

# Kapitel 5: Elementare I/O-Funktionen

## 5: Elementare I/O-Funktionen

- ▶ Bei den **elementare I/O-Funktionen** handelt es sich um **Systemcalls**.
- ▶ Im Gegensatz zu Standard-IO-Funktionen findet keine Pufferung statt.
- ▶ Elementare I/O-Funktionen erlauben das Öffnen, Lesen, Schreiben und Schließen von Dateien.
- ▶ Dies ist unter Unix extrem wichtig, da dort das Prinzip *Alles ist eine Datei* gilt.
- ▶ Die vorgestellten Systemcalls sind Teil des POSIX-Standards.

# Agenda

- ▶ Elementare I/O-Funktionen
  - ▶ Datei öffnen
  - ▶ Datei erzeugen
  - ▶ Datei schließen
  - ▶ Datei lesen
  - ▶ Datei schreiben
  - ▶ Datei löschen
- ▶ Kernel-Datenstrukturen für offene Dateien
- ▶ **File Sharing**: Gemeinsame Nutzung von Dateien durch verschiedene Prozesse und die damit verbundenen Probleme

# Das Öffnen einer Datei

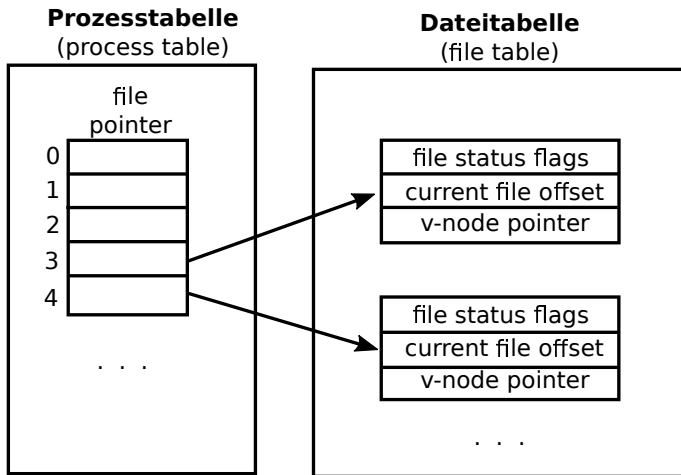
Was passiert, wenn der Prozess **P** die reguläre Datei **F** öffnet?

1. Der Kernel überprüft ob **F** existiert.  
Falls nicht, wird diese gegebenenfalls angelegt.
  2. Der Kernel verifiziert ob **P** über die benötigten Zugriffsrechte verfügt.
  3. Der Kernel erstellt für **P** ein Handle (Ganzzahl) **D**.
  4. Der Kernel erstellt für **F** einen Kontext.
  5. Der Kernel verknüpft **D** mit diesem Kontext.
  6. Der Kernel verknüpft **P** mit **D**.
  7. Der Kernel teilt **P** den Datei-Handle (**Dateideskriptor**) **D** mit.
- P** kann nun mittels **D** auf **F** zugreifen.

# Dateideskriptor

- ▶ Ein Dateideskriptor ist ein ganzzahliger Wert.
- ▶ Der Kernel verwaltet für jeden Prozess eine *Tabelle* mit *geöffneten* Dateien. Der Dateideskriptor entspricht der *Spaltennummer*.
- ▶ Der Tabelleneintrag ist ein Zeiger auf einen Dateikontext.
- ▶ Jedesmal, wenn ein Prozess eine Datei öffnet, wird ein Eintrag zu seiner Dateideskriptor-Tabelle hinzugefügt.
- ▶ Prozesse können über Systemcalls auf ihre geöffneten Dateien zugreifen.
- ▶ Ein Prozess hat **keinen** Zugriff auf die *Tabellen* der anderen Prozesse (Isolation).

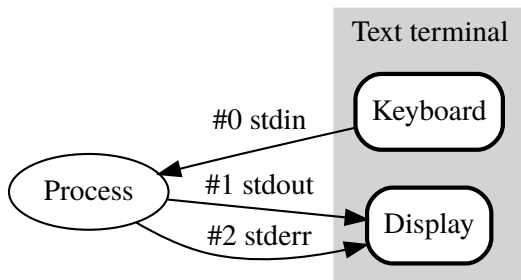
# Dateideskriptor: Illustration



# Default-Dateideskriptoren

Beim Erzeugen eines Prozesses werden diesem automatisch drei Dateideskriptoren (**<unistd.h>**) bereitgestellt:

```
#define STDIN_FILENO 0 // Standardeingabe  
#define STDOUT_FILENO 1 // Standardausgabe  
#define STDERR_FILENO 2 // Standardfehlerausgabe
```



By Danielpr85 based on Graphviz source of TuukkaH - Own work, Public I

# 5.1: Öffnen und Schließen von Dateien

## Close - Schließen von Dateien

```
#include <unistd.h>

int close(int fd);
```

- ▶ **close()** schließt einen Dateideskriptor, so dass dieser nicht mehr zu einer Datei gehört und wiederverwendet werden kann.
- ▶ **close()** gibt bei Erfolg **0** zurück, ansonsten **-1**.
- ▶ Ein erfolgreicher Aufruf von **close()** garantiert nicht, dass die Daten erfolgreich auf der Festplatte gespeichert wurden, da der Kernel verzögert schreibt (*caching*).



