Kapitel 6: Elementare I/O-Funktionen

6: 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 schliesen
  - 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
- Atomare Operationen

Das Öffnen einer Datei

Was passiert, wenn der Prozess **P** die regulären 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 Handel (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**.

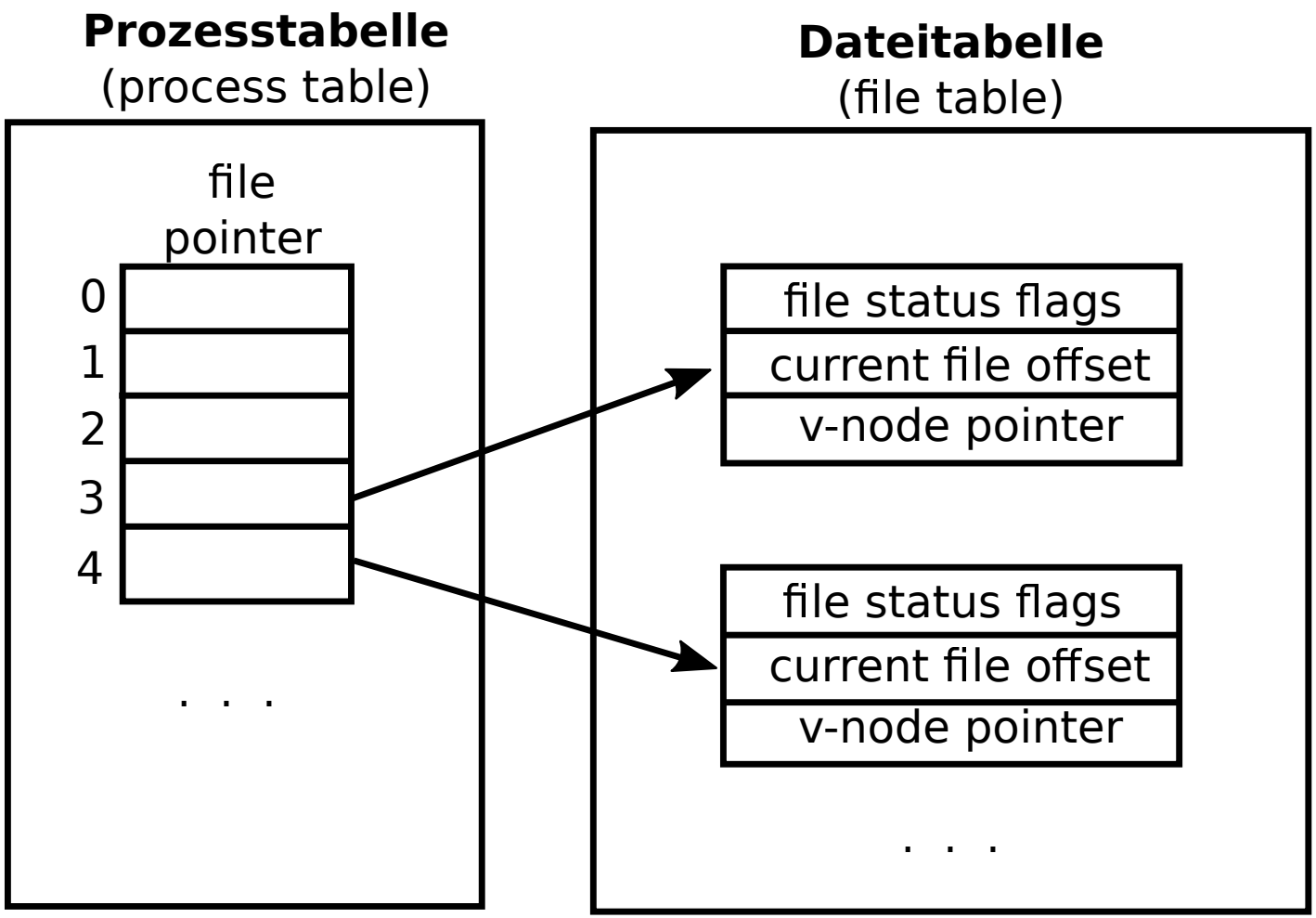
7. Der Kernel teilt **P** den Datei-Handle (Dateideskriptor) **D** mit.

