Systemprogrammierung

Jan Fässler

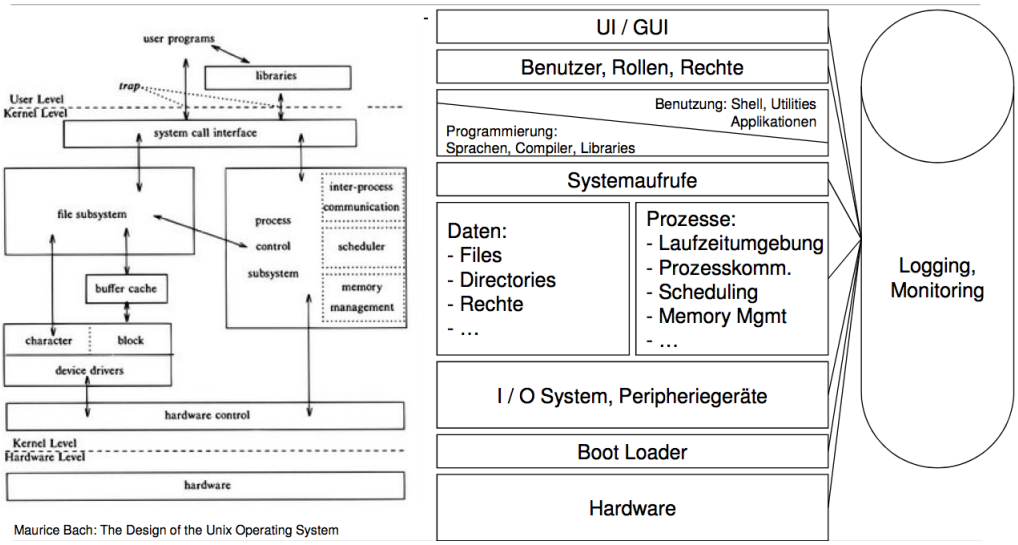
3. Semester (HS 2012)

If you use this documentation for a exam, you should offer a beer to the authors!

Inhaltsverzeichnis

1 Unix/Linux

1.1 Aufbau

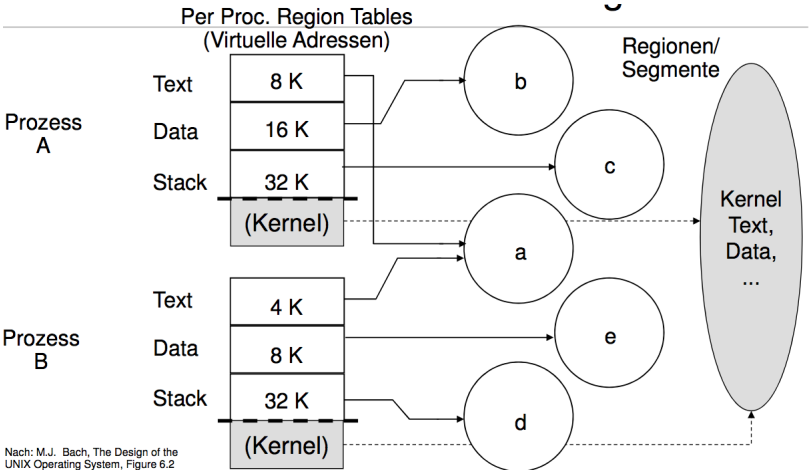


1.2 Prozesse

1.2.1 Steuerungssystem

- Prozesse kreieren & starten
- Prozesse schedulen, Warteschlangen, Ressourcenverbrauch
- Prozesse stoppen / unterbrechen / terminieren
- Prozess-Signalisierung und -kommunikation
- Faire Zuordnung von Hauptspeicher und anderen geteilten Ressourcen
- Ein-/Auslagerung von Prozessen bei vollem Speicher
- Prozesse und ihre Zustände anzeigen

1.2.2 Aufbau



1.2.3 Kernel und User Mode

- Ein Prozess hat mindestens zwei Ausführungsmodi:

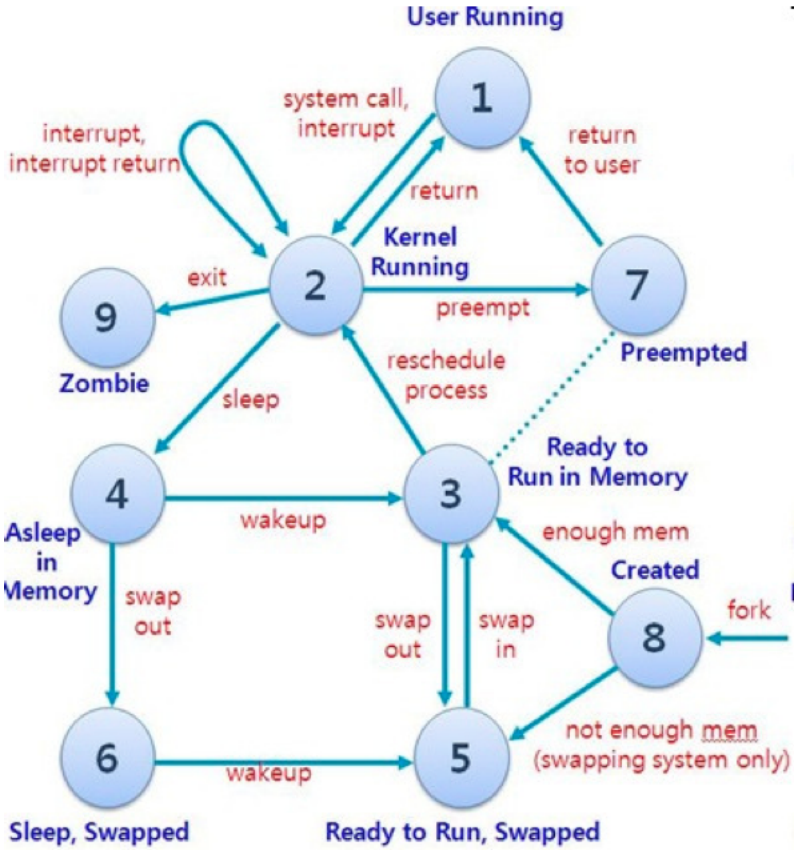
  User Mode Es wird der normale Programmcode ausgeführt.

  Kernl Mode Es werden Systemaufrufe ausgeführt oder Ausnahmen behandelt.

- Der Übergang erfolgt durch einen Systemaufruf durch das Programm, eine Ausnahmesituation oder durch asynchrone Events

- Beide Modi haben separate Segmente und sind voneinander abgeschirmt.

1.2.4 Zustände



1.3 Memory

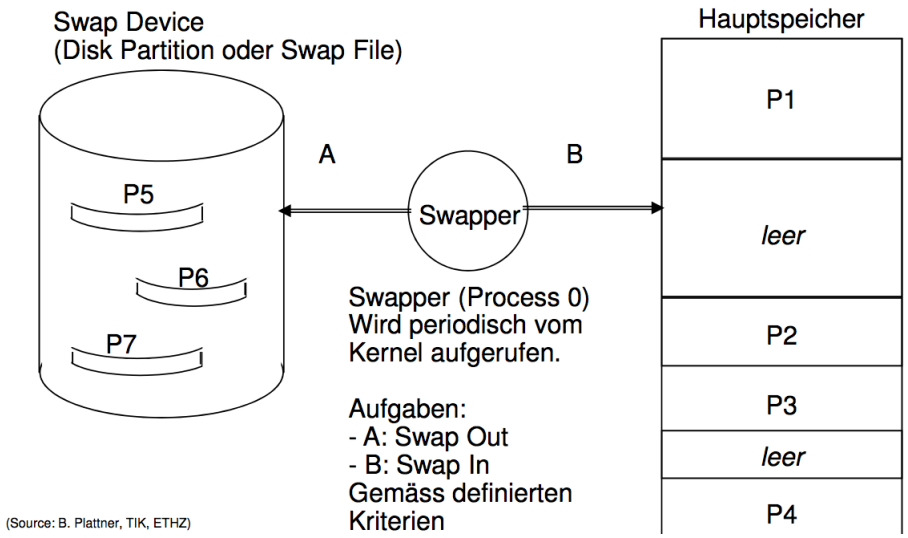
1.3.1 Virtual Memory

- Erweiterung des Hauptspeichers pro System (mehr Prozesse im System als Speicher verfügbar), oder pro Prozess (einzelner Prozess grösser als verfu?gbarer Hauptspeicher).