# Open – Öffnen von Dateien

```
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

- ▶ Im Erfolgsfall gibt **open()** einen Dateideskriptor zurück. Dabei handelt es sich um die kleinste noch nicht vergebene (nicht-negative) Nummer. Im Fehlerfall ist der Rückgabewert **-1** (→ Tafel)
- ▶ Der Parameter **flags** bestimmt unter anderem den Zugriffsmodus (Lesen, Schreiben, Lesen und Schreiben) und das Caching-Verhalten des Kernels
- ▶ Mittels dem optionalen dritten Argument **mode** können die Zugriffsrechte einer neu angelegten Datei bestimmt werden. Das Argument wird daher auch nur ausgewertet, falls eine neue Datei angelegt wird
- ▶ Gültige Optionen für **flags** und **mode** sind in **<fcntl.h>** als symbolische Konstanten definiert

# Open – Zugriffsmodus

```
int open(const char *pathname, int flags, mode_t mode);
```

Als Wert für das Argument **flags** muss genau eine der folgenden drei Konstanten, welche den Zugriffsmodus festlegen, angegeben werden:

- ▶ **O\_RDONLY**: Datei zum Lesen öffnen
- ▶ **O\_WRONLY**: Datei zum Schreiben öffnen
- ▶ **O\_RDWR**: Datei zum Lesen und Schreiben öffnen

**Beispiel:** Die Datei `test.txt` zum Lesen öffnen.

```
int fd = open("test.txt", O_RDONLY);
```

# Open – Ausgewählte Optionen

```
int open(const char *pathname, int flags);
```

Mit Hilfe des Arguments **flags** kann nicht nur der Dateizugriff festgelegt werden. Die folgenden Optionen können mit dem bitweisen Oder-Operator **|** ausgewählt werden:

- ▶ **O\_APPEND**: Datei *zum Schreiben am Ende* öffnen
- ▶ **O\_TRUNC**: Eine zum Schreiben geöffnete Datei wird geleert
- ▶ **O\_SYNC**: Die Funktion **write** blockiert solange bis der Schreibvorgang aus Sicht des Kernels abgeschlossen ist
- ▶ **O\_DIRECTORY**: Aufruf schlägt fehl falls **pathname** kein existierendes Verzeichnis ist

## Beispiel:

```
int fd = open("wichtig.txt", O_RDWR | O_APPEND | O_SYNC);
```

# Open – Neue Dateien erstellen

Bei den folgenden Optionen muss das optionale Argument **mode** mit angegeben werden:

- ▶ **O\_CREAT**: Datei wird neu angelegt falls Sie nicht existiert
- ▶ **O\_EXCL**: Zusammen mit **O\_CREAT** angegeben, wird die Datei nicht geöffnet falls diese bereits existiert; in diesem Fall liefert **open -1** zurück
- ▶ Vergabe von Zugriffsrechten:
  - ▶ **S\_IRUSR, S\_IRGRP, S\_IROTH**  
Leserecht für Eigentümer, Gruppe und Andere
  - ▶ **S\_IWUSR, S\_IWGRP, S\_IWOTH**  
Schreibrecht für Eigentümer, Gruppe und Andere
  - ▶ **S\_IXUSR, S\_IXGRP, S\_IXOTH**  
Ausführungsrecht für Eigentümer, Gruppe und Andere
  - ▶ **S\_IRWXU, S\_IRWXG, S\_IRWXO**  
Lese-, Schreib- und Ausführungsrecht für Eigentümer, Gruppe und Andere

# Beispiel: Neue Dateien erstellen

```
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <stdio.h>

int main(int args, char *argv[]) {
    const int flags = O_RDONLY | O_CREAT | O_EXCL;
    const int mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
    int fd;

    if(args != 2) {
        fprintf(stderr, "usage: _mkfile_<file>\n");
        exit(EXIT_FAILURE);
    }

    if ((fd = open(argv[1], flags, mode)) < 0 ) {
        perror(argv[1]);
        exit(EXIT_FAILURE);
    }
    close(fd);
    return EXIT_SUCCESS;
}
```

# Zugriffsrechte - Spezielle Zugriffsrechte

Es gibt noch drei weitere Spezialzugriffsrechte.

- ▶ **S\_ISUID**: set-user ID Bit  
Das Programm läuft mit den Rechten des Eigentümers
- ▶ **S\_ISGID**: set-group ID Bit  
Das Programm läuft mit den Rechten der Gruppe
- ▶ **S\_ISVTX**: sticky Bit (saved-text Bit)  
Das Bit hat unterschiedliche Auswirkungen für Dateien und Verzeichnisse
  - ▶ **Dateien**: Nach Beendigung des Prozesses wird das Programm nicht aus dem Arbeitsspeicher entfernt. Das Recht ist längst überholt, da aktuelle OSe bessere Caching-Konzepte haben.
  - ▶ **Verzeichnisse**: Einschränkung der Zugriffsrechte auf Dateien innerhalb des Verzeichnisses. Nur noch der Eigentümer einer Datei (bzw. des Verzeichnisses) darf diese löschen oder umbenennen.  
Beispiel: `/tmp`

# Datei unter Windows Öffnen

