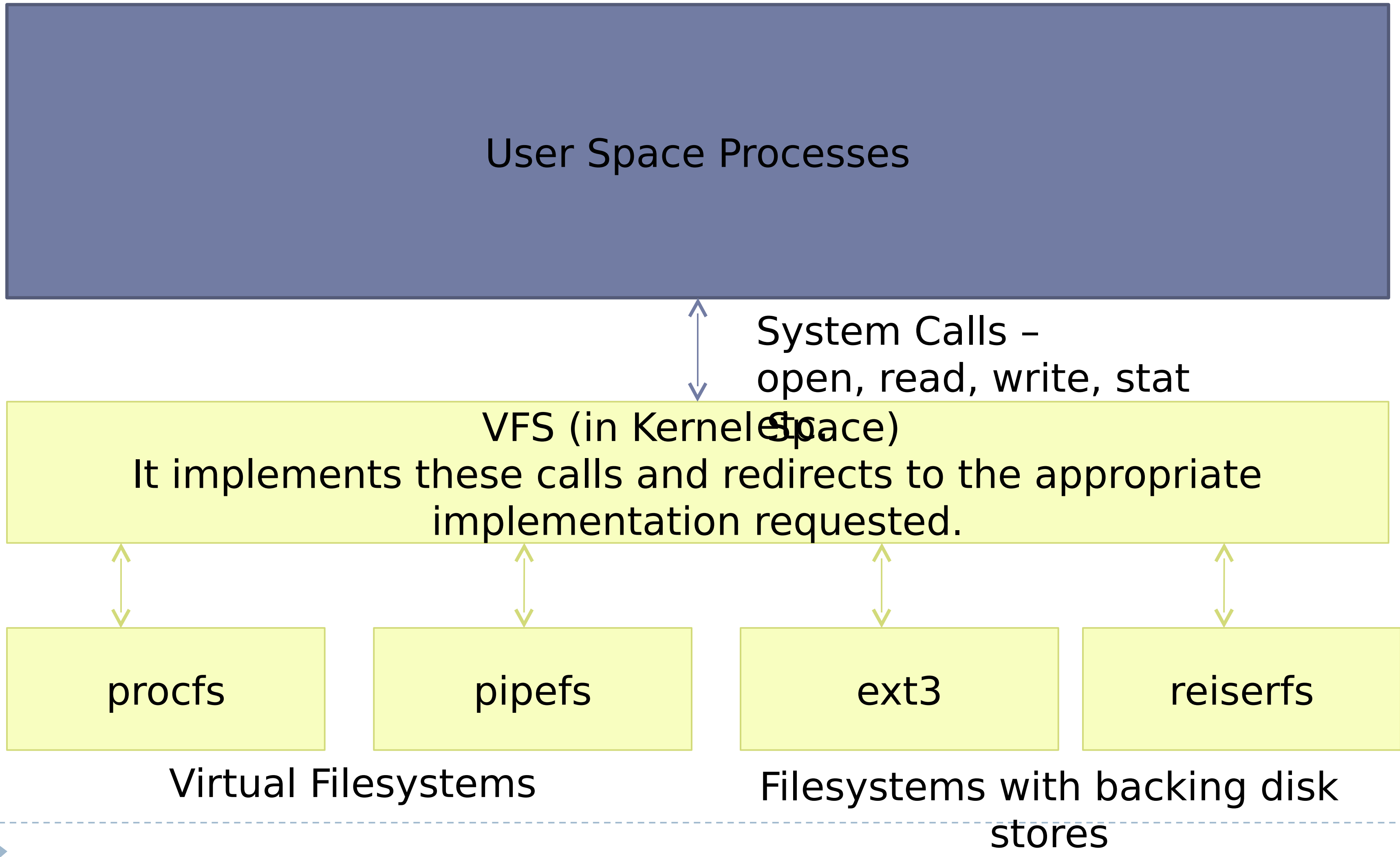
For the x86 architecture processors, this is provided by privilege rings

Privilege Rings

ick to edit Mast
Second level
- Third level
  - Fourth
    - Fift



General Context of the Filesystem



Linux Virtual Filesystem (VFS)

It is the software layer in the kernel that provides a uniform filesystem interface to userspace programs

It provides an abstraction within the kernel that allows for transparent working with a variety of filesystems.