- Systematische Abstraktion fu?r systemspezifische Overlay- Techniken.

- Organisation des Hauptspeichers in gleich grosse, einheitlich adressierbare Einheiten.

1.3.2 Swapping

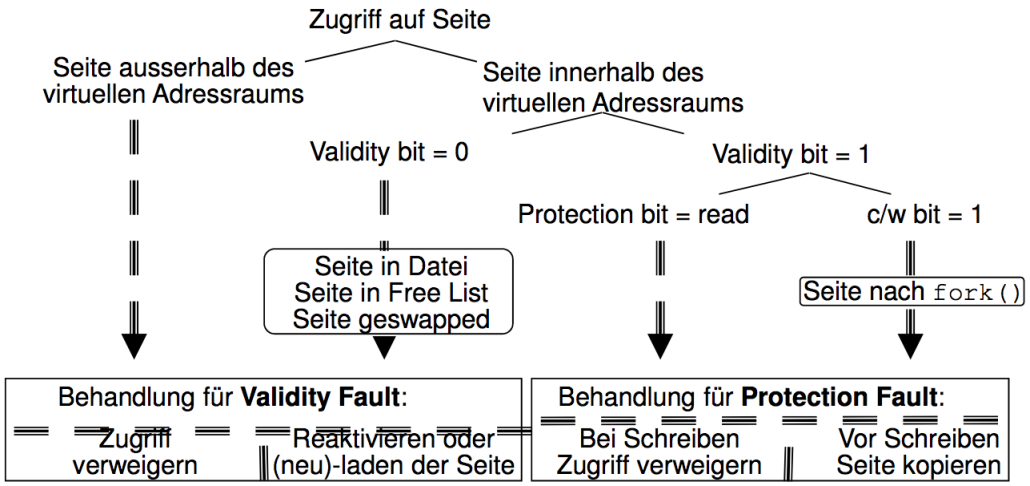


1.3.3 Paged Memory

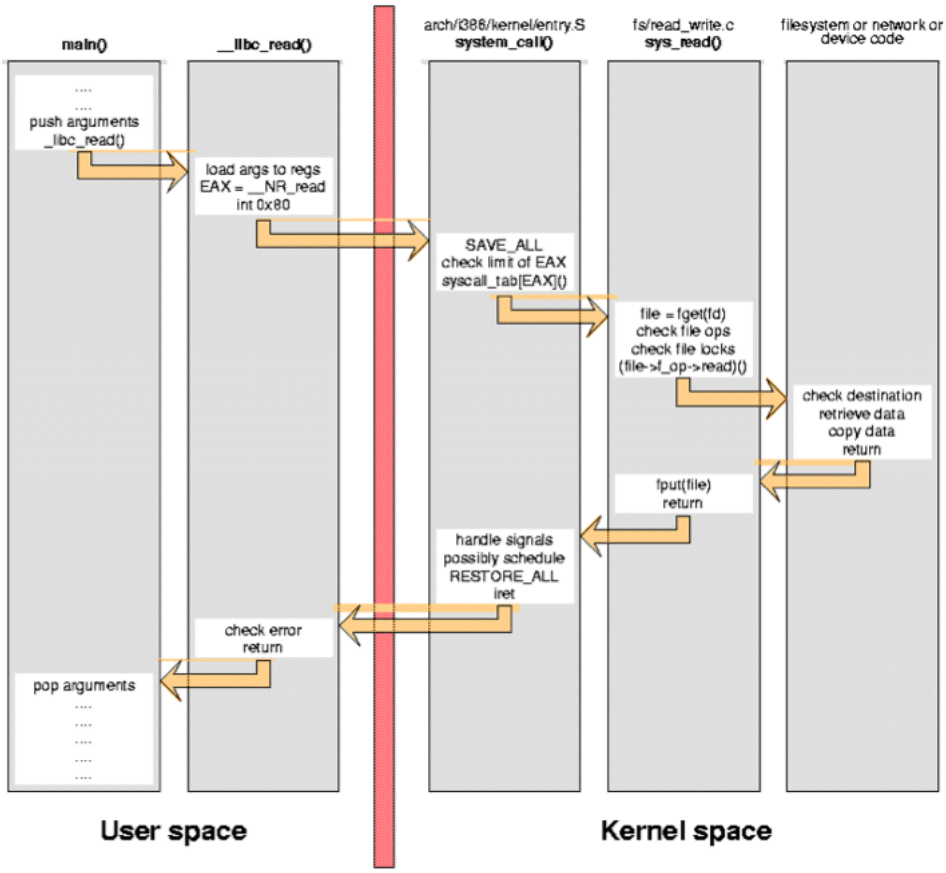
Design und Layout der Datenstrukturen fu?r die seiten-orientierte Speicherverwaltung variieren zwischen Unix- und Linux-Varianten und sind zudem von den Hardware- Eigenschaften abhängig. Linux verwendet zum Beuispiel eine 3-stufige Seitentabelle:

- Page Directory pro Prozess und für den BS-Kern
- Page Mid-level Directory (für 64 bit CPU-Architekturen) Paged Memory II
- Page Table, enthält Seitenbeschreibungen und Verweise auf den physische Speicheror

1.3.4 Fehlerzustände



2 System Call Schnittstelle



2.1 Prozess-System

fork() Erzeugung

exit() Beendigung

exec() Überlagerung des Prozesses

wait() Warten auf Prozesstermination (Kindprozesse)

sleep() Freiwilliges schlafen des Prozesses.

kill() Senden eines Signals (32 verschiedene)

signal() Signalbehandlung

2.2 Datei-System

creat() Anlegen einer Datei

mknod() Anlegen eines Ordners

open() Öffnen einer Datei

close() Schliessen einer Datei

unlink() Löschen einer Datei

read() Lesen aus einer Datei

write() Schreiben in eine Datei

lseek() Vorwärts-/Rückwertsbewegung

ioctl() Kontrollieren der Eigenschaften

dup() Duplizieren eines Dateideskriptors

chown, chmod, umask Zugriffsrechte

chdir() Navigation im Dateisystem

2.3 Directory Handling

opendir() Öffnen eines Verzeichnises

readdir() Lesen eines Verzeichnises

writedir() Schreiben eines Verzeichnises

closedir() Schliessen eines Verzeichnises

2.4 Speicherverwaltung

malloc() Speicher allozieren

free() Speicher freigeben

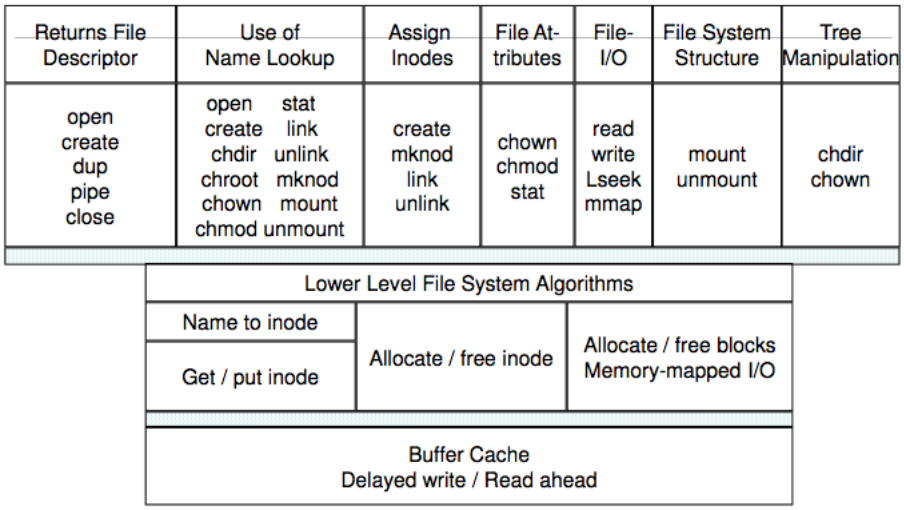
2.5 Weitere

pipe() Basis-Interprozesskommunikation

socket() Interprozesskommunikation lokal oder u?ber Netzwerke

3 Dateisystem

3.1 Übersicht



3.2 Relevante Dateisystem Algorithmen

3.2.1 Pfadnahme zu inode

namei() öffnet das aktuelle oder Root Verzeichnis . Navigiert rekursiv und basierend auf den Pfadkomponenten durch den Dateisystem- Baum bis ein Fehler auftritt oder die Datei gefunden wird. Umfasst eine Cache Struktur von kürzlich benutzten Namen und der zugehörigen inode-Nummer.