```
#include <windows.h>

HANDLE WINAPI CreateFile(
    _In_      LPCTSTR          lpFileName,
    _In_      DWORD            dwDesiredAccess,
    _In_      DWORD            dwShareMode,
    _In_opt_  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    _In_      DWORD            dwCreationDisposition,
    _In_      DWORD            dwFlagsAndAttributes,
    _In_opt_  HANDLE           hTemplateFile
);
```

Quelle: <https://msdn.microsoft.com/en-us/library/windows/desktop/aa363858%28v=vs.85%29.aspx>

## 5.2: Zugriffskontrolle

### Zugriffskontrolle

Satz von Regeln die festlegen welche **Subjekte** mit welchen **Rechten** auf welche **Objekte** zugreifen dürfen.

### Realisierungsansätze

- ▶ Whitelist
  - ▶ Regelsatz der definiert was **erlaubt** ist.
  - ▶ Verboten ist alles, was nicht explizit erlaubt ist.
- ▶ Blacklist:
  - ▶ Regelsatz der definiert was **verboten** ist.
  - ▶ Erlaubt ist alles, was nicht explizit verboten ist.
- ▶ Hybride Listen.



# Zugriffskontrollmatrix

Eine Zugriffskontrollmatrix  $A$  ist eine zweidimensionale  $|S| \times |O|$  Whitelist.

$$A = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1|O|} \\ a_{21} & a_{22} & \dots & a_{2|O|} \\ \vdots & \vdots & \vdots & \vdots \\ a_{|S|1} & a_{22} & \dots & a_{|S||O|} \end{pmatrix}$$

Bei den einzelnen Einträgen  $a_{ij}$  handelt es sich um Teilmengen von  $R$ , d.h. es gilt:  $a_{ij} \in R$ .

# Beispiel: Zugriffskontrollmatrix

- ▶  $S = \{\text{Alice, Bob, Gast}\}$
- ▶  $O = \{\text{Alice-Tagebuch, Einkaufsliste, Klingel}\}$
- ▶  $R = \{\text{lesen, schreiben}\}$

**Aufgabe:** Erstellen Sie eine sinnvolle Zugriffskontrollmatrix.

# Zugriffskontrolllisten

## Zugriffskontrolllisten (“Access Control List”, ACL)

Zu jedem Objekt  $o \in O$  existiert eine Liste von Paaren  $(s, r) \in S \times R$ .

### Aufgabe

- ▶  $S = \{\text{Alice, Bob, Gast}\}$
- ▶  $O = \{\text{Alice-Tagebuch, Einkaufsliste, Klingel}\}$
- ▶  $R = \{\text{lesen, schreiben}\}$

**Aufgabe:** Erstellen Sie eine sinnvolle ACL.

# UNIX Zugriffskontrolle

- ▶ Jedes *Objekt* (Datei, Verzeichnis, Socket, Symbolischer Link, Gerät (Block oder Character), FIFO) ist **genau einem Eigentümer** (-*Subjekt*) und **genau einer Gruppe** (von *Subjekten*) zugeordnet.
- ▶ Für den Eigentümer, für die Gruppe und für alle anderen Benutzer (die "Welt") kann man jeweils getrennt festlegen, ob sie die *Methoden* read, write und execute ausführen dürfen.

## Beispiel:

```
$ ls -lah sysprog03.pdf
-rw-r--r-- 1 cforler dozent 161K Mär 16 14:50 sysprog03.pdf
```

# Zugriffsrechte Verstehen

```
$ ls -lah foo.txt  
-rw-r--r-- 1 cforler dozent 161K Mär 16 14:50 foo.txt
```

## Bedeutung der Spaltenbasierten-Ausgabe von `ls`

- ▶ 1. Spalte: Typ und Zugriffsrechte
  - ▶ Das erste Zeichen steht für den Dateitypen (-: **Datei**; b: Block device; c: Character device; d: **Verzeichnis**; l symbolischer Link; p: named pipe (FIFO); s: Socket)
  - ▶ Die nächsten drei Zeichen: Zugriffsrechte des Eigentümers (**u**)
  - ▶ Die nächsten drei Zeichen: Zugriffsrechte der Gruppe (**g**)
  - ▶ Die nächsten drei Zeichen: Zugriffsrechte aller Anderen (**o**)
- ▶ 2. Spalte Anzahl Links
- ▶ 3. und 4. Spalte: Name und Gruppe des Eigentümers
- ▶ 5. Spalte: Dateigröße
- ▶ 6-8. Spalte: Zeitstempel des letzten Schreibzugriffes
- ▶ 9. Spalte Dateiname

# Zugriffsrechte ändern mittels `chmod`

Mit dem Tool `chmod` lassen sich die Zugriffsrechte ändern

## Allen Schreibrechte entziehen

```
$ chmod -w foo.txt
$ ls -lah foo.txt
-r--r--r-- 1 cforler cforler 162K Mär 16 14:55 foo.txt
```

## Den Anderen Leserechte entziehen

```
$ chmod o-r foo.txt
$ ls -lah foo.txt
-r--r----- 1 cforler cforler 163K Mär 16 14:57 foo.txt
```

## Der Gruppe Schreibrechte erteilen

```
$ chmod g+w foo.txt
$ ls -lah foo.txt
-r--rw---- 1 cforler cforler 163K Mär 16 14:57 foo.txt
```

# Zugriffsrechte als dreistellige Oktalzahl

- ▶ Jedem Zugriffsrecht wird eine Zahl (Zweierpotenz) zugeordnet
  - ▶ **Lesen (read):** 4
  - ▶ **Schreiben (write):** 2
  - ▶ **Ausführen (execute):** 1
- ▶ Jede Ziffer der Oktalzahl ist die Summe der Rechte.
- ▶ **Erste Ziffer:** Rechte des Eigentümers.
- ▶ **Zweite Ziffer:** Rechte der Gruppe.
- ▶ **Dritte Ziffer:** Rechte der Anderen.

