



# VFS - Virtual File System Switch

---

Dinh Khoa Nguyen & Eric Müller

Seminar Dateisystem

Prof. Dr. Thomas Ludwig  
Institut für Informatik  
Universität Heidelberg  
SS 2007



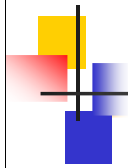
# Kapitel 1: VFS generell

---

## VFS Inhalt

- ▶ VFS General
- POSIX IO Semantik
- VFS Komponenten
- Anbindung an Dateisysteme
- Pfad Lookup
- Problemliste

- Einstiegsbeispiel
- Definition
- Implementationen des VFS
- Grobe Struktur (Common File Model)

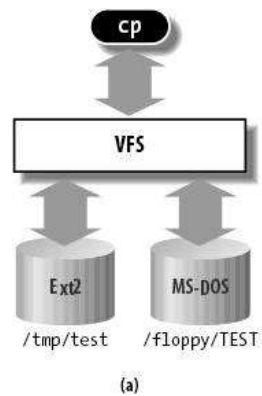


# Einstieg (Source: [Understand Linux Kernel])

VFS Inhalt

- VFS General
- POSIX IO Semantik
- VFS Komponenten
- Anbindung an Dateisysteme
- Pfad Lookup
- Problemliste

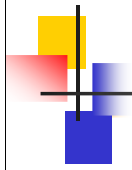
■ \$ cp /floppy/TEST /tmp/test



```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
            O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

(a)

(b)

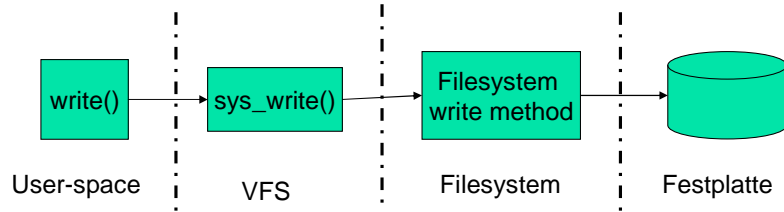


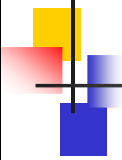
# Einstieg

VFS Inhalt

- VFS General
- POSIX IO Semantik
- VFS Komponenten
- Anbindung an Dateisysteme
- Pfad Lookup
- Problemliste

## ■ write(outf, buffer, i)





# VFS- Virtual File System Switch

VFS Inhalt

- VFS General
- POSIX IO Semantik
- VFS Komponenten
- Anbindung an Dateisysteme
- Pfad Lookup
- Problemliste

- VFS ist ein „kernel software layer“
- VFS implementiert Dateischnittstelle zwischen den Dateisystemen und Anwendungen
- VFS verarbeitet alle Systemaufrufe in Bezug auf ein Linux Dateisystem und leitet diese Aufrufe an konkrete Dateisysteme weiter.
- Interoperabilität zwischen verschiedenen Dateisystemen wird durch VFS ermöglicht  
→ Lesen von und Schreiben auf verschiedenen Dateisystemen auf verschiedenen Medien

- VFS arbeitet wie ein traditionelles Linux Dateisystem



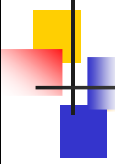
# Dateisysteme unter VFS

## VFS Inhalt

- VFS General
- POSIX IO Semantik
- VFS Komponenten
- Anbindung an Dateisysteme
- Pfad Lookup
- Problemliste

## Von VFS unterstützte Dateisysteme

- Platten-basierte Dateisysteme:
  - Linux: Ext2, Ext3, ReiserFS,..
  - Unix : SYSV,UFS,..
  - Windows: MS-DOS, VFAT, NTFS
  - CD, DVD: ISO9660, UDF,..
  - JFS, XFS,..
- Netzdateisysteme:  
NFS, Coda, AFS,..
- Spezielles Dateisysteme:  
/proc , /sys



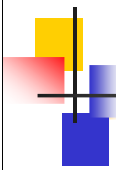
# Implementierungen des VFS

- VFS Inhalt
  - VFS General
  - POSIX IO Semantik
  - VFS Komponenten
  - Anbindung an Dateisysteme
  - Pfad Lookup
  - Problemliste
- Erstes VFS von Sun in SunOS 2.0 in 1985
  - erlaubt transparenten Zugriff auf UFS und NFS
  - andere Dateisysteme pluggable
  - ist Basis des VFS in System V Release 4 (1990)
- Andere:
  - File System Switch in System V Release 3
  - Generic File System in Ultrix
  - VFS in Linux
  - Installable File System in OS/2 und Microsoft Windows
- FUSE (File system in Userspace)
- VFS in Linux

Die Diskussion über VFS in Linux: Wann wurde VFS Linux entwickelt ?