Thus it allows many different filesystem implementations to coexist freely

Each pipe and FIFO is implemented as a “file” mounted on the pipefs filesystem.

- A FIFO is just a thin wrapper around a pipe

Inodes, Direntry and File

Inodes provide a method to access the actual data blocks allocated to a file.

struct inode

Inodes are created in the context of a Direntry which represents (conceptually) the directory in which the file resides with respect to the root of the filesystem on which it is located.

struct direntry

Every file is represented in the kernel as an object of the *file* structure. It requires a direntry and an inode provided to it.

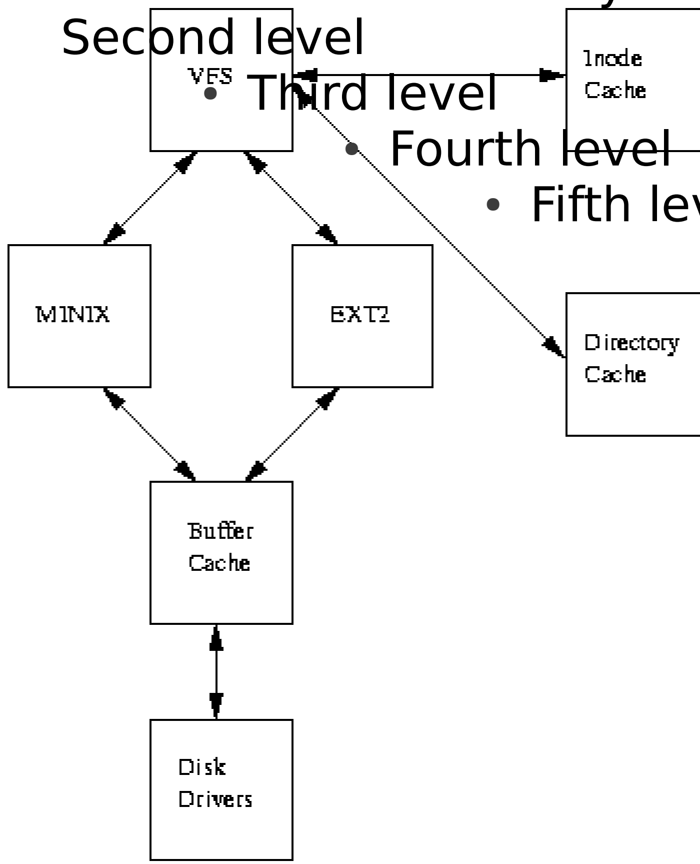
struct file

Internal Structure of VFS

Click to edit Master text styles

Second level

- Third level
  - Fourth level
    - Fifth level



Filesystem Registration

Somewhere in the kernel load path, the inbuilt file systems are loaded.

Filesystems that are implemented as modules register themselves with the kernel (VFS) by using the following API at module load time.

```
#include <linux/fs.h>

   extern int register_filesystem(struct file_system_type *);
   extern int unregister_filesystem(struct file_system_type *);
```

Filesystem Registration…

```
struct file_system_type {
      const char *name; // "pipefs"
      int fs_flags; // "mount flags"
    int (*get_sb) (struct file_system_type *, int,
              const char *, void *, struct vfsmount *); // "get superblock" on mount
    void (*kill_sb) (struct super_block *); // "destroy superblock" on umount
        struct module *owner; // which module is responsible for accesses
        struct file_system_type * next; // VFS internal list of filesystem types
/* others */
};
```

- The superblock is created by the VFS whenever a device of the appropriate filesystem type is mounted.

- The superblock provides the interface for VFS to access filesystem specific functions.

get_sb()

The get_sb() function returns the superblock structure which contains a pointer to the super_operations structure.

Functions specified in the super_operations specify how inodes are handled by the specific filesystem implementation.