   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).

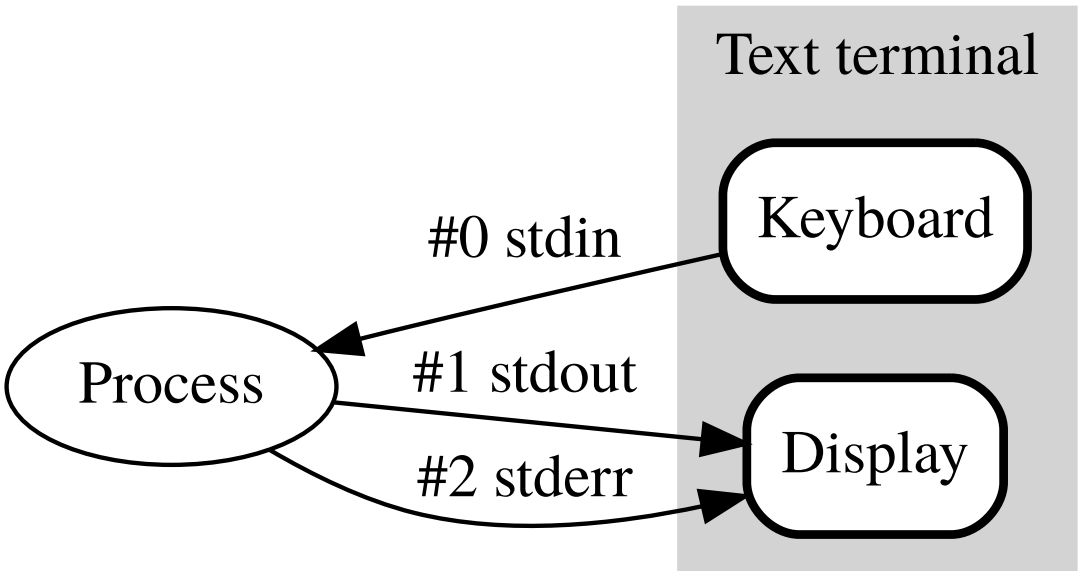
Dateideskriptor: Illustration



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 Domain

6.1: 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`** ($\rightarrow$ 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 fehlt 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;
}
```


BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

Zugriffsrechte - Spezielle Zugriffsrechte

Es gibt noch drei weitere Spezialzugriffsrechte.

- `S_ISUID`: set-user ID Bit
  Das Programm läuft mit den Rechten des Eigentümers
- `S_ISGIO`: set-group ID Bit
  Das Programm läuft mit den Rechten des Gruppe
- `S_ISVIX`: 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 Verzeichnis. 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

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 Zugriffssrechte

  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 }
```


BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

Exkurs: 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 geestartet 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 }
```


BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

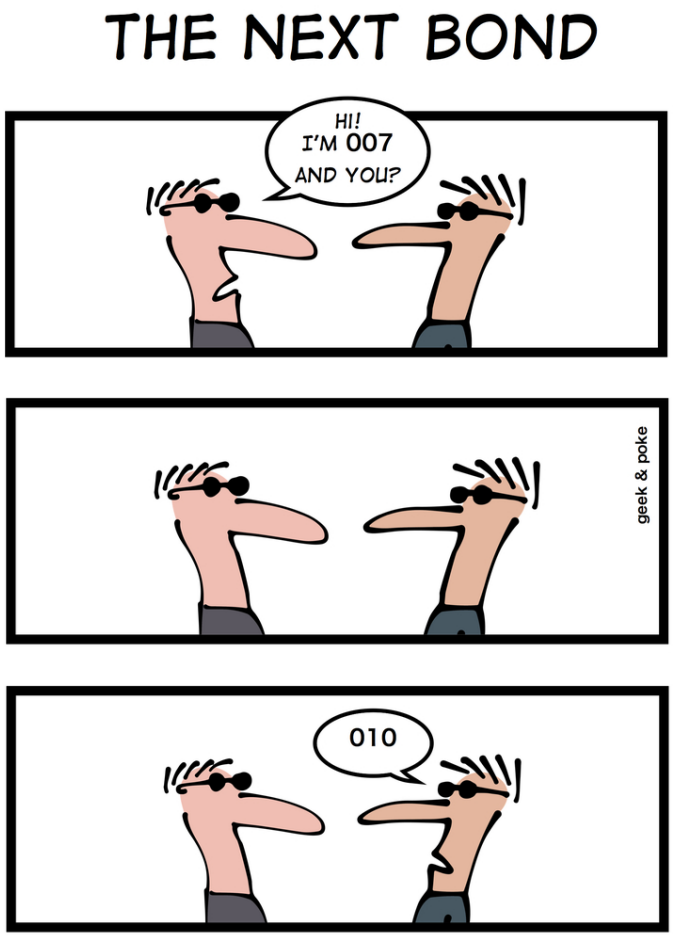
Die Dateierzeugungsmaske (`umask`)

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

mode_t umask(mode_t mask);
```