- Im Buch „Linux File Systems“ steht, VFS für Linux kam sehr spät, erst in Version 2.4.0

- Andere Quelle: VFS war bereits bei der Version 1.0 (1994) dabei



# Struktur des VFS

---

VFS Inhalt

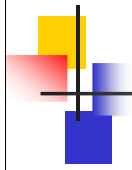
- ▶ VFS General
- POSIX IO Semantik
- VFS Komponenten
- Anbindung an Dateisysteme
- Pfad Lookup
- Problemliste

- Common File Model (CFM)
  - Ähnlich wie ein traditionelles Unix Dateimodell
  - Anderes nicht-Unix Dateisystem muss die physikalische Organization in CFM umwandeln
  - Objekt-orientiert ähnlich, aber mit C Datenstrukturen
  - Benutzt Zeiger, die auf die spezifischen Funktionen jedes Dateisystem zeigen
    - z.B : `read()` → `sys_read()` →
    - `file (Objekt)` → `f_op (pointer as a field)` →
    - `read(...)`

-Common File Model strictly mirrors the file model provided by the traditional Unix system.

-Non-Unix file system : z.B Verzeichnis ist nicht ein File, VFAT, NTFS

-Zeiger: Der Kernel ist verantwortlich dafür, einer Datei beim Öffnen die richtige Menge der Pointers zu geben und anhand der Pointers die richtigen Calls aufzurufen



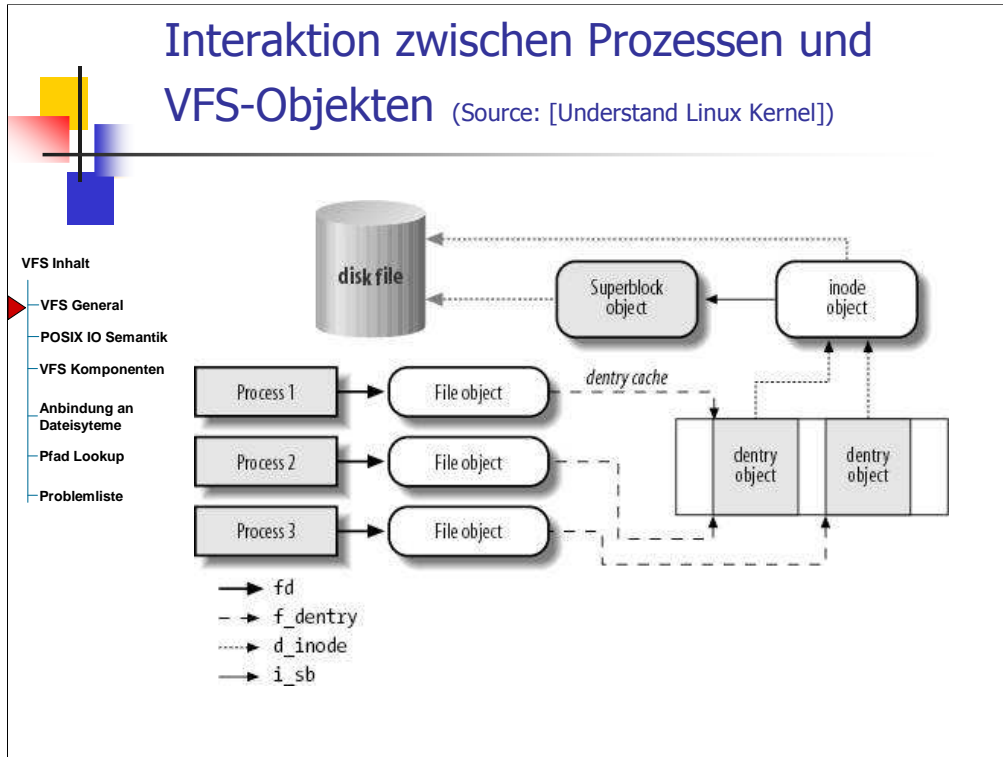
# VFS Objekte

---

## VFS Inhalt

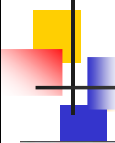
- ▶ VFS General
- POSIX IO Semantik
- VFS Komponenten
- Anbindung an Dateisysteme
- Pfad Lookup
- Problemliste

- Superblock-Objekt
- Inode-Objekt
- Datei-Objekt
- Dentry-Objekt



-Wichtige Rolle des VFS mit dentry cache : Speed-up der Transation vom Pfadname zum Inode der letzten Pfadekomponente

-Hier: 3 Prozesse greifen auf dieselbe Datei zu mit 2 hard links. Prozess 1 und 2 benutzen denselben hard link, haben jedoch jeweils ein eigenes Datei-Objekt



## Systemaufrufe auf VFS (1)

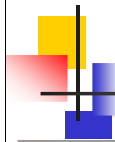
<code>mount( ) umount( ) umount2( )</code>	Mount/unmount filesystems
<code>\sysfs( )</code>	Get filesystem information
<code>statfs( ) fstatfs( ) statfs64( ) fstatfs64( )</code>	Get filesystem statistics
<code>ustat( )</code>	
<code>chroot( ) pivot_root( )</code>	Change root directory
<code>chdir( ) fchdir( ) getcwd( )</code>	Manipulate current directory
<code>mkdir( ) rmdir( )</code>	Create and destroy directories
<code>getdents( ) getdents64( ) readdir( ) link( )</code>	Manipulate directory entries
<code>unlink( ) rename( ) lookup_dcookie( )</code>	
<code>readlink( ) symlink( )</code>	Manipulate soft links
<code>chown( ) fchown( ) lchown( ) chown16( )</code>	Modify file owner
<code>fchown16( ) lchown16( )</code>	
<code>chmod( ) fchmod( ) utime( )</code>	Modify file attributes
<code>stat( ) fstat( ) lstat( ) access( ) oldstat( ) oldfstat( ) oldlstat( ) stat64( ) lstat64( )</code>	Read file status
<code>fstat64( )</code>	

-Außer verteilte und parallele Dateisysteme wie NFS sind diese Aufrufe für andere Dateisysteme unter VFS POSIX-konform

-In einigen Fällen kann VFS die Dateioperationen selbst durchführen, ohne die untere Ebene aufrufen zu müssen.

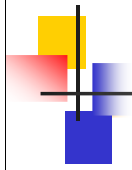