3.2.2 Gemeinsame Nutzung von Dateien

Zwei Prozesse können die selbe Datei für Lese- oder Schreibzugriff öffnen. Beide haben separate Lese-/Schreib-Indices und es gibt keine Konsistenzwahrung durch den Kernel, ausser für einzelne *read()* und *write()* Operationen, die atomar ausgeführt werden. Nach einem *fork()* eines Prozesses mit offenen Dateien teilen sich beide Prozesse den Lese-/Schreib-Index in der Dateitabelle.

3.2.3 Datei Locking

Eine Schwachstelle in Unix. Einige Unix- Varianten und Linux erlauben das Locking von Dateien pro read/write-Operation auf einem inode mittels Semaphoren. Der POSIX Standard verlangt sogar das Locken von Teilen einer Datei, aber dies wurde nur selten implementiert. Die meisten Unix Varianten unterstützen advisory locks (flock) oder Lock Dateien, die andere Prozesse jedoch ggf. Ignorieren können.

3.2.4 Mount / Unmount

Ein Directory kann als mount point diesen - das Directory muss dafür nicht leer sein, aber die enthaltenen Dateien sind nicht sichtbar, solange das Verzeichnis als Mount Point aktiv ist. Eine Mount-Tabelle entha?lt alle aktuell gemounteten Dateisysteme. Der automounter-Prozess kann zudem Dateisysteme bei Bedarf mounten/unmounten.

3.3 Allokation

3.3.1 inode

Dateisystem feststellen, Superblock locken (Schlafen wenn besetzt), nächsten inode aus der free list holen - wenn Liste leer, auffüllen, wenn danach inode verfügbar return inode, sonst return.

3.3.2 Datenblock

Dateisystem feststellen, Superblock locken (Schlafen wenn besetzt), nächsten freien Datenblock aus der free list (meist Bitmap) holen, wenn kein Datenblock frei ist, Schlafen bis Datenblock verfügbar wird).

3.4 Synchronisation von Zugriffen auf das Dateisystem

Der Superblock jedes Dateisystems enthält Lock Bits, um Prozessen bei schreibendem Zugriff (inode oder Datenblock holen) atomaren Zugriff auf die Datenstrukturen im Superblock zu erlauben. Dennoch kann es zu race conditions kommen, da die Locks nur sehr kurzlebig sein dürfen (Performance!) der Prozess jederzeit preempted werden kann. Es gibt Replikate des Superblocks im Dateisystem, diese werde jedoch nicht aktualisiert.

3.5 Geräte

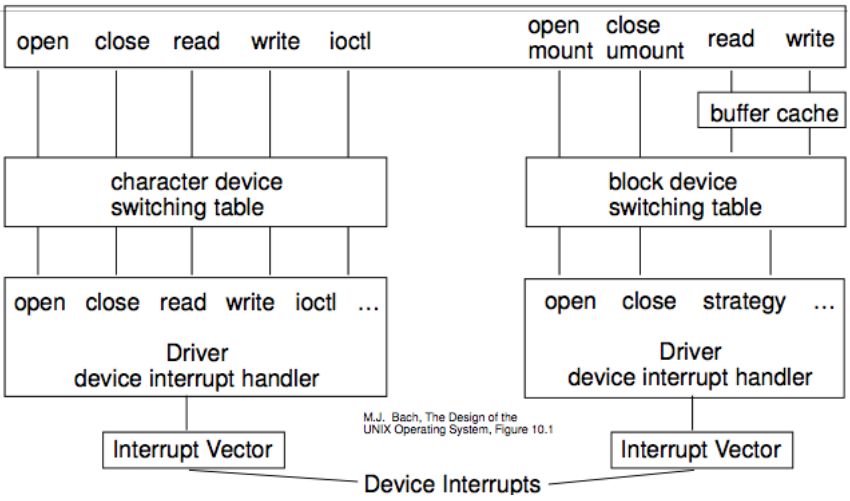
3.5.1 Einbindung

- Zentrales Element: Gerätespezialdateien im /dev bzw. /devices Dateisystem als einheitliche Schnittstelle, Dateideskriptor im Prozess.
- Major / Minor Device Number zur Identifikation
- Zugriffsrechte auf die Gerätespezialdateien sind relevant
- Geräte in verschiedenen Betriebs-Modi: block- oder zeichenweiser Zugriff
- Echte Geräte und Pseudo-Geräte (z.B. virtuelle Terminals, Netzwerkprotokolle oder /dev/null)