```
$ chmod 660 sysprog03.pdf
```

**Frage:** Was bewirkt das obige Kommando?

# Execute or Write

- ▶ In der Regel sollte man **entweder** Rechte zum ausführen
- ▶ **oder** Rechte zum schreiben verteilen.
- ▶ **Achtung:** Lesen erlaubt das kopieren von Dateien.

## Merke: Lesen ermöglicht ausführen

- ▶ Lesen erlaubt kopieren.
- ▶ Kopierer ist Eigentümer der kopierten Datei.
- ▶ Kopierer kann kopierte Datei die Rechte zum ausführen geben.
- ▶ Das ist uns nicht 1337 genug. Es geht noch besser. :-)
- ▶ Dynamisch gelinkte Programme können auch direkt von einem dynamischen *Loader* (`ld.so`) ausgeführt werden.
- ▶ Bsp: `$ /lib64/ld-linux-x86-64.so.2 /bin/ls`



# Das SUID Bit

- ▶ Normalerweise läuft ein Programm mit den Rechten des Aufrufers.
- ▶ Der Eigentümer kann eine ausführbaren Datei `foo` mittels `chmod u+s foo` als `suid` markieren ("set user id").
- ▶ Ist das `suid`-Bit gesetzt, dann läuft das Programm mit den Rechten des **Eigentümers**.

Analog: `sgid` ("set group id").

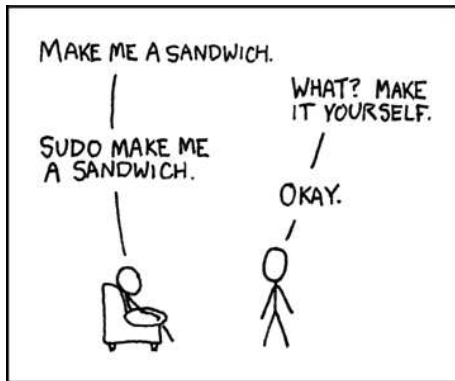
# Sudo

- ▶ Das Programm `sudo` ermöglicht es regulären Benutzer Programm als `root` auszuführen.
- ▶ `root` ist unter UNIX einen Super-User Account (`root`) dessen Zugriffsrechte praktisch unbeschränkt sind (unter Windows entspricht dies dem Account `system`).

## Beispiel:

```
$ whoami
cforler
$ ls -lah /usr/bin/sudo
-rwsr-xr-x 1 root root 126K Feb 23  2015 /usr/bin/sudo
$ sudo whoami
root
```

# Das Allgegenwärtige Sudo



Quelle: <https://xkcd.com/149/>

# Anmerkungen

- ▶ Das `suid/guid` Verfahren erlaubt es, präzise Zugriffsstrukturen festzulegen. Ein Benutzer **s** kann sogar das eigene Recht, eine Methode auf ein Objekt **o** anwenden zu können, auch anderen gewähren – ohne dass **s** selbst Eigentümer von **o** ist.
- ▶ Leider ist diese Verfahren auch kompliziert und fehlerträchtig. In der Praxis neigen viele Programmierer dazu, ausführbaren Dateien zu weitreichende Privilegien einzuräumen ("`suid root` – alles wird gut" : – ). Dies ist die Ursache für viele UNIX-Sicherheitslücken.
- ▶ **Kompliziertheit und Sicherheit sind natürliche Feinde!**
- ▶ Weitere Infos gibt es unter <https://de.wikipedia.org/wiki/Unix-Dateirechte>

# Änderung von Zugriffsrechten

```
#include <sys/stat.h>

int chmod(const char *pathname, mode_t mode);
int fchmod(int fd, mode_t mode);
```

- ▶ Bei Erfolg wird **0** zurück gegeben, ansonsten **-1**
- ▶ **chmod()**: Setzt die Zugriffsrechte der in `pathname` angegebenen Datei auf `mode`
- ▶ **fchmod()**: Setzt die Zugriffsrechte der Datei mit dem Dateideskriptor `fd` auf `mode`
- ▶ **mode**: Eine oder mehrere durch `|` verknüpfte Zugriffsrechte  
**Beispiel:** `S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP`

# Beispiel: Änderung von Zugriffsrechten

```
1  #include <sys/stat.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main(int args, char *argv[]) {
6      mode_t mode = S_IRUSR
7          | S_IWUSR
8          | S_IRGRP
9          | S_IWGRP
10         | S_IROTH;
11
12     for(int i=1; i <args; i++) {
13         if (chmod(argv[i], mode)) {
14             perror(argv[i]);
15         } else {
16             printf("%s: _permission_are_changed_to_%o\n",
17                 argv[i], mode);
18         }
19     }
20     exit(EXIT_SUCCESS);
21 }
```

# Effektive und Reale Benutzer ID

Bei POSIX wird zwischen der **realen** und der **effektiven** Benutzer ID unterschieden.

- ▶ **reale Benutzer ID**

Die ID des Benutzers, welcher den Prozess gestartet hat.

- ▶ **effektive Benutzer ID**

Die Benutzer ID unter welcher der Prozess läuft.

- ▶ Lässt sich mit dem Systemcall `setuid` anpassen.
- ▶ Ist normalerweise identisch mit der realen Benutzer ID.
- ▶ Wurde bei einer ausführbaren Datei das `suid`-Bit gesetzt (`# chmod +s <file>`), dann läuft es mit der effektiven Benutzer ID des Datei-Eigentümers.

**Frage:** Was ist der Sinn einer effektiven Benutzer ID?

# Zugriffsüberprüfung beim Start eines Programms

**Zugriffsüberprüfung:**

```
if effektive UID == 0 then  
    return Zugriff erlaubt  
end if
```

```
if effektive UID == UID der Datei then  
    return Benutzer-Zugriffsrechte  
end if
```

```
if effektive GID == GID der Datei then  
    return Gruppen-Zugriffsrechte  
end if
```