z.B close(): das Datei-Objekt einfach freigeben. Die eigentliche Datei auf Platte müssen wir nicht anfassen

lseek(): einfach das Feld „pointer“ im Datei-Objekt modifizieren



## Systemaufrufe auf VFS (2)

<code>open( ) close( ) creat( ) umask( )</code>	Open, close, and create files
<code>dup( ) dup2( ) fcntl( ) fcntl64( )</code>	Manipulate file descriptors
<code>select( ) poll( )</code>	Wait for events on a set of file descriptors
<code>truncate( ) ftruncate( ) truncate64( )</code> <code>ftruncate64( )</code>	Change file size
<code>lseek( ) _llseek( )</code>	Change file pointer
<code>read( ) write( ) readv( ) writev( ) sendfile( ) sendfile64( ) readahead( )</code>	Carry out file I/O operations
<code>io_setup( ) io_submit( ) io_getevents( ) io_cancel( ) io_destroy( )</code>	Asynchronous I/O (allows multiple outstanding read and write requests)
<code>pread64( ) pwrite64( )</code>	Seek file and access it
<code>mmap( ) mmap2( ) munmap( ) madvise( ) mincore( )</code> <code>remap_file_pages( )</code>	Handle file memory mapping
<code>fdatasync( ) fsync( ) sync( ) msync( )</code>	Synchronize file data
<code>flock( )</code>	Manipulate file lock
<code>setxattr( ) lsetxattr( ) fsetxattr( ) getxattr( ) lgetxattr( ) fgetxattr( ) listxattr( ) llistxattr( ) flistxattr( )</code> <code>removexattr( ) lremovexattr( ) fremovexattr( )</code>	Manipulate file extended attributes

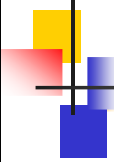


## Kapitel 2 : POSIX-I/O

### VFS Inhalt

- VFS General
- ▶ - POSIX IO Semantik
- VFS Komponenten
- Anbindung an Dateisysteme
- Pfad Lookup
- Problemliste

- POSIX-I/O-Standard
- Problem mit NFS, und deswegen auch mit VFS



# POSIX-I/O

---

VFS Inhalt

- VFS General
- ▶ - POSIX IO Semantik
- VFS Komponenten
- Anbindung an Dateisysteme
- Pfad Lookup
- Problemliste

- Was ist POSIX? [Wikipedia]
 

**POSIX (Portable Operating System Interface)** ist ein gemeinsam von der IEEE und der Open Group für Unix entwickeltes standardisiertes Applikationsebeneninterface, das die Schnittstelle zwischen Applikation und dem Betriebssystem darstellt. Der (inter-)nationale Standard trägt die Bezeichnung DIN/EN/ISO/IEC 9945
- Die POSIX-I/O Struktur, beschrieben in POSIX-Standard 1003.1,1996, spezifiziert Funktionen, die die primitiven POSIX-I/O Operationen (read, write, flock, lseek..) unterstützen
- Die meisten Linux-Distributions halten sich an den Großteil des Standards

- Außer POSIX-IO-Standard existieren auch noch viele anderen POSIX-Standards wie z.B : Threads-Interface, Real-time-Extensions, Security-Interface, POSIX Command and Utilities, POSIX Comformance Testing,...



## Beispiel mit lokalem Dateisystem

### VFS Inhalt

- VFS General

▶ - POSIX IO Semantik

- VFS Komponenten

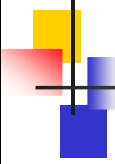
- Anbindung an  
Dateisysteme

- Pfad Lookup

- Problemliste

- (Fast) alle lokale Dateisysteme sind POSIX-konform.
- Z.B: read() & write() synchronisiert, d.h. jeder kann lesen, was die anderen gerade geschrieben haben.