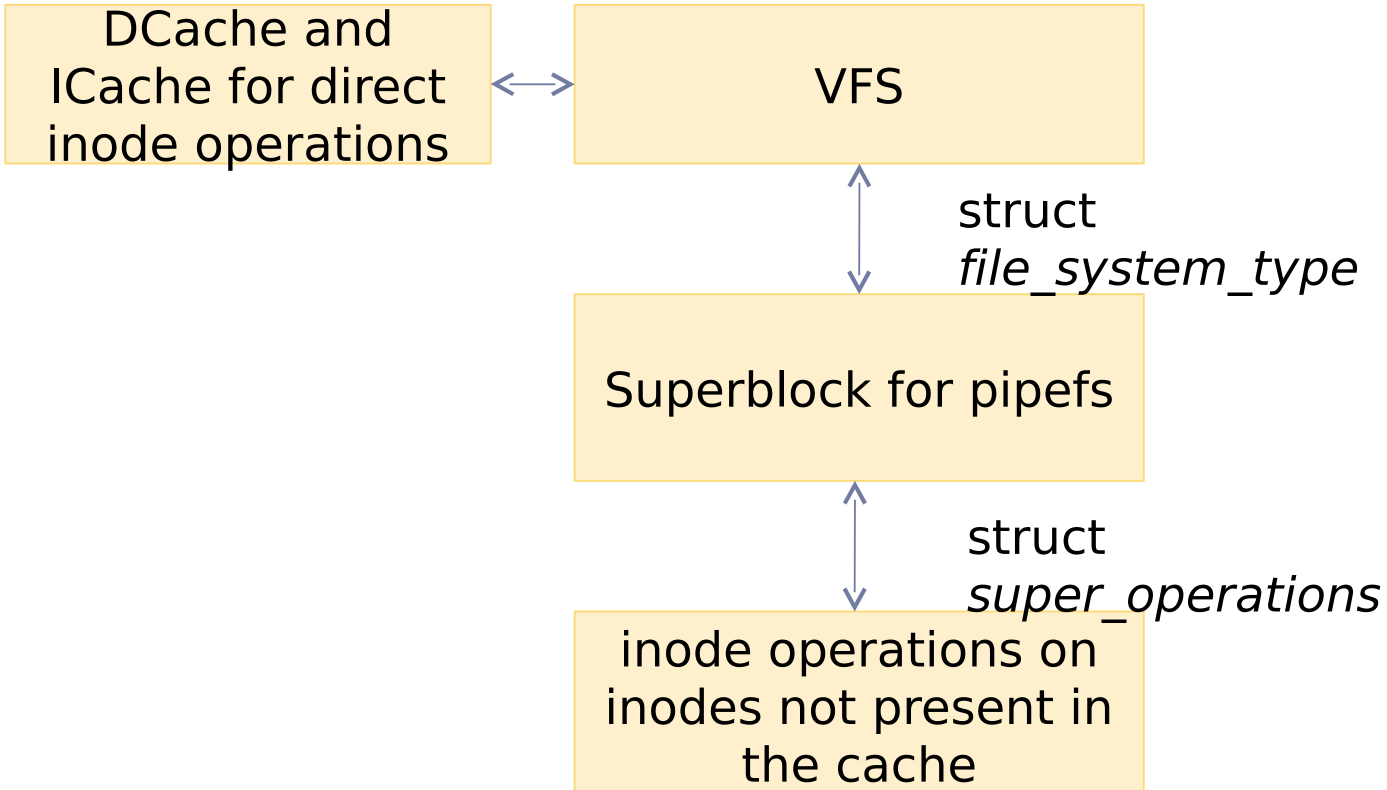
```
struct super_operations {
        struct inode *(*alloc_inode)(struct super_block *sb);
        void (*destroy_inode)(struct inode *);

        void (*dirty_inode) (struct inode *);
        int (*write_inode) (struct inode *, int);
        void (*drop_inode) (struct inode *);
        void (*delete_inode) (struct inode *);

   /* lots of other members */

};
```

Where are we now?