3.5.2 Gerätetreiber

Gerätetreiber sind die einzige Schnittstelle, über die ein Prozess mit Geräten kommunizieren kann. Sie sind Teil des kernel-Codes des Systems, und werden entweder statisch beim Systemstart oder zur Laufzeit (in Linux: insmod/rmmod) geladen. In Unix sind Gerätetreiber Teil jedes Prozesses (über den Kernel-Code) - in anderen Betriebssystemen sind sie nur speziellen Kommunikationsprozessen zugänglich über die die anderen Prozesse dann mit Geräten kommunizieren müssen.

3.5.3 Gerätespezialdateien



3.6 Beispiele

3.6.1 Verzeichnismanipulation

Listing 1: Verzeichnismanipulation

```
1  int main( int argc, char *argv[] ) {
       DIR *pDIR;
       struct dirent *pDirEnt;
       /* Open the current directory */
       pDIR = opendir(".");
6      if ( pDIR == NULL ) {
           fprintf( stderr, "%s %d: opendir() failed (%s)\n", __FILE__,
               __LINE__, strerror( errno ));
           exit( -1 );
       }
       /* Get each directory entry from pDIR and print its name */
11     pDirEnt = readdir( pDIR );
       while ( pDirEnt != NULL ) {
           printf( "%s\n", pDirEnt->d_name );
           pDirEnt = readdir( pDIR );
       }
16     /* Release the open directory */
       closedir( pDIR );
       return 0;
   }
```

3.6.2 Dateimanipulation

Listing 2: Dateimanipulation

```
main (argc, argv) int argc; char *argv[]; {
int fd, i; char read_buffer[RUNS]; struct stat fileStat;
for (i = 0; i < RUNS; ++i) read_buffer[i] = '\0';
if (argc != 2) { printf ("Missing file name, exiting\n"); return (-1); }
if ((fd = creat(argv[1], S_IRUSR)) == -1) { /* user has read rights */
   perror ("open failed"); return (-1);
} else printf ("file %s created, obtained file descriptor nr. %d\n", argv
    [1], fd);
if (chmod (argv[1], 0755) == -1) { perror ("chown failed"); return (-1); }
else printf ("mode of file %s changed to -rwxr-xr-x\n", argv[1]);
for (i = 0; i < RUNS; ++i) {
    write (fd, BUFFER, sizeof (BUFFER));
    write (fd, "\n", 1);
}
if (fstat (fd, &fileStat) == -1) { perror ("fstat failed"); return (-1); }
else {
    printf("Information for %s\n",argv[1]);
    printf("---------------------------\n");
    printf("File Size: \t\t%d bytes\n",(int) fileStat.st_size);
    printf("Number of Links: \t%d\n",fileStat.st_nlink);
    printf("File inode: \t\t%d\n",(int) fileStat.st_ino);
    printf("File Permissions: \t");
    printf( (S_ISDIR(fileStat.st_mode)) ? "d" : "-");
    printf( (fileStat.st_mode & S_IRUSR) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWUSR) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXUSR) ? "x" : "-");
    printf( (fileStat.st_mode & S_IRGRP) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWGRP) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXGRP) ? "x" : "-");
    printf( (fileStat.st_mode & S_IROTH) ? "r" : "-");
    printf( (fileStat.st_mode & S_IWOTH) ? "w" : "-");
    printf( (fileStat.st_mode & S_IXOTH) ? "x" : "-");
    printf("\n");
    printf("The file %s a symbolic link\n\n", (S_ISLNK(fileStat.st_mode)) ?
        "is" : "is not");
}
if (lseek (fd, 4000, SEEK_END) == -1) { /* file offset reset to EOF plus
    4000 */
   perror ("lseek failed");
   return (-1);
}
write (fd, "\n", 1);
for (i = 0; i < RUNS; ++i) {
    write (fd, BUFFER, sizeof (BUFFER));
    write (fd, "\n", 1);
}
if (close (fd) == -1) {
   perror ("close failed");
   return (-1);
}
}
```

