

Systemprogrammierung

Jan Fässler

3. Semester (HS 2012)

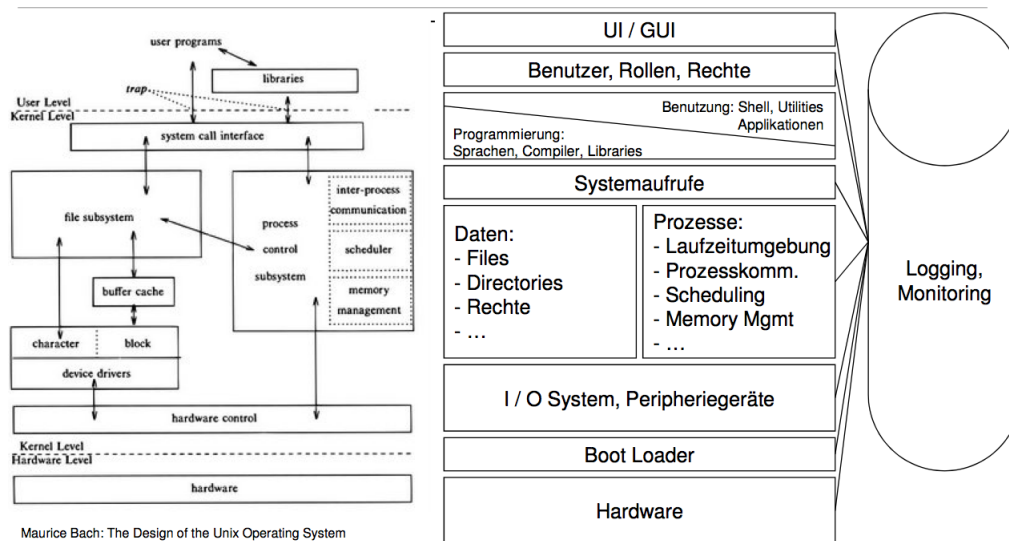
Inhaltsverzeichnis

1	Unix/Linux	1
1.1	Aufbau	1
1.2	Prozesse	1
1.2.1	Steuerungssystem	1
1.2.2	Aufbau	1
1.2.3	Kernel und User Mode	2
1.2.4	Zustände	2
1.3	Memory	2
1.3.1	Virtual Memory	2
1.3.2	Swapping	3
1.3.3	Paged Memory	3
1.3.4	Fehlerzustände	3
2	System Call Schnittstelle	4
2.1	Prozess-System	4
2.2	Datei-System	4
2.3	Directory Handling	5
2.4	Speicherverwaltung	5
2.5	Weitere	5
3	Dateisystem	6
3.1	Übersicht	6
3.2	Relevante Dateisystem Algorithmen	6
3.2.1	Pfadnahme zu inode	6
3.2.2	Gemeinsame Nutzung von Dateien	6
3.2.3	Datei Locking	6
3.2.4	Mount / Unmount	6
3.3	Allokation	7
3.3.1	inode	7
3.3.2	Datenblock	7
3.4	Synchronisation von Zugriffen auf das Dateisystem	7
3.5	Geräte	7
3.5.1	Einbindung	7
3.5.2	Gerätetreiber	7
3.5.3	Gerätespezialdateien	8
3.6	Beispiele	8
3.6.1	Verzeichnismanipulation	8
3.6.2	Dateimanipulation	9
3.6.3	Dateisystem-Baum	10
4	Prozess-Steuersystem	14
4.1	Process-System Calls	14
4.1.1	exec()	14
4.1.2	fork()	14
4.1.3	exit()	15
4.1.4	wait()	16
4.2	Threads	16
4.2.1	Prozesse versus Threads	16

4.2.2	System Calls	17
4.2.3	Beispiel	17
5	Pipes	19
5.1	Einleitung	19
5.2	Nutzung von Pipes	19
5.2.1	Variante 1:	19
5.2.2	Variante 2:	20
5.2.3	Variante 3:	20
5.3	Schliessen unbenutzter Pipe-Enden	20
5.4	Pseudocode	20
5.5	Dup Pipe Fork	21
5.6	popen() Bibliotheks- Funktion	21