- Z.B write() & write() funktioniert übers File Locking (write-lock, read-lock,..)

- Sind alle Dateisysteme POSIX-konform ??



# Problem mit NFS (VFS)

- NFS ist ZUSTANDLOS
- Read() & write() sind nicht synchronisiert bei verteilten oder parallelen Dateisysteme
- Typisches Problem beim NFS: Einer kann nur den alten Zustand einer Datei lesen obwohl die Datei von einem anderen woanders geändert wurde.
- File Locking unter NFS: Problem der Synchronisation der Clients mit dem Lock-Daemon auf dem Server
- Da verteilte und parallele Dateisysteme auch unter VFS angehängt werden können, ist VFS daher auch nicht ganz POSIX-konform
- Problem wurde gelöst im NFS 4 ?? (siehe späterer Vortrag)

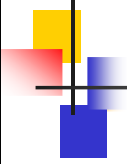
VFS Inhalt

- VFS General
- POSIX IO Semantik
- VFS Komponenten
- Anbindung an Dateisysteme
- Pfad Lookup
- Problemliste

-Problem des Lockings mit NFS : Auf NFS läuft ein Lock-Daemon und wenn ein Client eine Datei sperren will, informiert er den Daemon bescheid. Das Problem ist: was würde passieren wenn der Client abstürzt ? (File ist immer gesperrt), und wenn der Server abstürzt ? (dann denkt der Client immer dass die Datei immer noch für ihn gesperrt wird)

-Im NFS4 wurde wahrscheinlich das Problem von synchron Lesen und Schreiben gelöst, jedoch noch nicht das File Locking Problem

-Im Posix-Standard gibt es auch Standard fürs POSIX-Locking



# Kap.3 : VFS-Komponenten

- VFS Inhalt
- VFS General
- POSIX IO Semantik
- ▶ VFS Komponenten
- Anbindung an Dateisysteme
- Pfad Lookup
- Problemliste

- Superblock-Objekt
- Inode-Objekt
- Dentry-Objekt
- Datei-Objekt

- Diese Objekte haben C Datenstruktur



# Superblock-Objekt

## VFS Inhalt

VFS General

POSIX IO Semantik

VFS Komponenten

Anbindung an  
Dateisysteme

Pfad Lookup

Problemliste

- Alle Superblock-Objekte in einer Liste
- Objekt wird durch `alloc_super()` beim Mounten erzeugt.
- Repräsentiert die superblock Informationen eines Dateisystems. Das `s_fs_info` zeigt darauf.  
Z.B bei Ext2 zeigt es auf `ext2_sb_info`
- Superblock-Daten werden daher als Duplikat im Speicher gespeichert werden  
→ Kein Zugriff nötig für das Update  
→ Problem der Synchronisation → Lösung : `s_dirt` Flag
- Das Feld `s_op` zeigt auf die Operationen des Superblock-Objekts (nächste Folie)
- Jedes Dateisystem hat eigene Superblock-Operationen, die aber durch das Feld `s_op` zugegriffen werden können  
Z.B `sb` → `s_op` → `read_inode(inode)`

•Daten von `s_fs_info` ist Duplikat von Daten auf Platte → wir brauchen keinen Zugriff auf Platte um die Allocations zu aktualisieren



# Superblock-Operationen

## VFS Inhalt

— VFS General

— POSIX IO Semantik

▶ VFS Komponenten

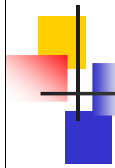
— Anbindung an  
Dateisysteme

— Pfad Lookup

— Problemliste

## ■ Einige wichtigen Operationen

- `alloc_inode(sb)`
- `destroy_inode(inode)`
- `read_inode(inode)`
- `dirty_inode(inode)`