3.6.3 Dateisystem-Baum

Listing 3: Dateisystem-Baum

```
void CleanUpOnError(int level)
{
    printf("Error encountered, cleaning up.\n");
    switch ( level )
       {
        case 1:
            printf("Could not get current working directory.\n");
            break;
        case 2:
            printf("Could not create file %s.\n",TEST_FILE);
            break;
        case 3:
            printf("Could not write to file %s.\n",TEST_FILE);
            close(FilDes);
            unlink(TEST_FILE);
            break;
        case 4:
            printf("Could not close file %s.\n",TEST_FILE);
            close(FilDes);
            unlink(TEST_FILE);
            break;
        case 5:
            printf("Could not make directory %s.\n",NEW_DIRECTORY);
            unlink(TEST_FILE);
            break;
        case 6:
            printf("Could not change to directory %s.\n",NEW_DIRECTORY);
            rmdir(NEW_DIRECTORY);
            unlink(TEST_FILE);
            break;
        case 7:
            printf("Could not create link %s to %s.\n",LinkName,InitialFile);
            chdir(PARENT_DIRECTORY);
            rmdir(NEW_DIRECTORY);
            unlink(TEST_FILE);
            break;
        case 8:
            printf("Could not open link %s.\n",LinkName);
            unlink(LinkName);
            chdir(PARENT_DIRECTORY);
            rmdir(NEW_DIRECTORY);
            unlink(TEST_FILE);
            break;
        case 9:
            printf("Could not read link %s.\n",LinkName);
            close(FilDes);
            unlink(LinkName);
            chdir(PARENT_DIRECTORY);
            rmdir(NEW_DIRECTORY);
            unlink(TEST_FILE);
            break;
        case 10:
            printf("Could not close link %s.\n",LinkName);
            close(FilDes);
            unlink(LinkName);
            chdir(PARENT_DIRECTORY);
```

```
57              rmdir(NEW_DIRECTORY);
                unlink(TEST_FILE);
                break;
            case 11:
                printf("Could not unlink link %s.\n",LinkName);
62              unlink(LinkName);
                chdir(PARENT_DIRECTORY);
                rmdir(NEW_DIRECTORY);
                unlink(TEST_FILE);
                break;
67          case 12:
                printf("Could not change to directory %s.\n",PARENT_DIRECTORY);
                chdir(PARENT_DIRECTORY);
                rmdir(NEW_DIRECTORY);
                unlink(TEST_FILE);
72              break;
            case 13:
                printf("Could not remove directory %s.\n",NEW_DIRECTORY);
                rmdir(NEW_DIRECTORY);
                unlink(TEST_FILE);
77              break;
            case 14:
                printf("Could not unlink file %s.\n",TEST_FILE);
                unlink(TEST_FILE);
                break;
82          default:
                break;
           }
       printf("Program ended with Error.\n"\
              "All test files and directories may not have been removed.\n");
87 }

   int main ()
   { /* Get and print the real user id with the getuid() function. */
       UserID = getuid();
92     printf("The real user id is %u. \n",UserID);

   /* Get the current working directory and store it in InitialDirectory. */
       if ( NULL == getcwd(InitialDirectory,BUFFER_SIZE) ) {
          perror("getcwd Error");
97        CleanUpOnError(1);
          return 0;
          }
       printf("The current working directory is %s. \n",InitialDirectory);

102 /* Create the file TEST_FILE for writing, if it does not exist.
       Give the owner authority to read, write, and execute. */
       FilDes = open(TEST_FILE, O_WRONLY | O_CREAT | O_EXCL, S_IRWXU);
       if ( -1 == FilDes ) {
          perror("open Error");
107       CleanUpOnError(2);
          return 0;
          }
       printf("Created %s in directory %s.\n",TEST_FILE,InitialDirectory);

112 /* Write TEST_DATA to TEST_FILE via FilDes */
       BytesWritten = write(FilDes,TEST_DATA,strlen(TEST_DATA));