1 Unix/Linux

1.1 Aufbau

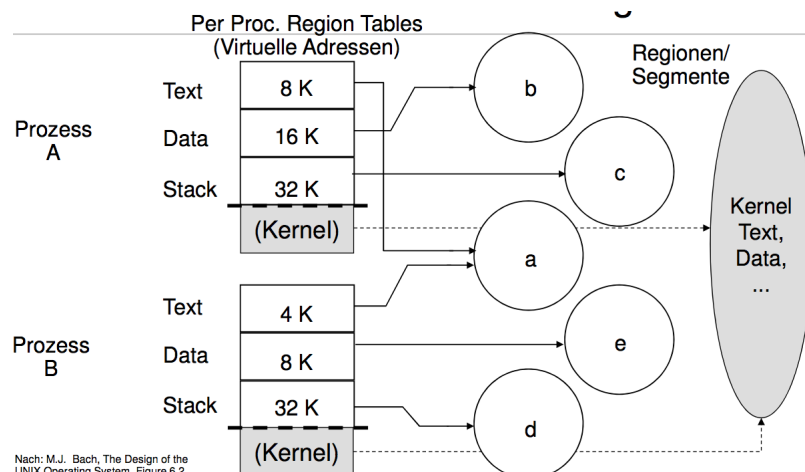


1.2 Prozesse

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

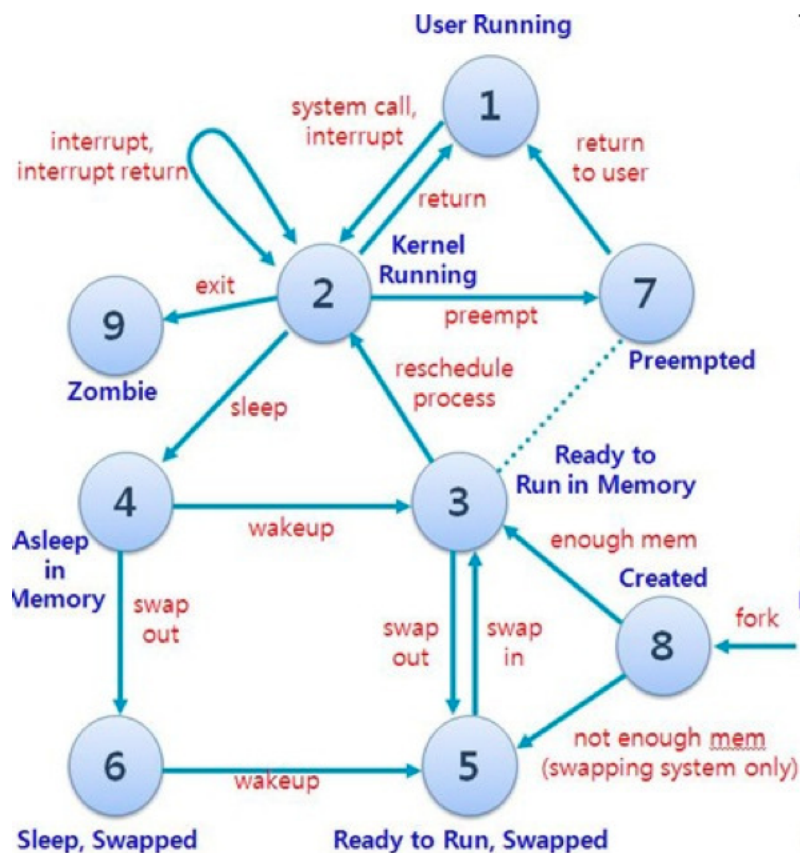
- Ein Prozess hat mindestens zwei Ausführungsmodi:

User Mode Es wird der normale Programmcode ausgeführt.

Kernel 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 abgesichert.

