6.3

Implementation of

VFS System Calls

and

File Locking

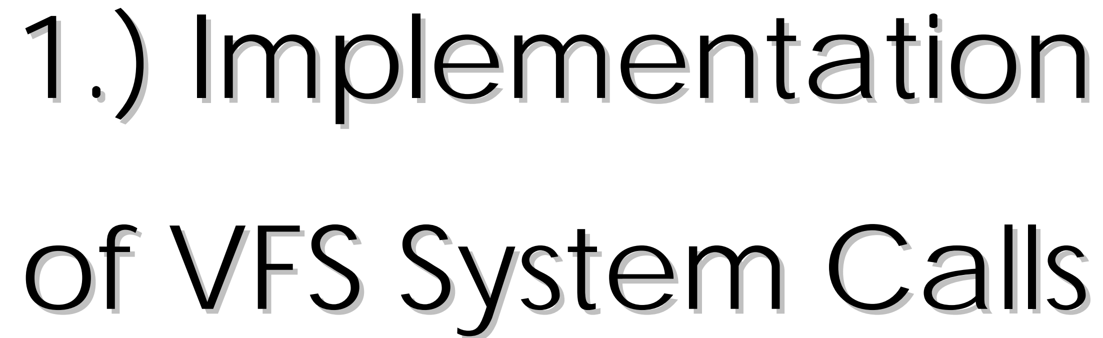
Overview

1.) Implementation of VFS System Calls

- open, read / write, close
- how to use VFS system calls

2.) File Locks:

- general concepts
- mandatory locks ⇔ advisory locks
- lock types & deadlocks
- system calls for file locking


1.) Implementation
of VFS System Calls

Implementation of VFS System Calls

- most important VFS system calls:
  open, read, write, close
- corresponding service routines:
  sys_open(), sys_read(), sys_write(), sys_close()
- further system calls:
  mount, umount, sysfs, chroot, chdir, mkdir, rmdir, readlink, symlink, chmod, dup, select, poll, fcntl, flock, etc...

The open() System Call

- usage: int open(char *file, int access [, int mode])
- example: open(“/testfile“, O_RDONLY, 0);
- parameters:
  1. file (pathname) to open
  2. flags for access type
  3. permission bitmask (for file creation)
- returns: file descriptor or -1 (error)
  (i.e. new file‘s index in the current→files→fd array)

The open() System Call

How sys_open() proceeds while execution:

1.) → getname()
2.) → get_unused_fd()
3.) → filp_open()
   3.1) → open_namei()
   3.2) → dentry_open()
4.) set current→files→fd[fd]
5.) return fd

The open() System Call

How sys_open() proceeds while execution:

1.) → getname()
2.) → get_unused_fd()
3.) → filp_open()
   3.1) → open_namei()
   3.2) → dentry_open()
4.) set current→files→fd[fd]
5.) return fd

- read file pathname from process adress space

The open() System Call

How sys_open() proceeds while execution:

1.) → getname()
2.) → get_unused_fd()
3.) → filp_open()
   3.1) → open_namei()
   3.2) → dentry_open()
4.) set current→files→fd[fd]
5.) return fd

- find an empty slot in current→files→fd
- store file descriptor locally in fd

The open() System Call

How sys_open() proceeds while execution:

1.) → getname()
2.) → get_unused_fd()
3.) → filp_open()
   3.1) → open_namei()
   3.2) → dentry_open()
4.) set current→files→fd[fd]
5.) return fd

- check access permissions to open the physical file
- return pointer to newly generated file object

The open() System Call

How sys_open() proceeds while execution:

1.) → getname()
2.) → get_unused_fd()
3.) → filp_open()
   3.1) → open_namei()
   3.2) → dentry_open()
4.) set current→files→fd[fd]
5.) return fd

- locate file‘s corresponding inode via pathname lookup or creates new one
- perform several security and access right checks

The open() System Call

How sys_open() proceeds while execution:

1.) → getname()
2.) → get_unused_fd()
3.) → filp_open()
    3.1) → open_namei()
    3.2) → dentry_open()
4.) set current→files→fd[fd]
5.) return fd

- allocate and initialize new file object
- insert the file object into the list of opened files (VFS‘s superblock→s_files)

The open() System Call

How sys_open() proceeds while execution:

1.) → getname()
2.) → get_unused_fd()
3.) → filp_open()
   3.1) → open_namei()
   3.2) → dentry_open()
4.) set current→files→fd[fd]
5.) return fd

- store file object adress returned by filp_open() in current→files→fd[fd] (fd defined in 2.)