- Neue Dateien und Verzeichnise 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 Zugriffssrechte oder
  - einer dreistelligen Oktalzahl (z. B. `0664`).
  - **Achtung:** In C beginnen Oktalzahlen mit dem Prefix `0`.

Oktalzahlen in C: Illustration



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?

6.2: 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 }
```


BEUTH HOCHSCHULE FÜR TECHNIK BERLIN
University of Applied Sciences

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[1]);
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**
- **lssek()** 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	`offset`-Wertebereich
`SEEK_SET`	Dateianfang	nur positive Werte
`SEEK_CUR`	Aktuelle Position	positive und negative Werte
`SEEK_END`	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 |
       S_IWUSR);
8    lseek(fd, 4096, SEEK_END);
9    write(fd,"a",1);
10   close(fd);
11
12   return errno;
13 }
```

Frage Was macht dieses Programm?

6.3: Kerneltabellen

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

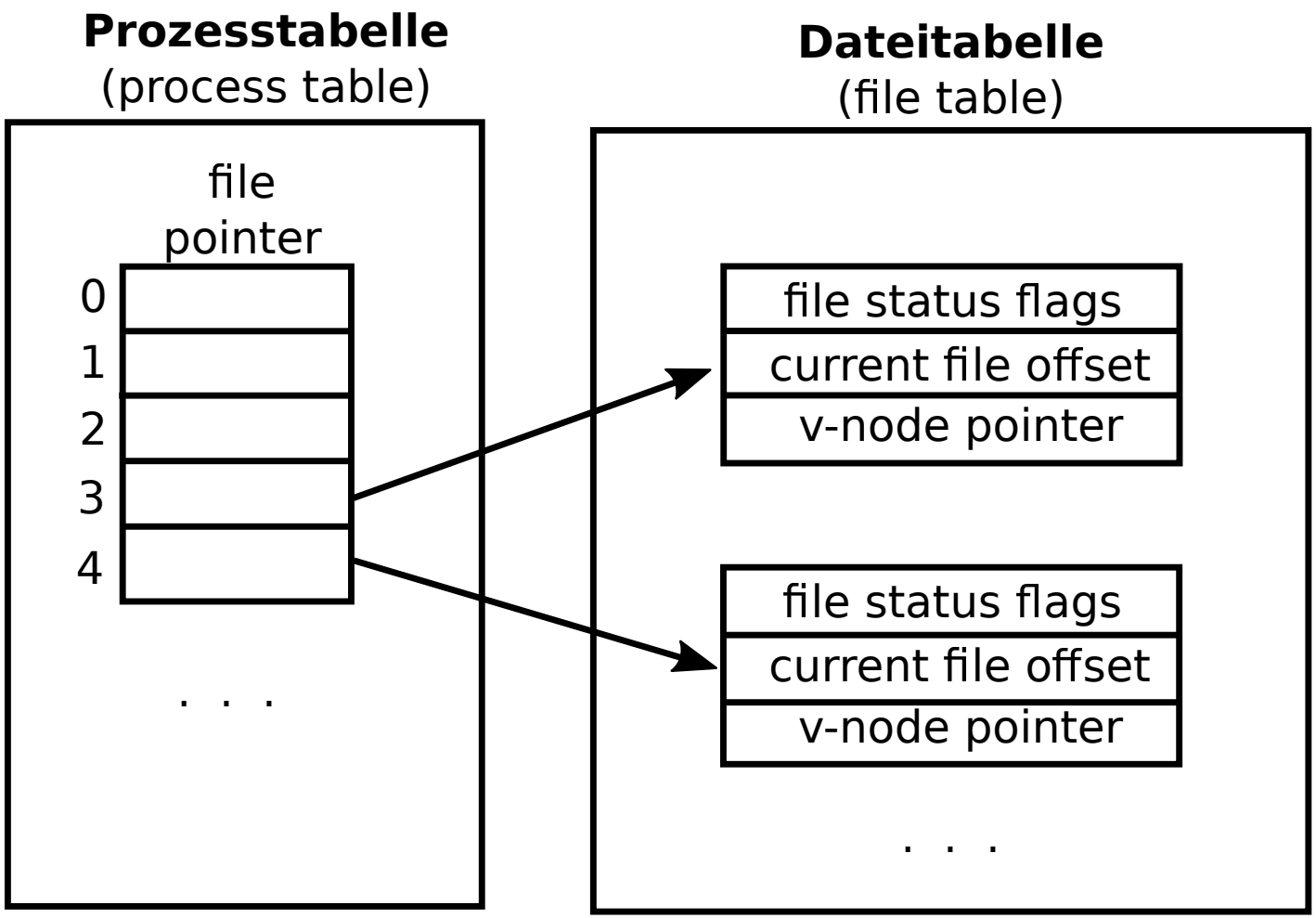
Der Linux-Kernel verwendet drei Datenstrukturen um geöffnete Dateien zu verwalten:

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 enhält 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

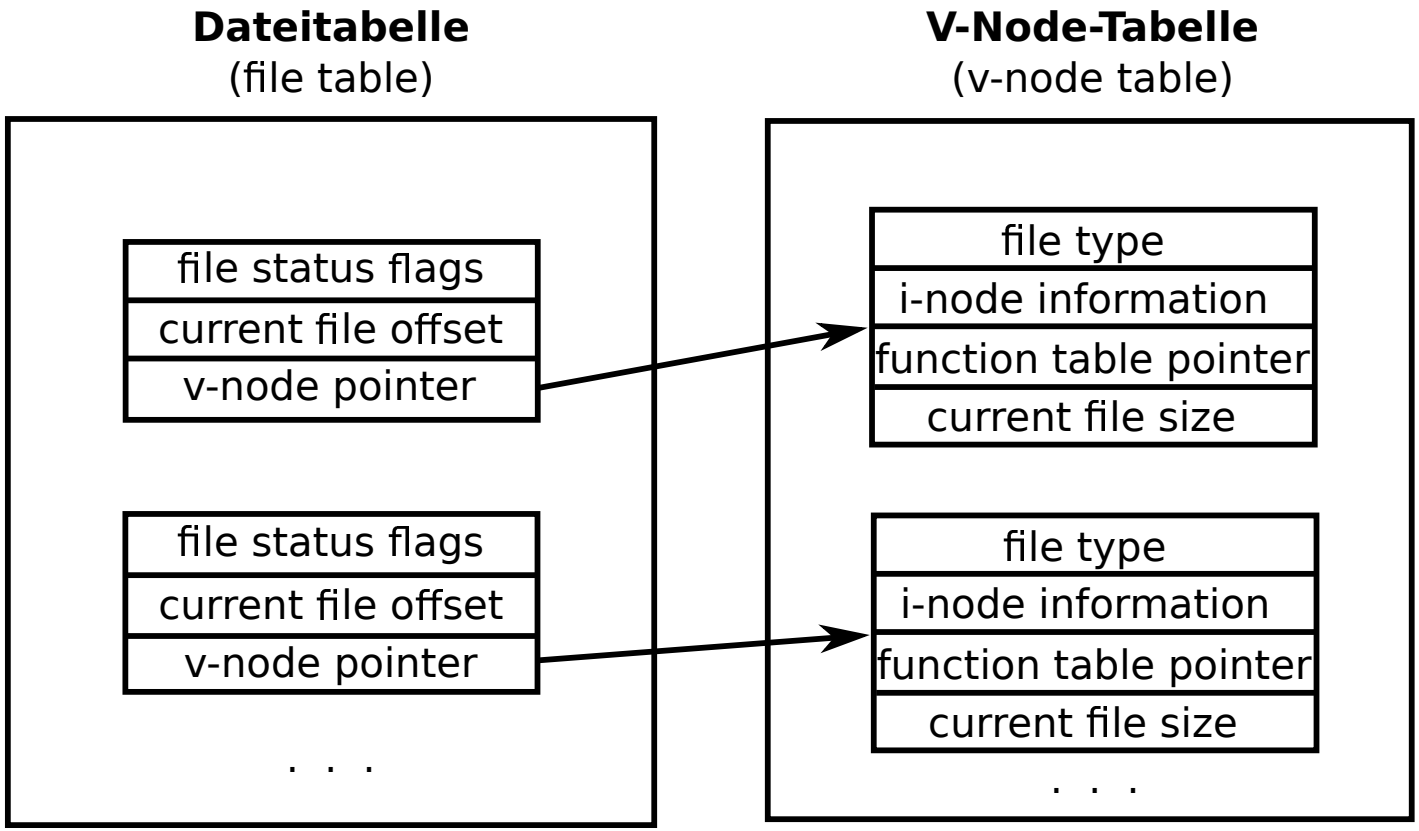
Prozesstabelle – Visualisierung



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