The pipe() system call

```
SYSCALL_DEFINE2(pipe2, int __user *, fildes, int, flags)
{
	int fd[2];
	int error;

error = do_pipe_flags(fd, flags); // Handle flags, create a file * representing the pipe
	if (!error) {
		if (copy_to_user(fildes, fd, sizeof(fd))) { // copy to userspace
			sys_close(fd[0]);		// if error – close and exit.
			sys_close(fd[1]);
			error = -EFAULT;
		}
	}
	return error;
}
```

The pipe() system call…

Creates a directory entry
Creates an inode
Creates a file
  Associates the directory entry and the inode with it

Creates 2 file * by opening it with O_RDONLY and O_WRONLY flags
  It associates a *file_operations* structure with each descriptor depending on the operations expected to be performed on it.

Installs the file *s in the file descriptor table of the process and generates 2 file descriptors (of type int)
Returns the file descriptors in the fd[2] array.

Registering file_operations

```
const struct file_operations read_pipefifo_fops = {
        .llseek = no_llseek,
        .read           = do_sync_read,
        .aio_read       = pipe_read,
        .write          = bad_pipe_w,
        .poll           = pipe_poll,
        .unlocked_ioctl = pipe_ioctl,
        .open           = pipe_read_open,
        .release        = pipe_read_release,
        .fasync         = pipe_read_fasync,
};
```

File operations are associated with each created file * when first opened
These operations tell the VFS how to perform various operations on the file.

The pipe() system call…

The file created to represent the pipe is backed by buffers managed by a *pipe_inode_info* object.

```
struct pipe_inode_info { // shown partially… : represents the pipe
	wait_queue_head_t wait; // waiting queue for blocked writers
	unsigned int readers;
	unsigned int writers;
	unsigned int waiting_writers;
	struct inode *inode;
	struct pipe_buffer bufs[PIPE_BUFFERS];  // backing data store – points to actual pages.
};
```