1.2.4 Zustände

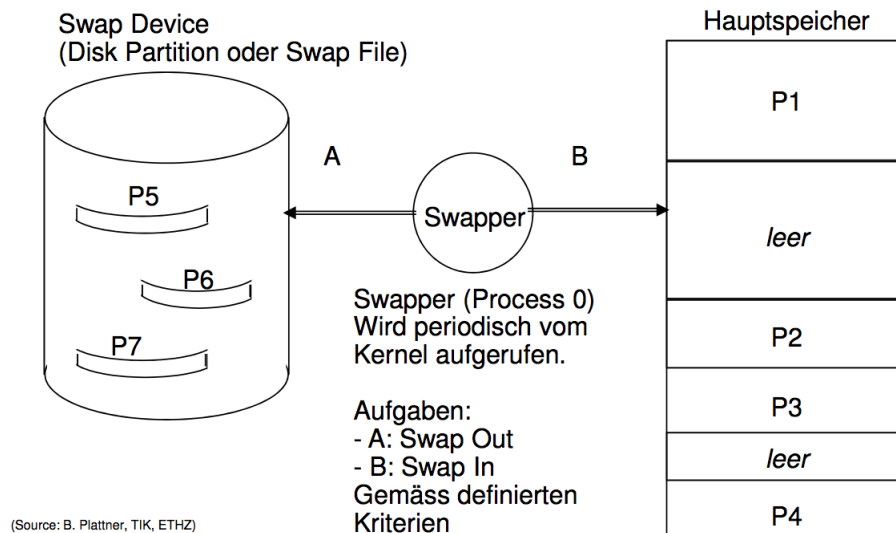


1.3 Memory

1.3.1 Virtual Memory

- Erweiterung des Hauptspeichers pro System (mehr Prozesse im System als Speicher verfügbar), oder pro Prozess (einzelner Prozess grösser als verfügbarer Hauptspeicher).
- Systematische Abstraktion für systemspezifische Overlay-Techniken.
- Organisation des Hauptspeichers in gleich grosse, einheitlich adressierbare Einheiten.