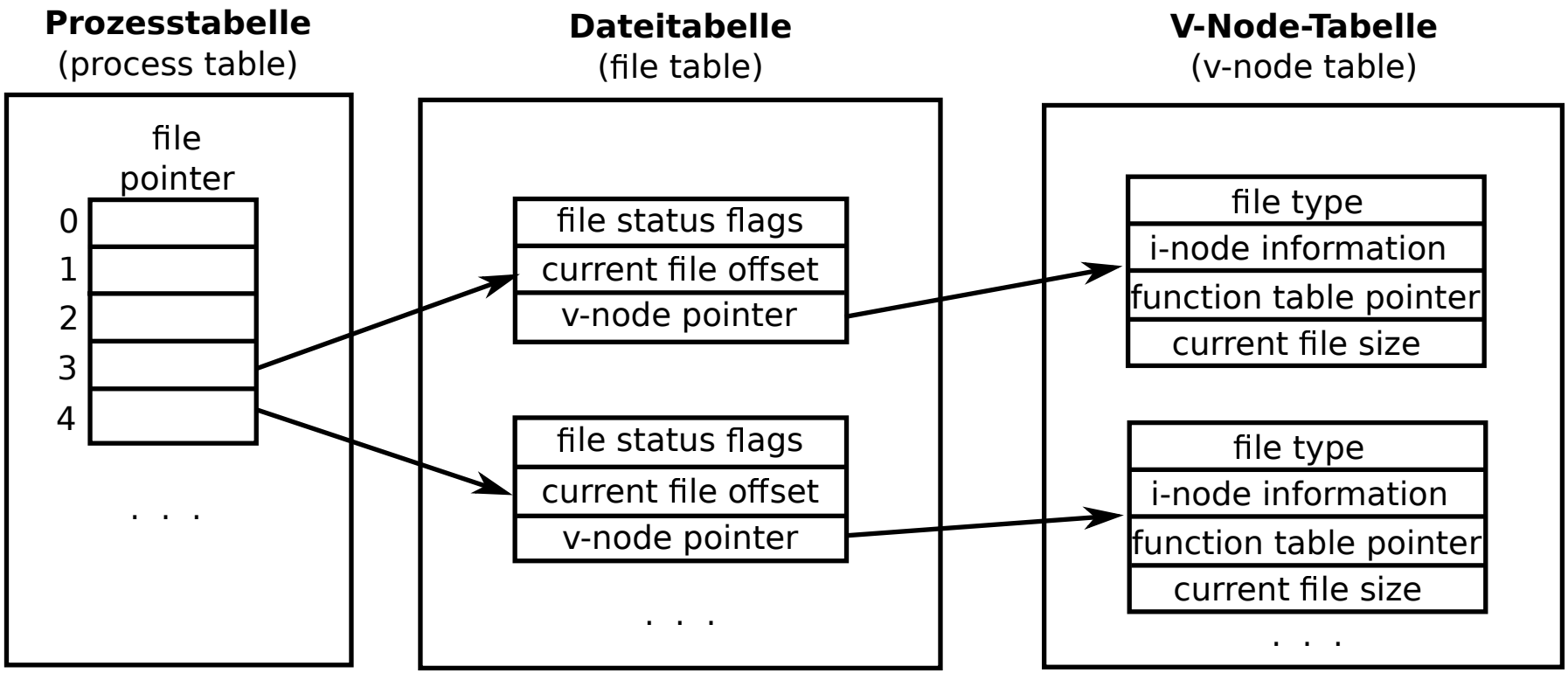
Dateitabelle – Visualisierung



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



6.4: 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

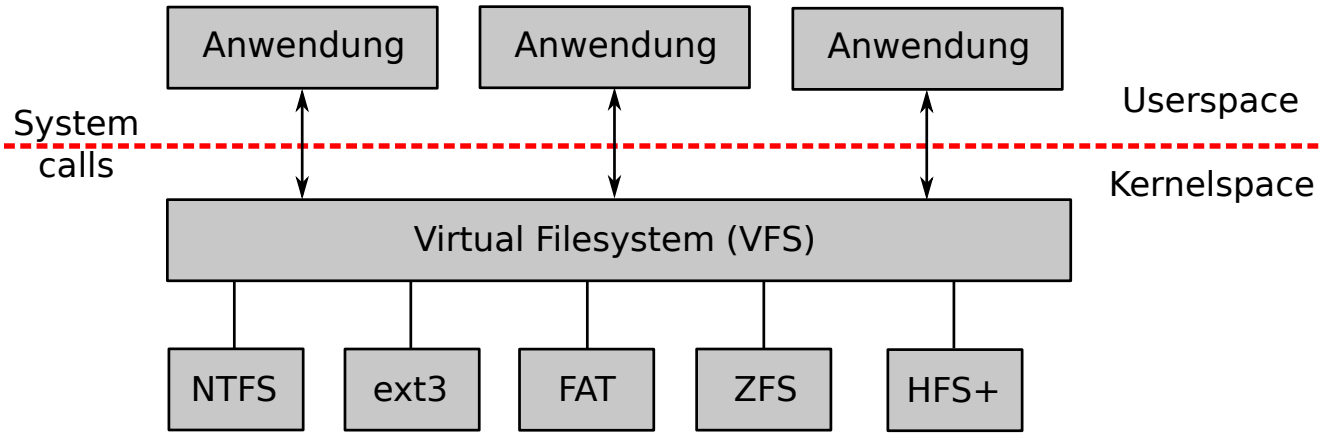
Der Windows-Ansatz

- Bei Windows ist der Laufwerksbuchstaben (C:, D:, . . .) einem bestimmten Dateisystem zugeordnet
- Wenn ein Prozess eine Datei öffnet, kann diese an Hand das 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 Dateisystemhierachie, 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