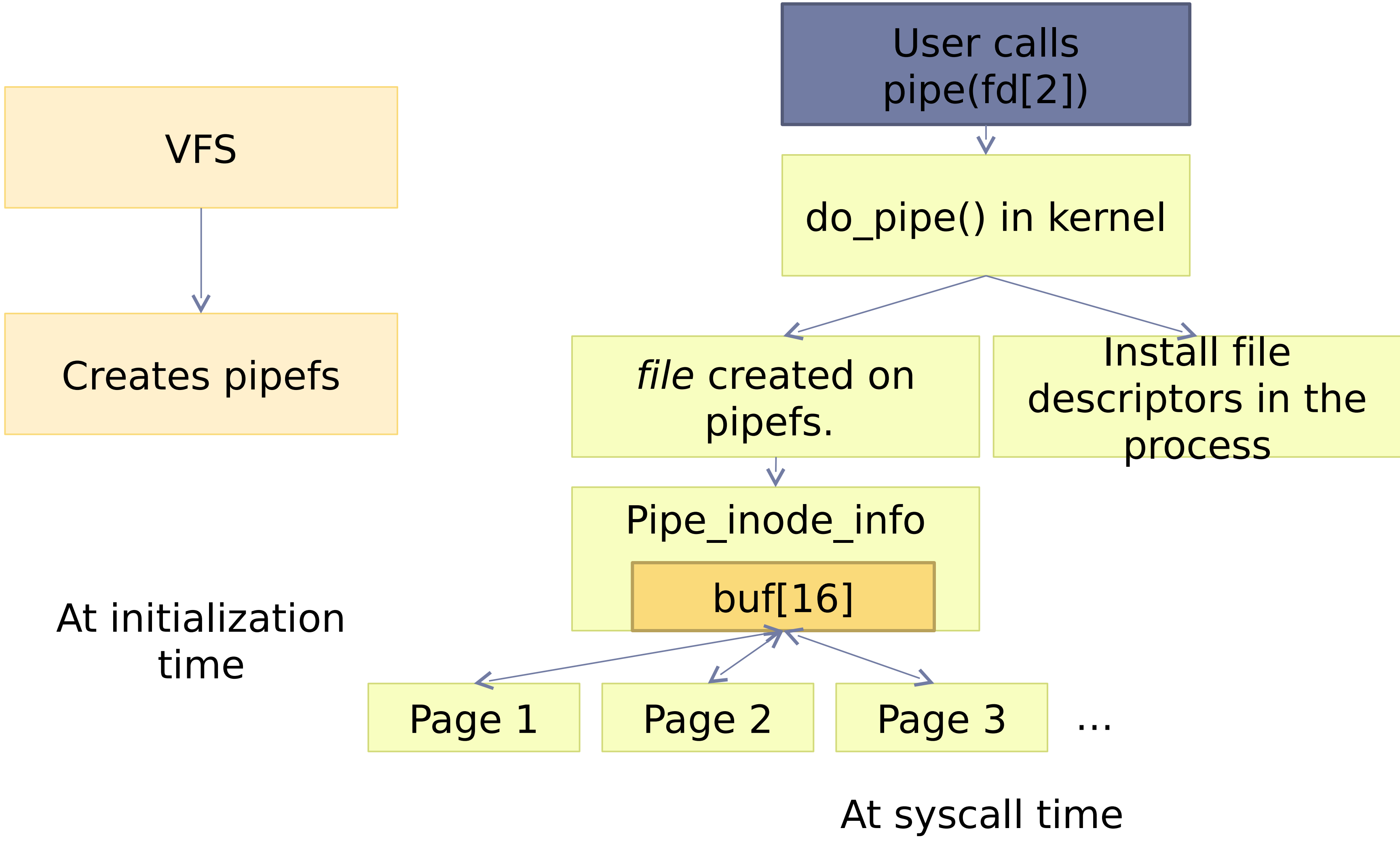
PIPE_BUFFERS == 16

Atomicity guarantees:

PIPE_BUF is the maximum limit which is guaranteed for atomic operations. PIPE_BUF = PAGE_SIZE (4096)

Atomicity is lost on page faults.

So, where are we?



Read system call

```
SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
{
	struct file *file;
	ssize_t ret = -EBADF;
	int fput_needed;

	file = fget_light(fd, &fput_needed); // get the file * associated with the file descriptor

	if (file) {
	loff_t pos = file_pos_read(file);
	ret = vfs_read(file, buf, count, &pos);
	file_pos_write(file, pos);
	fput_light(file, fput_needed);
	}

	return ret;
}
```

VFS Read

```
ssize_t vfs_read(struct file *file, char __user *buf, size_t count, loff_t *pos)
{
	// verification code

	if (ret >= 0) {
		count = ret;
		if (file->f_op->read) // The filesystem_operations
defined read function
			ret = file->f_op->read(file, buf, count, pos);
		else
			ret = do_sync_read(file, buf, count, pos); // default fallback

		// housekeeping code
	}

	return ret;
}
```

Pipe_read()

```
static ssize_t pipe_read(…) // simplified
{
      struct file *filp = …;
      struct inode *inode = …;
      mutex_lock(&inode->i_mutex); // lock the pipe inode –
concurrency protection
      for (;;) { // for each buffer
            // do all the reading from the buffers.
            // handle all the pending signals that might interrupt the system call.
            // wake up all waiting writers that there might be more room
            pipe_wait(pipe); //  if have to block – release mutex and block
      }
      mutex_unlock(&inode->i_mutex);

      // wake up waiting writers
      // if bytes read is > 0 mark file as accessed.
}
```

Pipe_read()