- `delete_inode(inode)`
- `put_super(super)`
- `write_super(super)`
- `sync_fs(sb,wait)`
- `write_super_lockfs(super)`
- `unlockfs(super)`
- `statfs(super,buf)`
- `remount_fs(super,flags,data)`
- `umount_begin(super)`
- `quota_read(super,type,data,size,offset)`
- `quota_write(super,type,data,size,offset)`



# Inode-Objekt

## VFS Inhalt

VFS General

POSIX IO Semantik

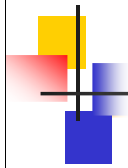
VFS Komponenten

Anbindung an  
Dateisysteme

Pfad Lookup

Problemliste

- Beinhaltet alle nötigen Informationen einer Datei
- Jede Datei kann verschiedene Dateinamen haben, jedoch nur ein Inode.
- Inode-Objekt hat `inode` Struktur und speichert als Duplikat die Inode-Daten des Dateisystems auf dem Platte.  
Z.B Anzahl der Datenblöcke
- Das Feld `i_state` :
  - `I_DIRTY_SYNC, I_DIRTY_DATASYNC, I_DIRTY_PAGES` : Inode ist „dirty“  
→ `I_DIRTY` Macro für das Update
  - `I_LOCK` : Locking
  - `I_CLEAR` : Inode-Inhalt hat keine Bedeutung mehr
  - `I_NEW`: erzeugt aber noch kein Inhalt drin
- Jedes Inode-Objekt wird in einer der folgenden 3 Listen gespeichert
  - List der „unused“ Inodes
  - List der „in-use“ Inodes
  - List der „dirty“ Inodes
- Im Superblock-Objekt gibt's auch das Feld `s_inodes`, das auf die Liste der Inodes zeigt  
Umgekehrt im Inode-Objekt gibt's `i_sb_list`, das auf das Element in der Liste zeigt
- Das Feld `i_op` zeigt auf verschiedene Inode-Operationen



# Inode-Operationen

VFS Inhalt

— VFS General

— POSIX IO Semantik

▶ VFS Komponenten

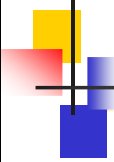
— Anbindung an  
Dateisysteme

— Pfad Lookup

— Problemliste

- create(dir,dentry,mode,nameidata)
- lookup(dir,dentry,nameidata)
- link(old\_dentry,dir,new\_dentry)
- unlink(dir,dentry)
- symlink(dir,dentry,symname)
- mkdir(dir,dentry,mode)
- rmdir(dir,dentry)
- mknod(dir,dentry,mode,rdev)
- rename(old\_dir,old\_dentry,new\_dir,new\_dentry)
- permission(inode,mask,nameidata)
- setattr(dentry,iattr)
- getattr(mnt,dentry,kstat)

.....



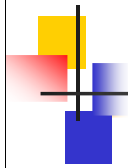
# Datei-Objekt

VFS Inhalt

- VFS General
- POSIX IO Semantik
- ▶ VFS Komponenten
- Anbindung an Dateisysteme
- Pfad Lookup
- Problemliste

- Datei-Objekt beschreibt, wie ein Prozess mit einer Datei interagiert.
- Wird erzeugt wenn die Datei geöffnet wird
- Keine Abbildung auf der Platte → Braucht kein „dirty“ Feld
- Wichtige Information: Dateizeiger (Feld `f_pos`) speichert die aktuelle Position.
- Alle Datei-Objekte sind in einem „slap cache“ `filp` gespeichert. Die Variable `files_stat` spezifiziert maximale Anzahl der Dateien im Cache
- Das Feld „`f_count`“ informiert wie viele Prozesse diese Datei momentan benutzen.
- Superblock-Objekt hat auch das Feld `s_files` zeigt auf eine Liste der „in-use“ Dateien.
- VFS öffnet eine Datei wie folgendes:
  - `get_empty_filp()` invokiert, um das Datei-Objekt zu allokiieren
  - Diese Funktion invokiert weiter `kmem_cache_alloc()`, um ein freies Objekt vom `filp` Cache zu erhalten