1.3.2 Swapping

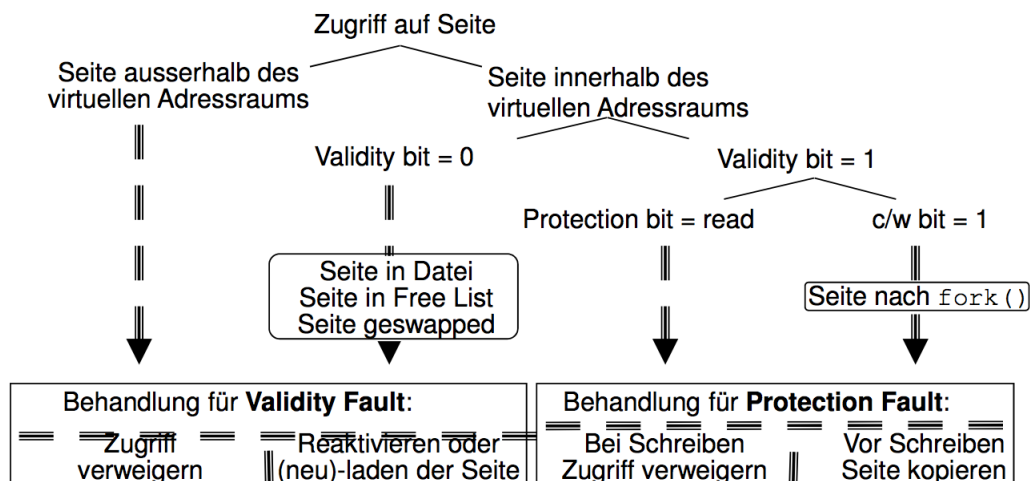


1.3.3 Paged Memory

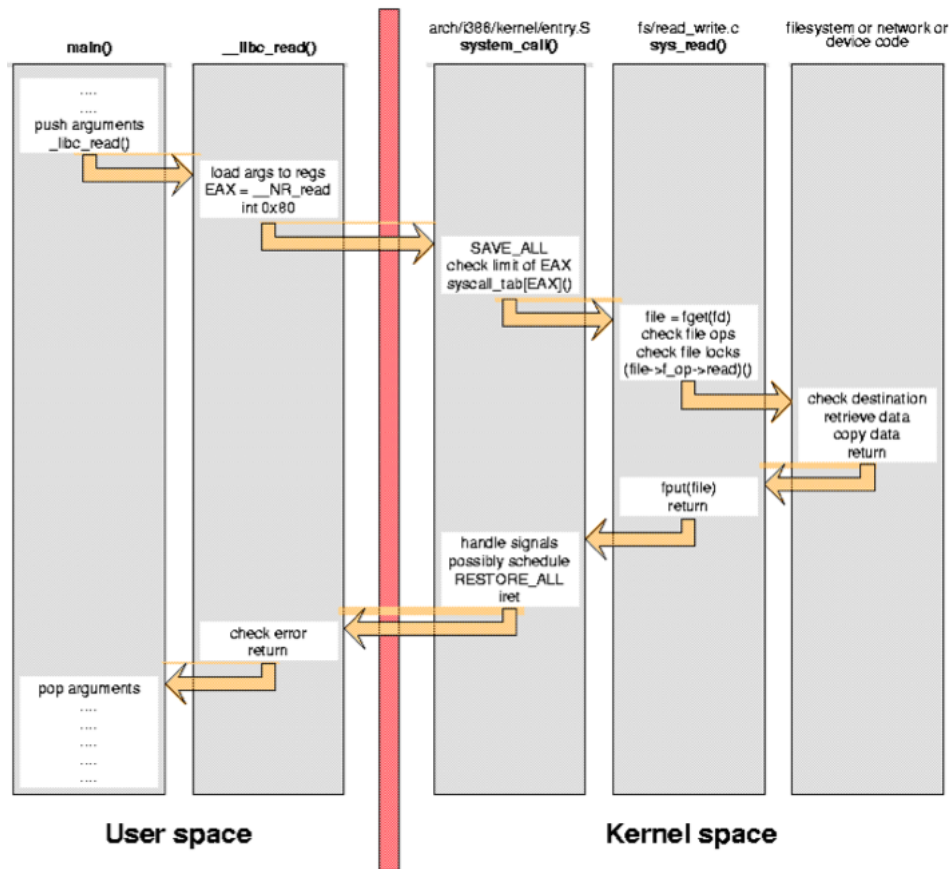
Design und Layout der Datenstrukturen für die seiten-orientierte Speicherverwaltung variieren zwischen Unix- und Linux-Varianten und sind zudem von den Hardware- Eigenschaften abhängig. Linux verwendet zum Beispiel eine 3-stufige Seitentabelle:

- 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 Speicherort

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-/Rückwärtsbewegung
ioctl() Kontrollieren der Eigenschaften
dup() Duplizieren eines Dateideskriptors
chown, chmod, umask Zugriffsrechte
chdir() Navigation im Dateisystem

2.3 Directory Handling

opendir() Öffnen eines Verzeichnisses
readdir() Lesen eines Verzeichnisses
writedir() Schreiben eines Verzeichnisses
closedir() Schliessen eines Verzeichnisses

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

Returns File Descriptor	Use of Name Lookup	Assign Inodes	File At-tributes	File-I/O	File System Structure	Tree Manipulation
open create dup pipe close	open stat create link chdir unlink chroot mknod chown mount chmod unmount	create mknod link unlink	chown chmod stat	read write Lseek mmap	mount unmount	chdir chown

Lower Level File System Algorithms			
Name to inode	Allocate / free inode	Allocate / free blocks Memory-mapped I/O	
Get / put inode			
Buffer Cache Delayed write / Read ahead			

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 dienen - 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 enthält 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 werden jedoch nicht aktualisiert.