```
return Andere-Zugriffsrechte
```



# Zugriffsrechte Prüfen

```
#include <unistd.h>

int access(const char *pathname, int mode);
```

- ▶ Bei Erfolg wird **0** zurückgegeben, ansonsten **-1**
- ▶ Prüft, ob der Prozess auf die Datei **pathname** zugreifen kann
- ▶ Falls es sich bei **pathname** um ein symbolischer Link handelt, werden die Zugriffsrechte der referenzierten Datei geprüft
- ▶ **mode** legt fest, welche Zugriffsprüfungen durchgeführt werden sollen:
  - ▶ **R\_OK**: Leserechte prüfen.
  - ▶ **W\_OK**: Schreibrechte prüfen.
  - ▶ **X\_OK**: Ausführungsrechte prüfen
  - ▶ **F\_OK**: Existenz der Datei prüfen.
- ▶ Prüfungen werden mit der **realen** UID und GID vorgenommen

# Beispiel: Zugriffsrechte Prüfen

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <unistd.h>
4
5  static void usage() {
6      fputs("Usage: _access_ -MODE_FILE\n"
7            "MODE_is_one_or_more_of_rwx._Exit_successfully_if_FILE_exists\n"
8            "and_is_readable_(r),_writable_(w),_or_executable_(x).\n", stdout);
9      exit(EXIT_FAILURE);
10 }
11
12 int main(int argc, char *argv[]) {
13     int mode=0;
14     if(argc!=3) usage();
15
16     if(++argv=='-')
17         while(++argv)
18             switch(**argv) {
19                 case 'r': mode |= R_OK; break;
20                 case 'w': mode |= W_OK; break;
21                 case 'x': mode |= X_OK; break;
22                 default : usage();
23             }
24     else usage();
25
26     if (access(++argv,mode)) {
27         perror(*argv);
28         return EXIT_FAILURE;
29     }
30     return EXIT_SUCCESS;
31 }
```

# Die Dateierzeugungsmaske (**umask**)

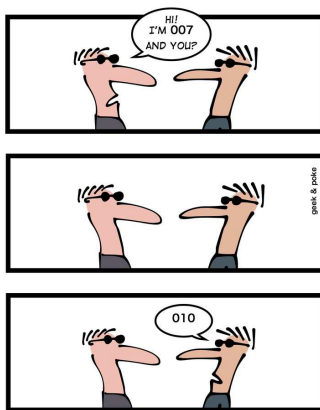
```
#include <sys/stat.h>

mode_t umask(mode_t mask);
```

- ▶ Neue Dateien und Verzeichnisse werden standardmäßig mit der Dateierzeugungsmaske (Umask) verknüpft
- ▶ Die Umask legt fest welche Rechte **nicht** zu vergeben sind
- ▶ **umask ()** gibt die vorherige Dateierzeugungsmaske zurück
- ▶ Zusammensetzung der Dateierzeugungsmaske
  - ▶ Einem oder mehrere durch | verknüpfte Zugriffsrechte oder
  - ▶ einer dreistelligen Oktalzahl (z. B. 0664).
  - ▶ **Achtung:** In C beginnen Oktalzahlen mit dem Prefix 0.

# Oktalzahlen in C: Illustration

## THE NEXT BOND



## Beispiel: Dateierzeugungsmaske

```
1  #include <stdlib.h>
2  #include <unistd.h>
3  #include <fcntl.h>
4  #include <stdio.h>
5  #include <sys/stat.h>
6
7  int main(int args, char *argv[]) {
8      umask(S_IXUSR | S_IXGRP | S_IRWXO | S_IWGRP);
9
10     for(int i=1; i <args; i++) {
11         int fd = open(argv[i], O_CREAT | O_RDONLY, 0777);
12         if(fd < 0) {
13             perror(argv[i]);
14             return EXIT_FAILURE;
15         }
16     }
17     return EXIT_SUCCESS;
18 }
```

**Frage:** Welche Zugriffsrechte hat eine mit dem obigen Programm erstellte Datei?

## 5.3: Lesen, Schreiben und Positionieren von Dateien

In diesem Abschnitt behandeln wir die folgenden drei Systemcalls:

1. `read()`

Lesen einer Datei

2. `write()`

Schreiben in eine Datei

3. `lseek()`

Positionieren des Datei-Offsets (*file offset*)

# Lesen von einer Datei

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

- ▶ **read()** liest von der Dateideskriptor **fd** zugewiesenen Datei bis zu **count** Bytes und kopiert diese in den Buffer **buf**
- ▶ Der Rückgabewert entspricht der tatsächlichen Anzahl gelesener Bytes oder **-1** im Fehlerfall
- ▶ **count** sollte immer ein Vielfaches von 512 sein
- ▶ **read()** startet den Lesevorgang beim Datei-Offset und erhöht dann dessen Position um die tatsächliche Anzahl gelesener Bytes

**Frage:** Warum kann die Anzahl tatsächlich gelesener Bytes von **count** abweichen?

# Beispiel: Dateien Vergleichen I

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <fcntl.h>
5  #include <string.h>
6
7  #define BUF_LEN 4096
8
9  void error(char *msg) {
10     perror(msg);
11     exit(EXIT_FAILURE);
12 }
13
14 void nequal(char *s1, char *s2) {
15     printf("%s_and_%s_differ.\n", s1, s2);
16     exit(2*EXIT_FAILURE);
17 }
18
19 int main(int args, char *argv[]) {
20     int fd1;
21     int fd2;
22     char buf1[BUF_LEN];
23     char buf2[BUF_LEN];
24
25     if (args != 3) {
26         fprintf(stderr, "usage: _readcompare_<file1>_<file2>\n");
27         exit(EXIT_FAILURE);
28     }
29 }
```



# Beispiel: Dateien Vergleichen II