The open() System Call

How sys_open() proceeds while execution:

1.) → getname()
2.) → get_unused_fd()
3.) → filp_open()
   3.1) → open_namei()
   3.2) → dentry_open()
4.) set current→files→fd[fd]
5.) return fd

- return file descriptor

The read() / write() System Calls

- usage: int read(int fd, char *buffer, unsigned size)
- example: read(5, buffer, 1024);
- parameters:
  1. file descriptor
  2. buffer adress
  3. number of bytes to move
- returns: number of transferred bytes, 0 (EOF) or -1 (error)

The read() / write() System Calls

How sys_read() / sys_write proceed:

1.) → fget()
2.) check file→f_mode flags for access rights
3.) → locks_verify_area()
4.) → file→f_op→{read | write}
5.) → fput()
6.) return number of transferred bytes, 0 or -1

The read() / write() System Calls

How sys_read() / sys_write proceed:

1.) → fget()
2.) check file→f_mode flags for access rights
3.) → locks_verify_area()
4.) → file→f_op→{read | write}
5.) → fput()
6.) return number of transferred bytes, 0 or -1

- derive adress of corresponding file object
  with given file descriptor
- increment file usage counter (file→f_count)

The read() / write() System Calls

How sys_read() / sys_write proceed:

1.) → fget()
2.) check file→f_mode flags for access rights
3.) → locks_verify_area()
4.) → file→f_op→{read | write}
5.) → fput()
6.) return number of transferred bytes, 0 or -1

- check whether file allows requested access

The read() / write() System Calls

How sys_read() / sys_write proceed:

1.) → fget()
2.) check file→f_mode flags for access rights
3.) → locks_verify_area()
4.) → file→f_op→{read | write}
5.) → fput()
6.) return number of transferred bytes, 0 or -1

- check for mandatory locks on accessed file region

The read() / write() System Calls

How sys_read() / sys_write proceed:

1.) → fget()
2.) check file→f_mode flags for access rights
3.) → locks_verify_area()
4.) → file→f_op→{read | write}
5.) → fput()
6.) return number of transferred bytes, 0 or -1

- call (VFS‘s) functions for transfering the data
- return number of transferred bytes
- side effect: file object‘s offset pointer is updated

The read() / write() System Calls

How sys_read() / sys_write proceed:

1.) → fget()
2.) check file→f_mode flags for access rights
3.) → locks_verify_area()
4.) → file→f_op→{read | write}
5.) → fput()
6.) return number of transferred bytes, 0 or -1

- decrement file usage counter (file→f_count)

The read() / write() System Calls

How sys_read() / sys_write proceed:

1.) → fget()
2.) check file→f_mode flags for access rights
3.) → locks_verify_area()
4.) → file→f_op→{read | write}
5.) → fput()
6.) return number of transferred bytes, 0 or -1

- return number of transferred bytes, 0 or -1
- no bytes transferred and not EOF => error

The close() System Call

- usage: int close(int fd)
- example: close(5);
- parameters: 1. file descriptor for file to be closed
- returns: 0 if closed correctly or error code

The close() System Call

How sys_close() proceeds while execution:

1.) get file object adress (current→files→fd[fd])

2.) set current→files→fd[fd] to NULL

3.) activates file object‘s flush method, if defined

4.) releases any mandatory locks

5.) call fput() to release the file object

6.) return the flush method‘s errorcode (usually 0)