3.5 Geräte

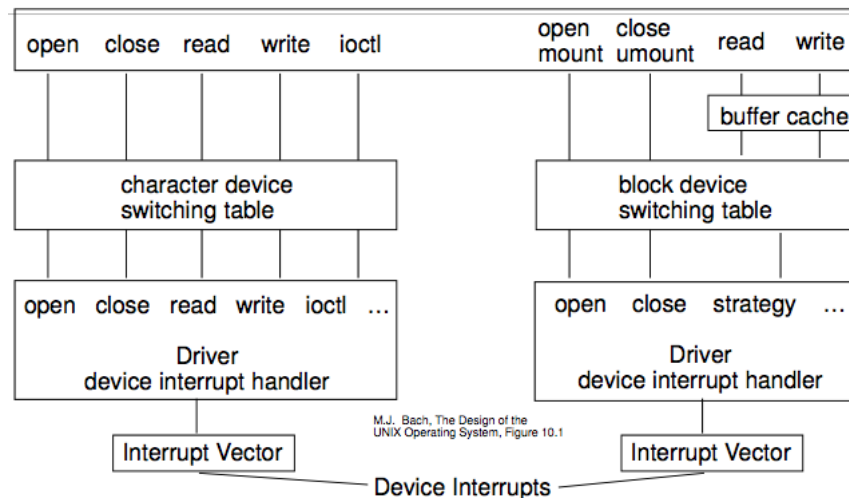
3.5.1 Einbindung

- 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

```
1 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 */
6        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) {
11        write (fd, BUFFER, sizeof (BUFFER));
        write (fd, "\n", 1);
    }
    if (fstat (fd, &fileStat) == -1) { perror ("fstat failed"); return (-1); }
    else {
16        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);
21        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" : "-");
26        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" : "-");
31        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 */
36        perror ("lseek failed");
        return (-1);
    }
    write (fd, "\n", 1);
    for (i = 0; i < RUNS; ++i) {
41        write (fd, BUFFER, sizeof (BUFFER));
        write (fd, "\n", 1);
    }
    if (close (fd) == -1) {
        perror ("close failed");
46        return (-1);
    }
}
```

3.6.3 Dateisystem-Baum

Listing 3: Dateisystem-Baum

```
void CleanUpOnError(int level)
2 {
    printf("Error encountered, cleaning up.\n");
    switch ( level )
    {
        case 1:
7         printf("Could not get current working directory.\n");
            break;
        case 2:
            printf("Could not create file %s.\n",TEST_FILE);
            break;
12        case 3:
            printf("Could not write to file %s.\n",TEST_FILE);
            close(FilDes);
            unlink(TEST_FILE);
            break;
17        case 4:
            printf("Could not close file %s.\n",TEST_FILE);
            close(FilDes);
            unlink(TEST_FILE);
            break;
22        case 5:
            printf("Could not make directory %s.\n",NEW_DIRECTORY);
            unlink(TEST_FILE);
            break;
        case 6:
27        printf("Could not change to directory %s.\n",NEW_DIRECTORY);
            rmdir(NEW_DIRECTORY);
            unlink(TEST_FILE);
            break;
        case 7:
32        printf("Could not create link %s to %s.\n",LinkName,InitialFile);
            chdir(PARENT_DIRECTORY);
            rmdir(NEW_DIRECTORY);
            unlink(TEST_FILE);
            break;
37        case 8:
            printf("Could not open link %s.\n",LinkName);
            unlink(LinkName);
            chdir(PARENT_DIRECTORY);
            rmdir(NEW_DIRECTORY);
42        unlink(TEST_FILE);
            break;
        case 9:
            printf("Could not read link %s.\n",LinkName);
            close(FilDes);
47        unlink(LinkName);
            chdir(PARENT_DIRECTORY);
            rmdir(NEW_DIRECTORY);
            unlink(TEST_FILE);
            break;
52        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 }

```

4 Prozess-Steuersystem

4.1 Process-System Calls

4.1.1 exec()

Ausführen eines neuen Programms in einer vorhandenen Prozesshülle

Listing 4: exec() Beispiel

```
(void) printf ("process started\n");
2  execl ("/bin/date", "/bin/date", (char *) 0);
    (void) printf ("This should not happen\n");
```

Listing 5: exec() Pseudocode

```
algorithm exec
2 input: (1) file name
          (2) parameter list
          (3) environment variables list
output: none {
  get file inode (algorithm namei);
7 verify file executable, user has permission to execute;
  read file headers, check that it is a load module;
  copy exec parameters from old address space to system space; for (every
    region attached to process)
  detach all old regions (algorithm detachreg);
  for (every region attached to process)
12 detach all old regions (algorithm detachreg);
  for (every region specified in load module) {
    allocate new regions (algorithm allocreg);
    attach the regions (algorithm attachreg);
    load region into memory if appropriate (algorithm loadreg);
17 }
  copy exec parameters into new user stack region;
  special processing for setuid programs, tracing;
  initialize user register save area for return to user mode;
  release inode of file (algorithm iput);
22 }
```