       if ( -1 == BytesWritten ) {
          perror("write Error");
          CleanUpOnError(3);
```

```
117         return 0;
            }
        printf("Wrote %s to file %s.\n",TEST_DATA,TEST_FILE);

    /* Close TEST_FILE via FilDes */
122     if ( -1 == close(FilDes) ) {
            perror("close Error");
            CleanUpOnError(4);
            return 0;
            }
127     FilDes = -1;
        printf("File %s closed.\n",TEST_FILE);

    /* Make a new directory in the current working directory and
       grant the owner read, write and execute authority */
132     if ( -1 == mkdir(NEW_DIRECTORY, S_IRWXU) ) {
            perror("mkdir Error");
            CleanUpOnError(5);
            return 0;
            }
137     printf("Created directory %s in directory %s.\n",NEW_DIRECTORY,
             InitialDirectory);

    /* Change the current working directory to the
       directory NEW_DIRECTORY just created. */
        if ( -1 == chdir(NEW_DIRECTORY) ) {
142         perror("chdir Error");
            CleanUpOnError(6);
            return 0;
            }
        printf("Changed to directory %s/%s.\n",InitialDirectory,NEW_DIRECTORY);
147
    /* Copy PARENT_DIRECTORY to InitialFile and
       append  "/" and TEST_FILE to InitialFile. */
        strcpy(InitialFile,PARENT_DIRECTORY);
        strcat(InitialFile,"/");
152     strcat(InitialFile,TEST_FILE);

    /* Copy USER_ID to LinkName then append the
       UserID as a string to LinkName. */
        strcpy(LinkName, USER_ID);
157     sprintf(Buffer, "%d\0", (int)UserID);
        strcat(LinkName, Buffer);

    /* Create a link to the InitialFile name with the LinkName. */
        if ( -1 == link(InitialFile,LinkName) ) {
162         perror("link Error");
            CleanUpOnError(7);
            return 0;
            }
        printf("Created a link %s to %s.\n",LinkName,InitialFile);
167
    /* Open the LinkName file for reading only. */
        if ( -1 == (FilDes = open(LinkName,O_RDONLY)) ) {
            perror("open Error");
            CleanUpOnError(8);
172         return 0;
            }
        printf("Opened %s for reading.\n",LinkName);
```

```
   /* Read from the LinkName file, via FilDes, into Buffer. */
177    BytesRead = read(FilDes,Buffer,sizeof(Buffer));
       if ( -1 == BytesRead ) {
          perror("read Error");
          CleanUpOnError(9);
          return 0;
182       }
       printf("Read %s from %s.\n",Buffer,LinkName);
       if ( BytesRead != BytesWritten ) {
          printf("WARNING: the number of bytes read is "\
                 "not equal to the number of bytes written.\n");
187       }

   /* Close the LinkName file via FilDes. */
       if ( -1 == close(FilDes) ) {
          perror("close Error");
192       CleanUpOnError(10);
          return 0;
          }
       FilDes = -1;
       printf("Closed %s.\n",LinkName);
197
   /* Unlink the LinkName link to InitialFile. */
       if ( -1 == unlink(LinkName) ) {
          perror("unlink Error");
          CleanUpOnError(11);
202       return 0;
          }
       printf("%s is unlinked.\n",LinkName);

   /* Change the current working directory
207    back to the starting directory. */
       if ( -1 == chdir(PARENT_DIRECTORY) ) {
          perror("chdir Error");
          CleanUpOnError(12);
          return 0;
212       }
       printf("changing directory to %s.\n",InitialDirectory);

   /* Remove the directory NEW_DIRECTORY */
       if ( -1 == rmdir(NEW_DIRECTORY) ) {
217       perror("rmdir Error");
          CleanUpOnError(13);
          return 0;
          }
       printf("Removing directory %s.\n",NEW_DIRECTORY);
222
   /* Unlink the file TEST_FILE */
       if ( -1 == unlink(TEST_FILE) ) {
          perror("unlink Error");
          CleanUpOnError(14);
227       return 0;
          }
       printf("Unlinking file %s.\n",TEST_FILE);
       printf("Program completed successfully.\n");
       return 0;
232 }
```