```
30  if( (fd1 = open(argv[1], O_RDONLY)) < 0 ) error(argv[1]);
31  if( (fd2 = open(argv[2], O_RDONLY)) < 0 ) error(argv[2]);
32
33  while(1) {
34      ssize_t len1 = read(fd1, buf1, BUF_LEN);
35      ssize_t len2 = read(fd2, buf2, BUF_LEN);
36
37      if(len1 != len2)                nequal(argv[1], argv[2]);
38      if(memcmp(buf1, buf2, len1)) nequal(argv[1], argv[2]);
39
40      if(len1 < 1) break;
41  }
42  printf("%s_and_%s_are_identical.\n",argv[1] , argv[2]);
43
44  close(fd1);
45  close(fd2);
46  return EXIT_SUCCESS;
47  }
```

# Schreiben in eine Datei

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

- ▶ **write()** versucht **count** Bytes von **buf** in die Datei auf welche der Dateideskriptor **fd** verweist zu schreiben
- ▶ Der Rückgabewert entspricht der tatsächlichen Anzahl geschriebener Bytes, oder **-1** im Fehlerfall
- ▶ Nach dem Schreiben wird der Datei-Offset um die tatsächliche Anzahl geschriebener Bytes erhöht

**Frage:** Warum kann die Anzahl tatsächlich geschriebener Bytes von **count** abweichen?

# Beispiel: Dateien ausgeben

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <unistd.h>
4  #include <fcntl.h>
5
6  #define BUF_LEN 4096
7
8  void fdprint(int fd) {
9      char buf[BUF_LEN];
10     int len;
11     while( (len = read(fd, buf, BUF_LEN)) > 0 )
12         write(STDOUT_FILENO, buf, len);
13 }
14
15 int main(int args, char *argv[]) {
16     int fd;
17
18     if(args ==1) {
19         fdprint(STDIN_FILENO);
20         return EXIT_SUCCESS;
21     }
22
23     for(int i=1; i < args; i++) {
24         if( (fd = open(argv[i], O_RDONLY)) < 0 ) perror(argv[i]);
25         else { fdprint(fd); close(fd); }
26     }
27
28     return EXIT_SUCCESS;
29 }
```

# Positionieren des Datei-Offsets

```
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

- ▶ **off\_t** steht für **long int**
- ▶ **lseek()** verschiebt den zu **fd** zugehörigen Datei-Offset um **offset** Bytes. Der Parameter **whence** legt fest von welcher Ausgangsposition aus die Verschiebung stattfindet
- ▶ Bei Erfolg gibt **lseek()** den Datei-Offset zurück, ansonsten **-1**
- ▶ **lseek()** erlaubt es den Datei-Offset hinter das Dateiende zu setzen. Ein späterer Schreibzugriff füllt die Lücke mit **\0** Bytes
- ▶ **lseek(fd, 0, SEEK\_CUR)**: Ermittelt den aktuellen Datei-Offset

# Startpositionen

```
off_t lseek(int fd, off_t offset, int whence);
```

Liste der gültigen Werte für das Argument **whence**:

Wert	Ausgangsposition	<b>offset</b> -Wertebereich
<b>SEEK_SET</b>	Dateianfang	nur positive Werte
<b>SEEK_CUR</b>	Aktuelle Position	positive und negative Werte
<b>SEEK_END</b>	Dateiende	positive und negative Werte

# lseek - Beispiele

**Annahme:** `fd` ist der Dateideskriptor einer regulären Datei

**Frage:** Was bewirken die folgenden Aufrufe?

- ▶ `lseek(fd, 0L, SEEK_SET)`
- ▶ `lseek(fd, -23L, SEEK_CUR)`
- ▶ `lseek(fd, 42L, SEEK_END)`
- ▶ `lseek(fd, 20L, SEEK_SET)`
- ▶ `lseek(fd, -20L, SEEK_SET)`

# fill.c