4.1.2 fork()

Der Aufruf *fork()* erstellt einen neuen Prozess als Seiten- effekt und gibt einen numerischen Return-Code zurück:

- Der neue Prozess (Kind) erhält eine 0.
- Der aufrufende Prozess (Eltern) erhält entweder -1 im Fehlerfall oder einen Wert > 0 , der der Prozess-ID des Kindprozesses entspricht.

Der Kindprozess ist eine identische Kopie des Elternprozesses (inkl. offenen Dateien, Speicher etc.), mit Ausnahme der Prozess- ID, der parent process ID und der Ressourcenverbrauchszähler. *fork()* hat im Unterschied zur Prozesserschöpfung in anderen Betriebssystemen keine Parameter.

Listing 6: fork() Beispiel

```

switch (fork()) {
3   case -1:(void) printf ("fork failed\n");
       break;
       case 0: (void) printf ("child executing\n");
           break;
       default: (void) printf ("parent executing\n");
8       break;
}

```

Listing 7: fork() Pseudocode

```

1  algorithm fork
    input: none
    output: to parent process, child PID number; to child, 0 {
        check for available kernel resources;
        get free proc table slot, unique PID number;
6   check that user not running too many processes;
        mark child state "being created";
        copy data from parent proc table slot to new child slot;
        incr. counts on curr. directory inode and changed root (if appl.);
        increment open file counts in file table;
11  make copy of parent context (u_area, text, data, stack) in memory;
        push dummy system level context layer on child system level context;
        dummy context contains data allowing child process to recognize
        itself, and start running from here when scheduled;
        if (executing process is parent process) {
16  change child state to "ready to run";
            return (child ID); /* from system to user */
        } else { /* executing process is the child process */
            initialize u_area timing fields;
            return (0); /* to user */
21  }
    }

```

4.1.3 exit()

Der Aufruf von *exit()* terminiert einen Prozess. Alle Ressourcen (Speicher, offene Dateien, ...) werden freigegeben, nur der Eintrag in der Prozesstabelle bleibt bestehen (Zombie-Prozess), bis der Elternprozess den Rückgabewert abrufen.

Ein Prozess kann *exit()* selbst aufrufen, oder das Betriebssystem ruft *_exit()* auf, wenn ein Prozess zwangsweise terminiert werden muss.

Listing 8: exit() Pseudocode

```

algorithm exit
input: return code for parent proc
3 output: to parent process, child PID number; to child, 0 {
    ignore all signals;
    if (process group leader with associated control terminal) {
        send hangup signal to all members of process group;
        reset process group for all members to 0;
8    }
    close all open files (internal version of algorithm iclose);
    release current directory (algorithm iput);
    release current (changed) root, if exists (algorithm iput);
    free regions, memory associated with process (algorithm freereg);
13 write accounting record;

```

```

    make process state zombie;
    assign parent proc ID of all child processes to be init proc (1);
    if any children were zombie, send death of child signal to init;
    send death of child signal to parent process;
18 context switch;
}

```

4.1.4 wait()

Der Aufruf von *wait()* suspendiert einen Prozess von der CPU (Kontextwechsel) bis ein Kindprozess terminiert oder eine Ausnahme (z.B. stoppen) signalisiert.

wait() liefert dem Elternprozess dann den Exit Code des Kindprozesses zurück, oder einen Indikator, warum der Kindprozess vom Betriebssystem terminiert wurde.

Wenn der Prozess beim Aufruf von *wait()* keine Kindprozesse hat, wird er nicht schlafen gelegt. In einigen Unix-Varianten liefert der Aufruf von *wait()* den Exit Code des ersten terminierten Kindprozesses zurück (d.h. *wait()* muss mehrfach aufgerufen werden, wenn mehrere Kindprozesse vorhanden sind), in anderen Varianten kehrt *wait()* erst zurück, wenn der letzte Kindprozess terminiert ist, und liefert auch dessen Prozess-ID zurück.