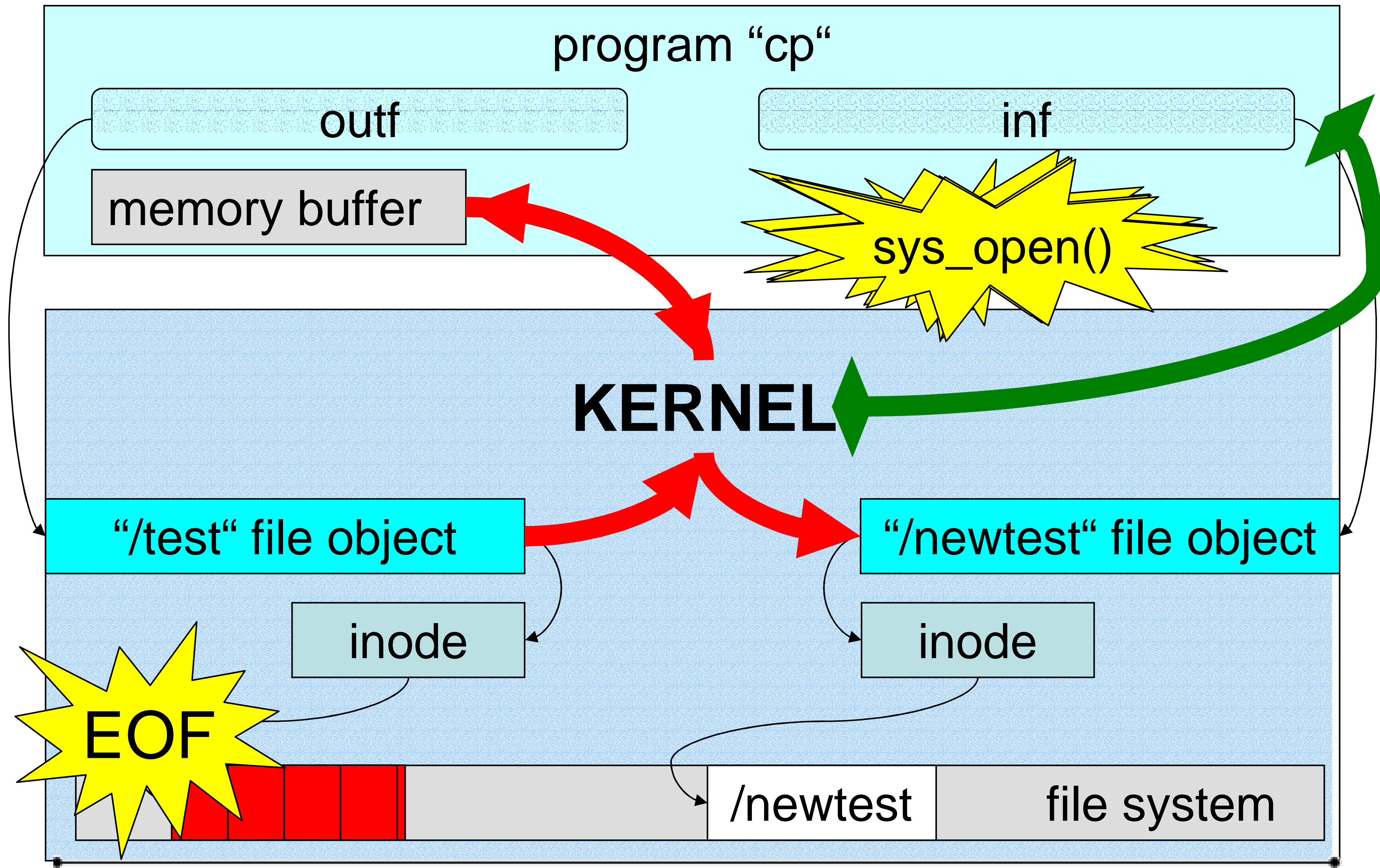
Example Using System Calls

- let‘s see how a program copying a file could work…

Example program (error checking omitted):

```
inf = open(“/test“, O_RDONLY, 0);
outf = open(“/newtest“, O_WRONLY | O_CREAT |
                    O_TRUNC, 0600);
do {
          len = read(inf, buf, 4096);
          write(outf, buf, len);
} while (len);
close(outf);
close(inf);
```

A Little Visualization…



2.) File Locking

General Concepts of File Locking

- multiple processes may hold read locks on a file region
- only one single process may set a (exclusive) write lock on a region to prevent data corruption

Typical locking hierarchy:

	Grant request for…	
Current Locks	…Read lock?	…Write lock?
No lock	Yes	Yes
Read lock	Yes	No
Write lock	No	No

Mandatory Locks ⇔ Advisory Locks

<u>advisory locks (dt. beratend) :</u>

- may be ignored by any process
- processes have to check and obey access permissions by themselves
- standard way of locking files in Linux
- only locking mechanism defined in the POSIX Standard
- caution: malicious code may lead to inconsistent data

Mandatory Locks ⇔ Advisory Locks

<u>mandatory locks (dt. obligatorisch, zwingend) :</u>

- cannot be ignored by any process
- kernel checks permissions on every file-modifying system call

→ permits violations directly

- supported by Linux but not recommended (also not in POSIX Standard)
- caution: important, blocked processes can freeze the whole system

Mandatory Locks ⇔ Advisory Locks

mandatory locks (cont.):

- to enable mandatory locking use mount option “-mand“

→ every file with set-group bit on and group-execute bit off (normally illogical) is considered to be locked