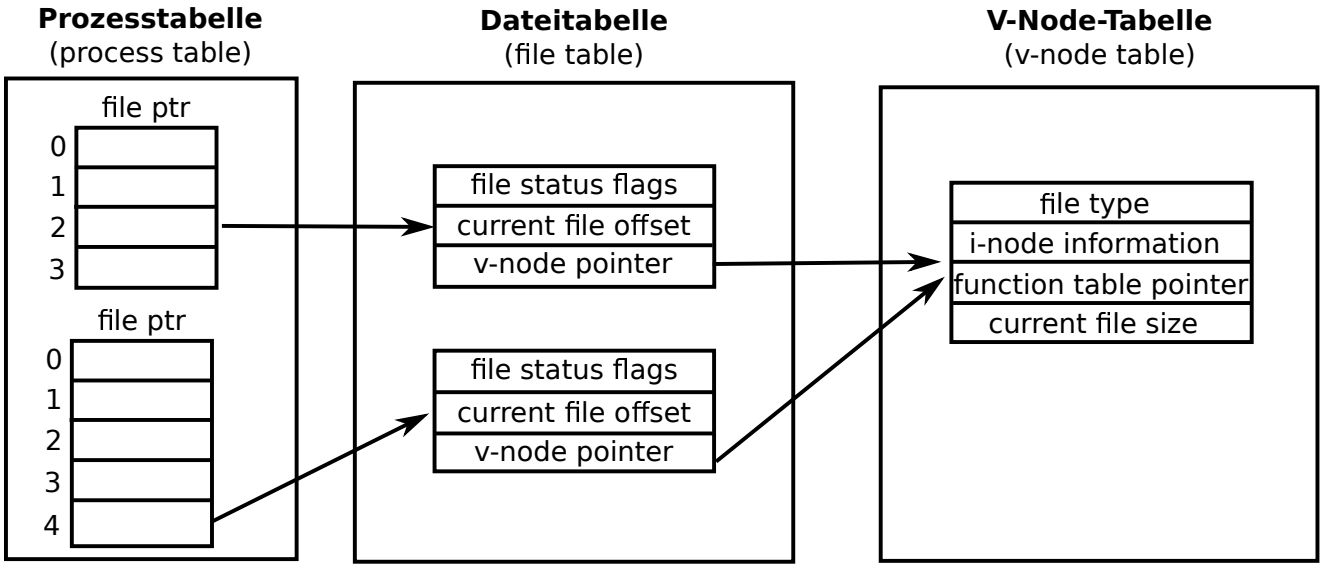


Quelle: `http://www.rasger.de/linux/virtuellesdateisystem.htm`

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

6.5: File Sharing und atomare Operationen

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 schreibenen Prozessen werden atomare Operationen benötigt

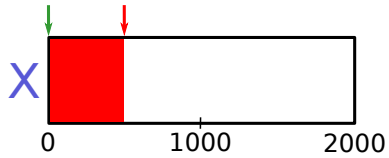
Schreib-Schreib-Konflikt

- 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

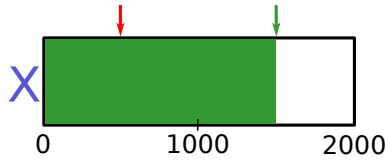
Schreib-Schreib-Konflikt – Beispiel

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

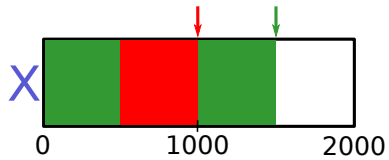
- A wird nach dem Schreiben von 500 Bytes von B unterbrochen



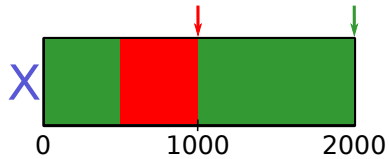
- B wird nach dem Schreiben von 1500 Bytes von A unterbrochen



- A schreibt die restlichen 500 Bytes

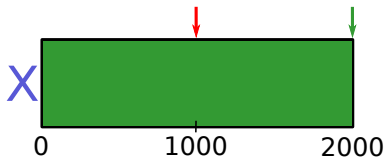


- B schreibt die restlichen 500 Bytes

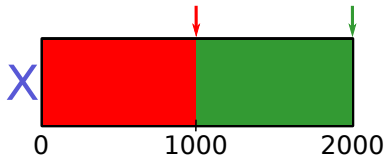


Atomare Operationen

- Bei unixoiden Betriebssystemen ist der Syscall `write()` atomar
- 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.

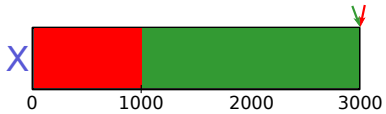


  - B kommt vor A zum Zug.

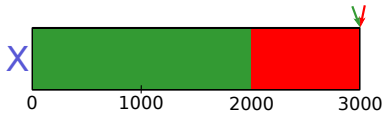


Atomare Operationen und Anhängen

- 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 Zugriffsrechtesystem 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 `atomare Operation` zu erklären.
- . . . wissen was ein Schreib-Schreib-Konflikt ist.