Listing 9: wait() Beispiel

```

1 int main(int argc, char *argv[]) {
    pid_t cpid, w;
    int status;
    cpid = fork();
    if (cpid == -1) { perror("fork"); exit(EXIT_FAILURE); }
6  if (cpid == 0) { /* Code executed by child */
    printf("Child PID is %ld\n", (long) getpid());
    if (argc == 1) pause(); /* Wait for signals */
    _exit(atoi(argv[1]));
} else { do { /* Code executed by parent */
11  w = waitpid(cpid, &status, WUNTRACED | WCONTINUED);
    if (w == -1) { perror("waitpid");
    exit(EXIT_FAILURE); }
    if (WIFEXITED(status)) {
        printf("exited, status=%d\n", WEXITSTATUS(status));
16  } else if (WIFSIGNALED(status)) {
        printf("killed by signal %d\n", WTERMSIG(status));
    } else if (WIFSTOPPED(status)) {
        printf("stopped by signal %d\n", WSTOPSIG(status));
    } else if (WIFCONTINUED(status)) {
21  printf("continued\n"); }
    } while (!WIFEXITED(status) && !WIFSIGNALED(status));
    exit(EXIT_SUCCESS);
}
}

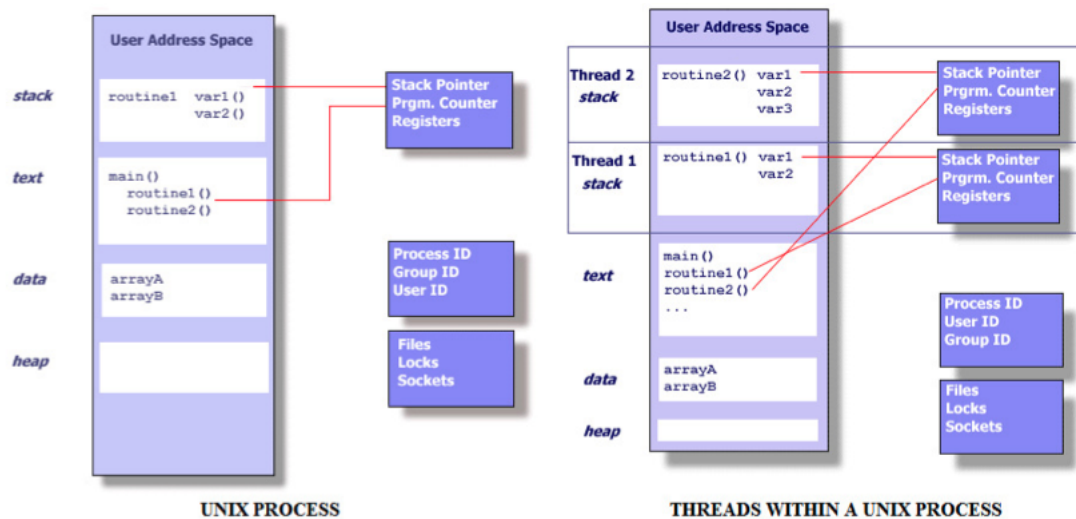
```

4.2 Threads

4.2.1 Prozesse versus Threads

- Kontextwechsel sind eine schwere Operation mit viel Verarbeitungsaufwand durch den Kernel.
- Da viele Unix-Prozesse I/O-intensiv sind, verbringen sie die meiste Laufzeit mit Warten, dadurch erhöht sich die Anzahl von Kontextwechseln im System.

- Neuere Unix-/Linux-Systeme unterstützten mehr als einen parallelen Ausführungspfad innerhalb eines Prozesses (multi-threading) → es muss kein Kontextwechsel vorgenommen werden, um eine andere Aktivität zu starten.