```
1  #include <unistd.h>
2  #include <fcntl.h>
3  #include <errno.h>
4  #include <stdio.h>
5
6  int main() {
7      int fd = open("zyx.txt", O_RDWR | O_CREAT, S_IRUSR |
8          S_IWUSR);
9      lseek(fd, 4096, SEEK_END);
10     write(fd, "a", 1);
11     close(fd);
12
13     return errno;
14 }
```

**Frage** Was macht dieses Programm?

## 5.4: Kerneltabellen

Im Folgenden wollen wir uns anschauen wie der Linux-Kernel die geöffneten Dateien verwaltet, sprich was sich hinter einem Dateidesriptor **fd** verbirgt.

Der Linux-Kernel verwendet drei Datenstrukturen um geöffnete Dateien zu verwalten:

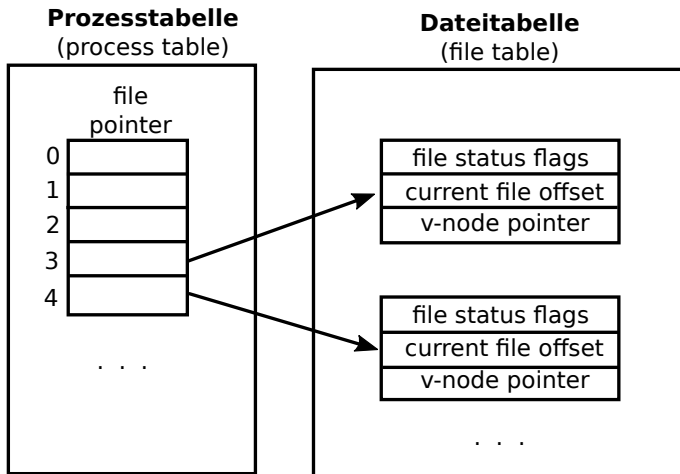
1. **Prozess-Liste** (engl. *process list*)
2. **Dateitabelle** (engl. *file table*)
3. **V-Node-Tabelle** (engl. *v-node table*)



# Prozesstabelle

- ▶ Die Verwaltung von Prozessen (ausgeführte Programme) gehört zu den wichtigsten Aufgaben eines Kernels
- ▶ Der Kernel legt zu jedem Prozess einen *process control block* (PCB) an, mit deren Hilfe er den Prozess überwacht
- ▶ PCBs sind Einträge in der **Prozesstabelle** (engl. *process table*).
- ▶ Ein PCB enthält zu jeder geöffneten Datei den Dateideskriptor **fd** und einen Zeiger auf einen Eintrag in der Dateitabelle
- ▶ Über die Prozess-ID (**PID**) kann auf einen bestimmten PCB zugegriffen werden.
- ▶ Geschäfts-Analogie:
  - ▶ Kernel: Firma
  - ▶ Prozess: Kunde
  - ▶ PCB: Kundendaten

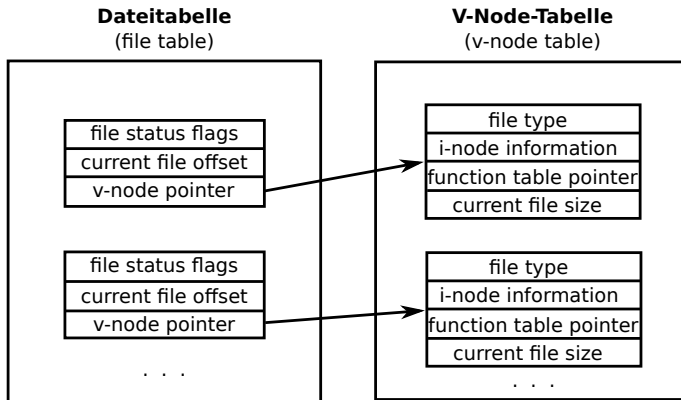
# Prozesstabelle – Visualisierung



# Dateitabelle

- ▶ Die Dateitabelle ist eine systemweite Tabelle
- ▶ Zu jeder geöffneten Datei gibt es einen Eintrag
- ▶ Analogie: Ein **Eintrag** ist eine **Verbindung** zu einer **Datei**
- ▶ Unter Unix gibt es die Möglichkeit, dass sich mehrere Prozesse eine Verbindung teilen. Ein Prozess kann einen Kindprozess erzeugen welcher die *Dateiverbindungen* erbt
- ▶ Eine Dateiverbindung enthält die folgenden Informationen:
  - ▶ Gesetzte `file status flags` (**O\_RDONLY**, **O\_APPEND**, ...)
  - ▶ Aktueller File-Offset
  - ▶ Zeiger auf einen Eintrag in der V-Node-Tabelle

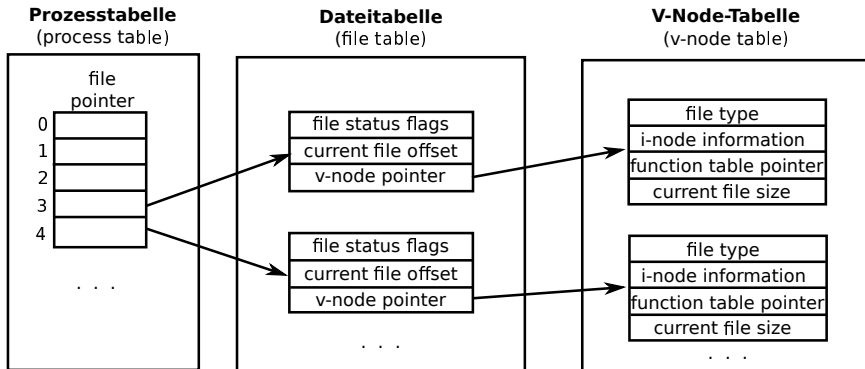
# Dateitabelle – Visualisierung



# V-Node-Tabelle

- ▶ Ein V-Node (virtual node) ist ein Eintrag in der V-Node-Tabelle
- ▶ Der Kernel legt für jede geöffnete Datei einen V-Node an
- ▶ Ein V-Node enthält die folgenden Informationen:
  - ▶ Dateityp (reguläre Datei, Verzeichnis, Geräte, Socket, ...)
  - ▶ **I-Node**: Datenstruktur mit den physikalischen Informationen über eine Datei
  - ▶ Ein Zeiger auf eine Tabelle von Funktionen welche auf dem V-Node ausgeführt werden können (`read`, `write`, `close`, ...). Die Funktionen werden durch die entsprechenden Systemcalls ausgeführt
  - ▶ Die aktuelle Dateigröße

# Kerneltabellen – Visualisierung



## 5.5: Virtual File System (VFS)

- ▶ Ein Virtual File System (VFS) ist eine Abstraktionsschicht für tatsächliche Dateisysteme (engl. `file system`)
- ▶ Die vorgestellten Funktionen und Kerneltabellen sind Teil des Virtual File System (VFS)
- ▶ Im VFS ist eine Datei als eine Sequenz von Bytes abstrahiert
- ▶ Durch den Aufruf Systemcalls `read(fd, ...)` wird die Methode `read()`, welche durch den entsprechenden V-Node referenziert wird, unabhängig vom Dateisystem ausgeführt
- ▶ Der Dateisystemtreiber muss die `read()` Funktionen, welche durch einen V-Node referenziert werden, implementieren
- ▶ Der Vorgang ist für die Systemcalls `write()` und `close()` analog

# Der Windows-Ansatz

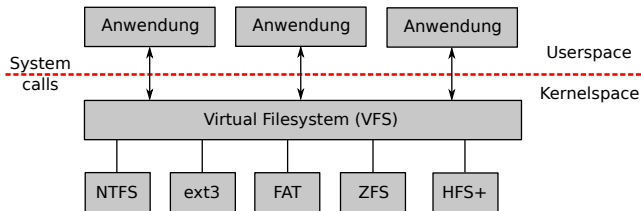
- ▶ Bei Windows ist der Laufwerksbuchstaben (C:, D:, ...) einem bestimmten Dateisystem zugeordnet
- ▶ Wenn ein Prozess eine Datei öffnet, kann diese an Hand des kanonischen Dateipfades ermittelt werden
- ▶ Windows weiß daher stets welchem Dateisystem Sie eine Anfrage übergeben muss
- ▶ Es wird nicht versucht, uneinheitliche Dateisysteme in einem großen Ganzen zu vereinigen



# Der Unix-Ansatz

- ▶ Unter Unix kann ein Dateisystem in jedem beliebigen Ordner eingehangen werden
- ▶ Es kann also sein, dass unter dem `ext3`-Wurzeldateisystem `/` eine `ext4`-Partition unter `/usr` eingehängt ist, und ein `xfs`-Dateisystem unter `/home/bob` eingebunden ist