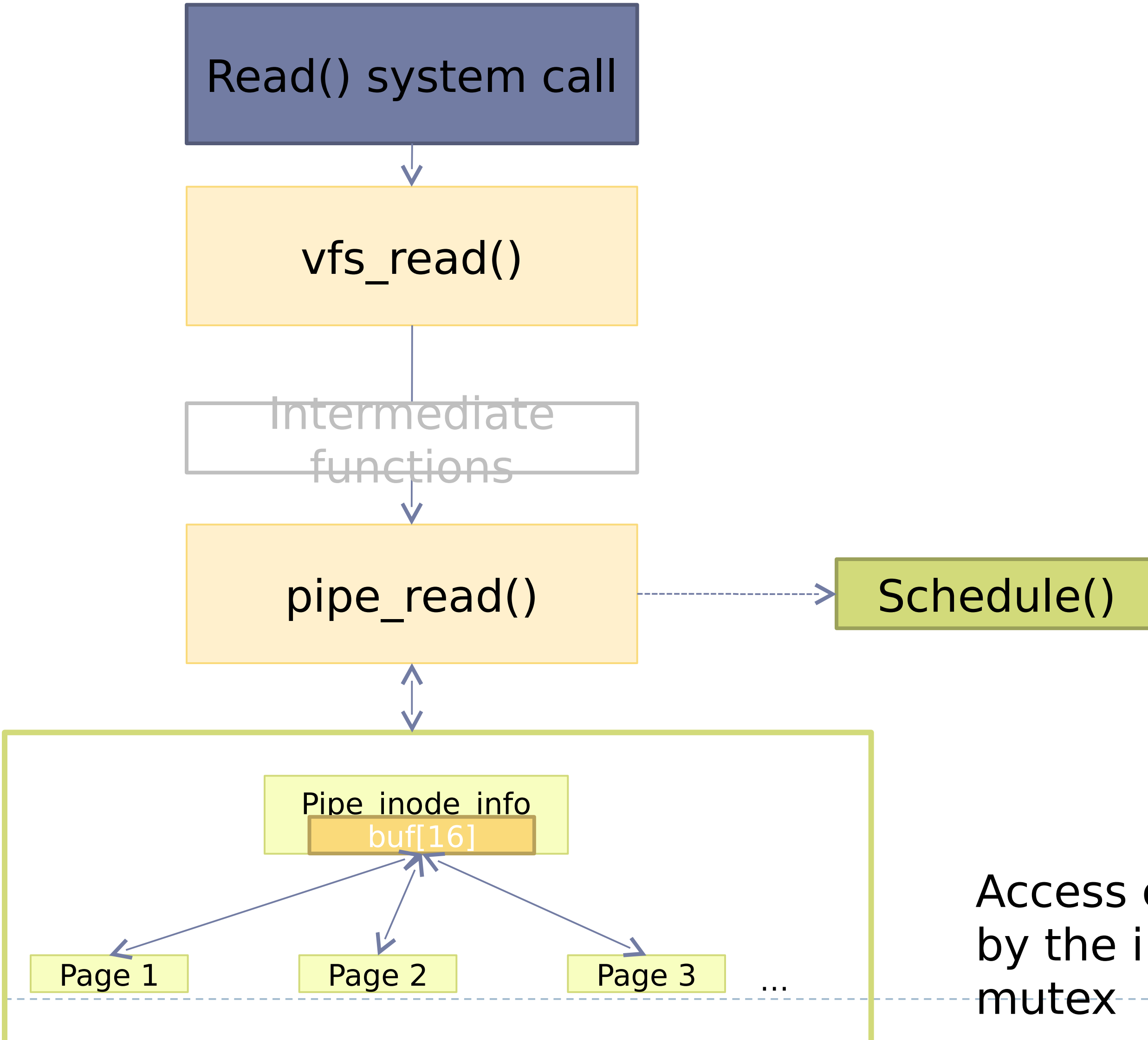
Gets the file * for reading
Locks its associated inode
Accesses the pipe buffers and reads off them

- Copies the read data to userspace in the passed char * buffer in the read() system call.

Wakes up the writers that are blocked on the wait queue
Releases the lock on the inode

In case the read blocks, it releases the mutex and calls `schedule()` which invokes the scheduler, allowing a context switch.

So where are we?



Access controlled by the inode mutex

The Write System Call

```
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
        size_t, count)
{
        struct file *file;
        ssize_t ret = -EBADF;
        int fput_needed;

        file = fget_light(fd, &fput_needed); // get the kernel file * from the (int)
fd provided by the user
        if (file) {
        loff_t pos = file_pos_read(file);
        ret = vfs_write(file, buf, count, &pos); // go to the VFS layer
        file_pos_write(file, pos);
        fput_light(file, fput_needed);
        }

        return ret;
}
```

The Write System Call…

Follows a similar code path as for the Read system call

Functions

- write()
- vfs_write()
- pipe_write()

A point to note is that all copying of data to/from userspace is done via vector operations and page faults destroy atomicity constraints.