  - Initialisiere die Felde des Datei-Objekts
- Das Feld `f_op` zeigt auf die Datei-Operationen.

- `f_op` wurde vom `i_fop` initialisiert



# Datei-Operationen

VFS Inhalt

— VFS General

— POSIX IO Semantik

▶ VFS Komponenten

— Anbindung an  
Dateisysteme

— Pfad Lookup

— Problemliste

- llseek(file,offset,origin)
- read(file,buf,count,offset)
- aio\_read(reg,buf,len,pos)
- write(file,buf,count,offset)
- aio\_write(reg,buf,len,pos)
- open(inode, file)
- fsync(file,dentry,flag)
- release(inode,file)
- lock(file,cmd,file\_lock)

.....



# Dentry-Objekt

## VFS Inhalt

VFS General

POSIX IO Semantik

VFS Komponenten

Anbindung an  
Dateisysteme

Pfad Lookup

Problemliste

- Spezifiziert die Verknüpfung von einer Datei zu einem Verzeichnis
- Jede Komponente vom Pfadname einer Datei wird zu einem Dentry-Objekt umgesetzt
- Keine Abbildung auf Platte → braucht kein „dirty“ Feld
- Dentry-Objekte werden in einem slap-Cache `dentry_cache` gespeichert.  
erzeugt mit `kmem_cache_alloc()`  
gereinigt mit `kmem_cache_free()`
- 4 Zustände der Dentry-Objekte
  - free
  - unused
  - in use
  - negative
- Das Feld `d_op` zeigt auf die Operationen



# Dentry-Cache

VFS Inhalt

— VFS General

— POSIX IO Semantik

▶ VFS Komponenten

— Anbindung an  
Dateisysteme

— Pfad Lookup

— Problemliste

- Problem: Lesen und Konstruieren ein Dentry-Objekt vom Platte langsam  
→ halte im Speicher die dentry-Objekte
- Dentry-Cache enthält
  - Eine Liste der Dentry-Objekte in 4 Zuständen
  - Eine Hash-Liste für schnelles Finden der Dentry-Objekte einer Datei und ihres darüberliegenden Verzeichnisses
- Inode-Cache in RAM: hält Inode-Objekte der „unused“ Dentry-Objekte im Speicher

# Datei & Prozess

VFS Inhalt

VFS General

POSIX IO Semantik

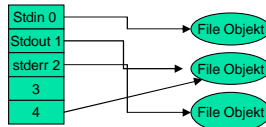
VFS Komponenten

Anbindung an Dateisysteme

Pfad Lookup

Problemliste

- Jeder Prozess hat ein Zeiger `fs` auf eine `fs_struct` Struktur für die Interaktionen mit einem Dateisystem. Einige wichtigen Felder
  - `struct dentry* root`
  - `struct dentry* pwd`
  - `atomic_t count`
  - `rwlock_t lock`
- Ein anderer Zeiger `files` auf die `files_struct` Struktur spezifiziert welche Dateien momentan von diesem Prozess geöffnet werden
  - `int max_fds`
  - `struct file** fd`
  - `fd_set * open_fds`
  - `fd_set open_fds_init`
  - `struct file*[] fd_array`
- Für jede Datei im Array `fd` ist die Arrayindex der Datei-Deskriptor



- Prozess greift auf das File Objekt via Datei-Deskriptor zu  
`current → files → fd[fd]`

- Bei der Initialisierung zeigt `fd` (Zeiger auf Datei-Objekte) auf `fd_array` und `open_fds` (Zeiger auf Dateibeschreibung) auf `open_fds_init`
- Die Anzahl der Datei-Deskriptors ist begrenzt. Jedoch kann der Kernel erweitern wenn nötig.
- Das Feld `fd` hat als Standard
  - 1. Datei-Deskriptor ist `stdin`
  - 2. Datei-Deskriptor ist `stdout`
  - 3. Datei-Deskriptor ist `stderr`
- 2 Indexe von `fd` können auf dasselbe Objekt zeigen