The Variety of Linux‘ Locks

- there are several lock types mainly due to compatibility reasons

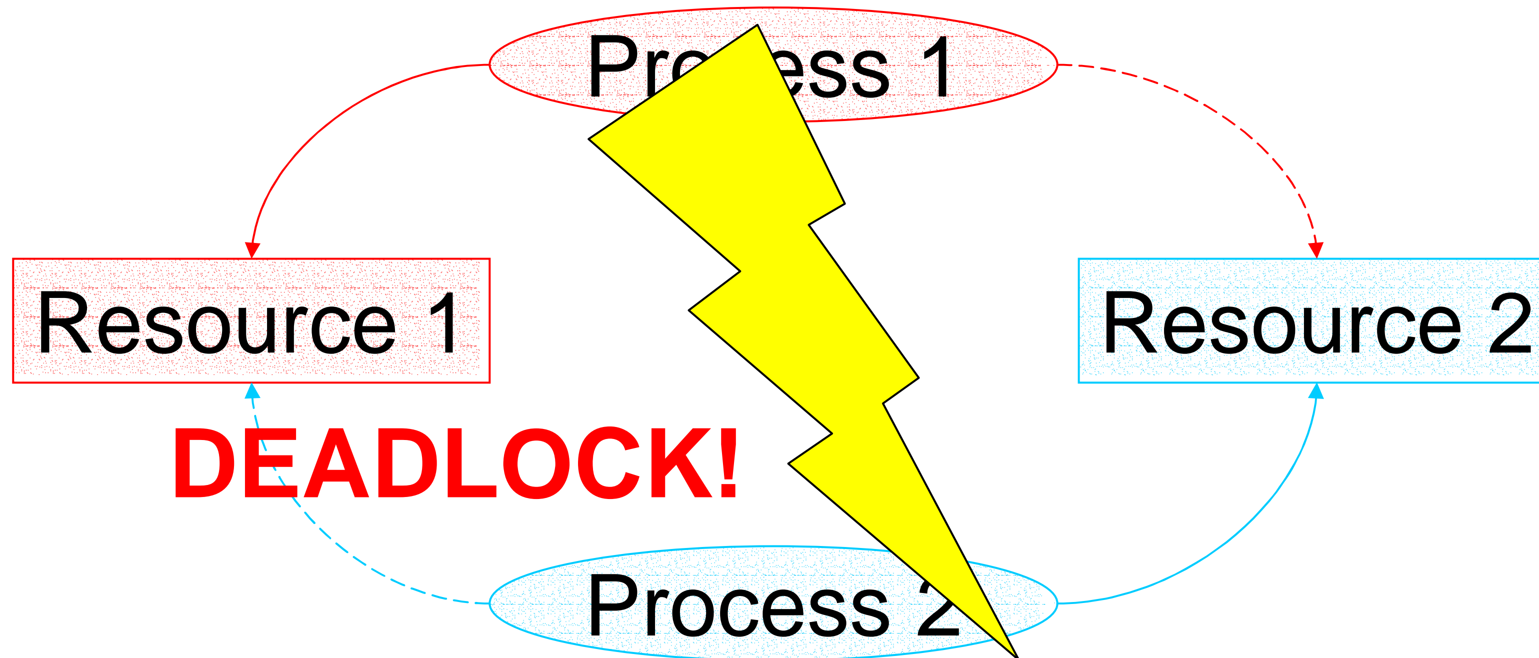
System Call	Lock Type	mandatory	advisory	whole file	region
fcntl()	FL_POSIX	✔	✔	✔	✔
flock()	FL_FLOCK	✕	✔	✔	✕
	FL_MAND	✔	✕	✔	✕
	FL_LEASE	✕	✔	✔	✕

lock types are independent from each other!

The file_lock struct

- locking information held in file_lock objects:
  - offsets of locked file regions
  - lock type and locking options
  - pointers to corresponding file object and its specific functions
  - owner‘s PID and files_struct
  - queue of blocked processes
  - …
- globally arranged in a doubly linked file_lock_list

What is a Deadlock?



process holds a lock on resource

process wants to establish a lock on resource

How to prevent deadlocks?

- implement deadlock-checking routines within the locking system calls

- deadlocks can be very complex
  => prevention might get quite expensive

- common way: lock file regions in same order

The fcntl() System Call

- usage: int fcntl(int fd, int cmd, struct flock *flp)

- arbitrary file regions can be locked
  (even single bytes)
- command determines waiting behaviour
- deadlock detection
- conform to POSIX Standard (lock type FL_POSIX)
- locking function lockf() is just a wrapper for fcntl()

The flock() System Call

- usage: int flock(int fd, int cmd)

- only whole files can be locked
- no deadlock detection
- creates locks of following types:
  FL_FLOCK, FL_MAND or FL_LEASE
- not recommended (resulting code is hardly portable)

Overview

1.) Implementation of VFS System Calls

- open, read / write, close
- how to use VFS system calls

2.) File Locks:

- general concepts
- mandatory locks ⇔ advisory locks
- lock types & deadlocks
- system calls for file locking