- ▶ Für den Benutzer gibt es eine einzige Dateisystemhierarchie, welche eine Vielzahl von (inkompatiblen) Dateisystemen umfassen kann ohne dass dies für den Benutzer oder Prozess sichtbar ist
- ▶ Hinter der technischen Realisierung verbirgt sich das VFS-Konzept bei dem Kleinman (Sun Microsystems 1986) maßgeblich beteiligt war

# Virtual File System

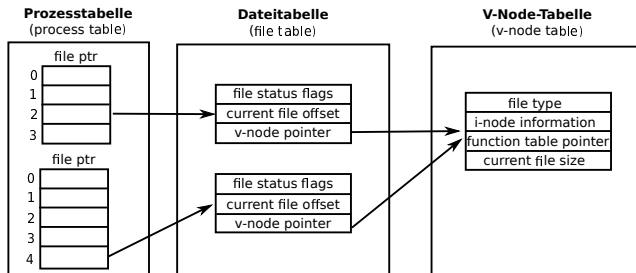


Quelle: <http://www.rasger.de/linux/virtuellesdateisystem.htm>

- ▶ Alle Systemaufrufe (POSIX.1), die Dateien betreffen, werden an das VFS weitergeleitet
- ▶ Das VFS ruft für jeden Systemcall die Funktion des konkreten Dateisystems auf

## 5.6: File Sharing

### File-Sharing



- ▶ **File Sharing:** Mehrere Prozesse öffnen die gleiche Datei
- ▶ Jeder Prozess bekommt seinen eigenen Eintrag in der Dateitabelle
- ▶ Daher bleiben die Datei-Offsets von verschiedenen Prozessen unabhängig
- ▶ In der V-Node-Tabelle existiert nur ein Eintrag

# File Sharing – Auswirkungen von Dateioperationen

- ▶ Nach jedem **write()** -Aufruf wird der Datei-Offset des entsprechenden Dateitabellen-Eintrags um die Anzahl geschriebener Bytes erhöht
- ▶ Falls sich durch einen **write()** -Aufruf die Dateigröße ändert, wird der V-Node aktualisiert
- ▶ Wird eine Datei mit **O\_APPEND** geöffnet, wird bei einem **write()** -Aufruf zuerst der Datei-Offset auf die aktuelle Dateigröße gesetzt (v-node); dann wird geschrieben
- ▶ Bei einem **lseek()** -Aufruf wird der entsprechende Dateitabellen-Eintrag aktualisiert. **SEEK\_END** entspricht immer der aktuellen Dateigröße (v-node)
- ▶ Solange nur ein Prozess schreibt, funktioniert das Konzept
- ▶ Bei mehreren schreibenden Prozessen werden atomare Operationen benötigt

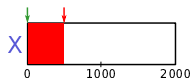
# Schreib-Schreib-Konflikt

- ▶ Angenommen, mehrere Prozesse haben parallel eine Datei zum Schreiben geöffnet
- ▶ Der Kernel verwaltet für jeden Prozess einen eigenständigen Datei-Offset
- ▶ Problem: Ein Prozess kann jederzeit unterbrochen werden
- ▶ Dadurch können sich die Prozesser gegenseitig beliebige Teile ihrer Inhalte überschreiben

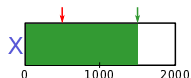
# Schreib-Schreib-Konflikt – Beispiel

Die Prozesse **A** und **B** schreiben parallel in die Datei **X**. **A** schreibt 1000 und **B** 2000 Bytes. Das Resultat könnte wie folgt ausschauen:

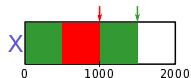
- ▶ **A** wird nach dem Schreiben von 500 Bytes von **B** unterbrochen



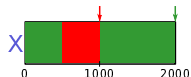
- ▶ **B** wird nach dem Schreiben von 1500 Bytes von **A** unterbrochen



- ▶ **A** schreibt die restlichen 500 Bytes

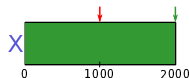


- ▶ **B** schreibt die restlichen 500 Bytes

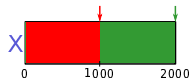


# Die write()-Operationen

- ▶ Eine **write()**-Syscall kann nicht von einem anderen **write()**-Syscall unterbrochen werden.
- ▶ In unserem Beispiel gibt es daher nur die beiden folgenden Möglichkeiten:
  - ▶ **A** kommt vor **B** zum Zug.

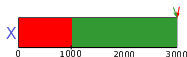


- ▶ **B** kommt vor **A** zum Zug.

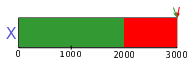


# Daten Anhängen

- ▶ Das **O\_APPEND**-Flag verhindert, dass Daten von anderen Prozessen überschrieben werden.
- ▶ Falls mehrere Prozesse in eine Datei schreiben, ist es daher ratsam, dass alle beim Öffnen das **O\_APPEND**-Flag setzen
- ▶ Falls in unserem Beispiel **A** und **B** beim Öffnen der Datei **X** das **O\_APPEND**-Flag gesetzt hätten, dann wäre der Ausgang wie folgt:
  - ▶ **A** kommt vor **B** zum Zug.



- ▶ **B** kommt vor **A** zum Zug.





# Zusammenfassung

Sie sollten ...

- ▶ ... die Systemcalls zum Öffnen, Schließen, Lesen, Schreiben und der Positionierung des File-Offset verinnerlicht haben.
- ▶ ... Das Zugriffsrechtssystem verstanden haben.
- ▶ ... Den Unterschied zwischen realer und effektiver Benutzer-ID kennen.
- ▶ ... verstanden haben wie der besprochene VFS-Teil bestehend aus Prozesstabelle, Dateitabelle und V-Node-Tabelle funktioniert.
- ▶ ... in der Lage sein die Begriffe `File Sharing` und die funktionsweise von `write()` zu erklären.
- ▶ ... wissen was ein Schreib-Schreib-Konflikt ist.