Kernel Pages for temporary storage of data are allocated on demand upto PIPE_BUFFERS pages.

Pages are "pre-faulted" in userspace to ensure that copying data remains atomic (under certain size constraints).

FIFOs

FIFOs are created with the mkfifo() or mknod() calls

mkfifo(const char *pathname, mode_t mode) – userspace call

- mknod(…) system call
- vfs_mknod(…) for a FIFO creation
  - creates and installs a ***node*** in the global filesystem at the specified path and with the requested permissions.
  - The file node has an associated inode and directory entry just like for a pipe.

The mknod(…) system call handles creation of nodes in the filesystem of a variety of things:

- Regular Files
- Block and Character devices
- FIFOs and Sockets

FIFOs

When an open(...) system call takes place on the filename:

```
long do_sys_open(int dfd, const char __user *filename, int flags, int mode) // simplified
{
    if (...) {
        if (fd >= 0) {
            struct file *f = do_filp_open(dfd, tmp, flags, mode); // open the fifo
            if (IS_ERR(f)) {
                // ...
            } else {
                fsnotify_open(f->f_path.dentry);
                fd_install(fd, f); // install it to the file descriptor table
            }
        }
    }
    return fd;
}
```

Do_filep_open()

Does a lot of processing at the VFS level and finally calls the **.open** handler registered in the *file_operations* for the FIFO file.

Recap: Every file has an associated *file_operations* structure that specifies how the file may be manipulated by VFS.

```
struct file_operations { // simplified
    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 *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
};
```

fifo_open(…)

The f*ile_operations* .open member points to fifo_open(…)

fifo_open(…) does the following tasks:

Locks the inode of the file

Allocates a pipe_inode_info object for providing buffer space

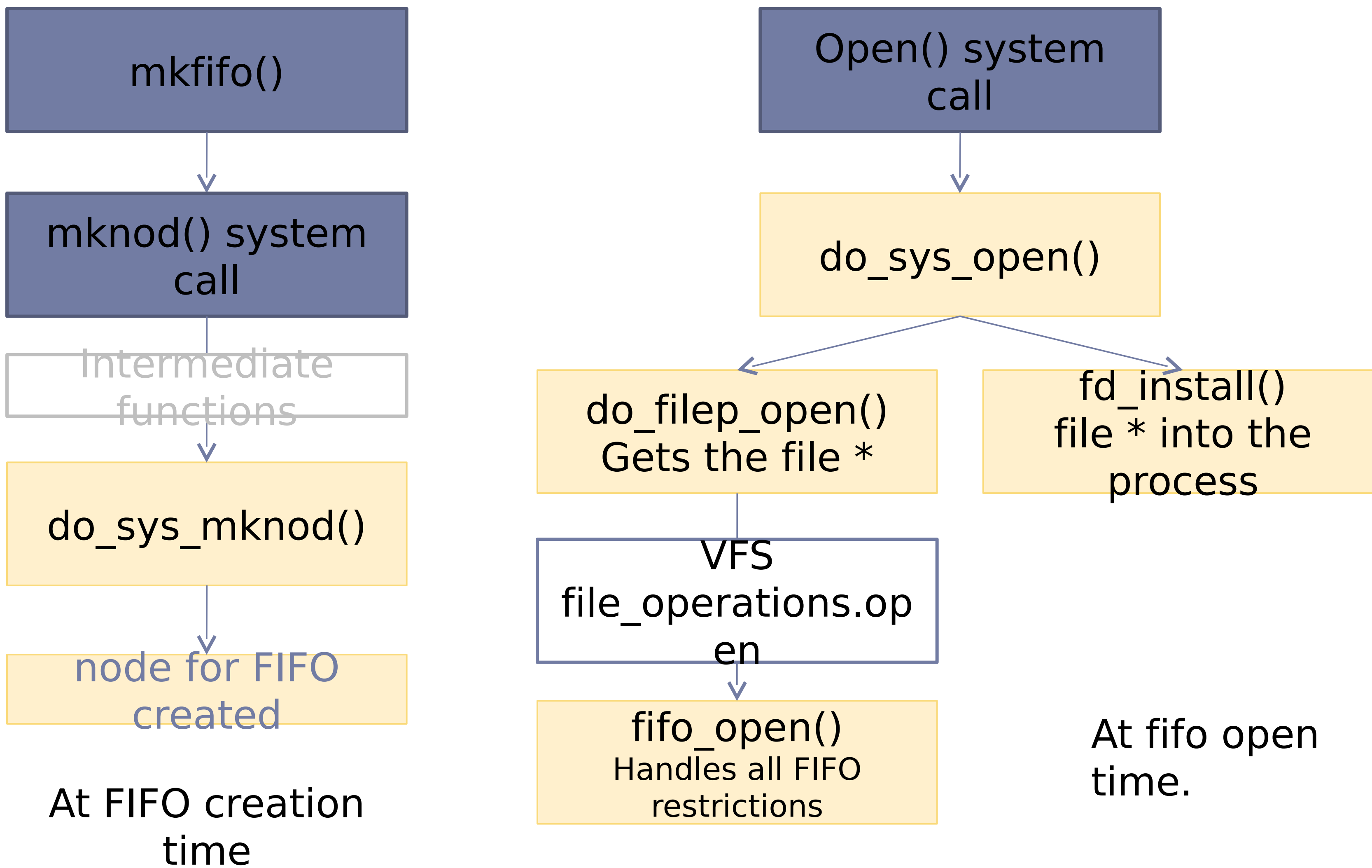
- This is the actual “pipe”

Depending on the open flags: O_RDONLY, O_WRONLY it allocates a file pointer

Increments the reference counter of the number of readers/writers on the pipe

Unlocks the inode and returns the file *

So, how did it happen again?



After open(...)

After the FIFO is successfully opened, it is equivalent to a pipe.

Reading and writing to it are governed by the pipe read/write functions.

Effect of fork()

Forking a new process simply copies the file descriptor table and its associated file * to the new process and updates the readers/writers count as appropriate within the *pipe_inode_info* object.

Effect of close()

The file descriptor is flushed

The reader/writer count for the pipe is decremented

If the count is zero, deallocate the pipe buffers and get rid of the pipe file object.

If it is a fifo, don’t get rid of the node entry, but get rid of the file object in the kernel

FIFOs require an explicit remove/unlink(…) on the file **node** for their removal from the filesystem.

Generalization to other filesystems

Other (device backed) filesystems go a bit further.

The buffers that are managed by the inodes are actually allocated from a common buffer cache.

A read or write request to a buffer that is not found in the buffer cache generates a device block IO request otherwise IO savings are made.

Dirty buffers are periodically flushed to the IO devices and the cache maintenance is done by the `bdflush` kernel daemon.

/proc

Proc is a virtual file system (procfs)

All nodes in the /proc filesystem are created from kernel space in an on-demand manner.

- Filesystem node
- File
- Direntry
- Inodes

Reads/Writes/Opens of files on this device are handled by kernel functions/modules wishing to export/import their data to/from userspace.

- These functions implement the appropriate API to communicate with user processes via /proc

Conclusion

The linux implementation of pipes is very efficient and allows for easy IPC.

FIFOs are implemented as wrappers around pipes

The /proc filesystem is created dynamically by the kernel at runtime by suitable manipulation behind the scenes.

References

The linux kernel sources documentation

linux-source-2.6.27/Documentation/filesystems/vfs.txt

The actual linux kernel source code

linux-source-2.6.27/include/linux/pipe_fs_i.h

linux-source-2.6.27/fs/open.c

linux-source-2.6.27/fs/pipe.c

linux-source-2.6.27/fs/fifo.c

A tour of the Linux VFS by Michael K. Johnson. 1996

http://www.tldp.org/LDP/khg/HyperNews/get/fs/vfstour.html

Thank You

Any Questions?