4.2.2 System Calls

`pthread_create()` Erzeugung eines Threads

`pthread_exit()` Selbst-Termination eines Threads

`pthread_cancel()` Fremd-Termination eines Threads

`pthread_join()` Warten auf Termination und Exit-Code eines Threads

`pthread_detach()` Kein Warten auf einen Thread

`pthread_self()` Ausgabe der unique thread-ID

`pthread_equal()` Vergleich zweier thread-IDs

`pthread_once()` Ausführung einer Initialisierungsroutine

4.2.3 Beispiel

Listing 10: Threads

```

1  int main() {
      pthread_t thread_id[NTHREADS]; int i, j;

      for(i=0; i < NTHREADS; i++) {
5         pthread_create( &thread_id[i], NULL, thread_function, NULL );
      }
      for(j=0; j < NTHREADS; j++) {
          pthread_join( thread_id[j], NULL);
      }
10     /* Now that all threads are complete I can print the final result.      */
       /* Without the join I could be printing a value before all the threads */
       /* have been completed.                                                */
       printf("Final counter value: %d\n", counter);

```

```
    }  
15 void *thread_function(void *dummyPtr) {  
    printf("Thread number %ld\n", pthread_self());  
    pthread_mutex_lock( &mutex1 );  
    counter++;  
    pthread_mutex_unlock( &mutex1 );  
20 }
```

5 Pipes

5.1 Einleitung

- Unidirektionaler Kommunikationskanal zwischen verwandten Prozessen (d.h. Prozessen, die eine direkte oder indirekte / vererbte Eltern-Kind- Beziehung nach einem *fork()* haben).
- Direkt nutzbar für Input/Output Umleitung in der Shell.
- Wird innerhalb des Prozesses als 2 Dateideskriptoren repräsentiert, um Kompatibilität mit Standard-I/O mit *read()* und *write()* zu erlauben.
- Volatiler Dateninhalt für kleine Datenmengen - d.h. keine permanente Speicherung im Dateisystem - der Inhalt einer Pipe geht verloren, wenn das System herunterfährt.
- Keine Struktur der Daten in der Pipe (Bytestrom).
- Blockierendes, nicht-atomares Schreiben
- Blockierendes, destruktives Lesen
- Sonderform named pipe mit permanenter Repräsentanz im Dateisystem

5.2 Nutzung von Pipes

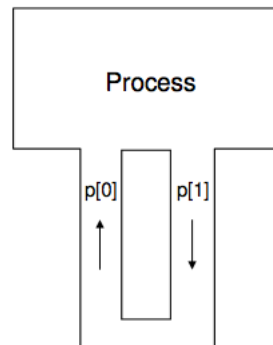
5.2.1 Variante 1:

```
#include <stdio.h>
#define MSGSIZE 16
char *msg1 = "hello, world #1";
char *msg2 = "hello, world #2";
char *msg3 = "hello, world #3";

main () {
    char inbuf[MSGSIZE];
    int p[2], j;

    if (pipe (p) < 0) {
        perror ("pipe call");
        exit (1);
    }
    write (p[1], msg1, MSGSIZE);
    write (p[1], msg2, MSGSIZE);
    write (p[1], msg3, MSGSIZE);

    for (j = 0; j < 3; j++) {
        read (p[0], inbuf, MSGSIZE);
        printf ("%s\n", inbuf);
    }
    exit (0);
}
```

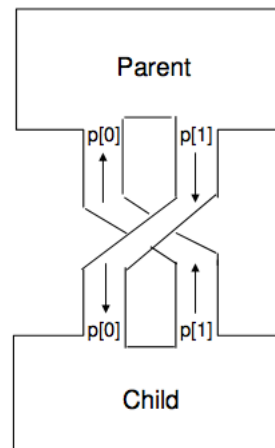


Haviland/Salama, UNIX System Programming, page 143 und Figure 6.1

5.2.2 Variante 2:

```
#include <stdio.h>
#define MSGSIZE 16
char *msg1 = "hello, world #1";
char *msg2 = "hello, world #2";
char *msg3 = "hello, world #3";
main () {
    char inbuf[MSGSIZE];
    int p[2], j, pid;
    if (pipe (p) < 0) {
        perror ("pipe call");
        exit (1); }
    if ((pid = fork()) < 0) {
        perror ("fork call");
        exit (2); }
    if (pid > 0) {
        write (p[1], msg1, MSGSIZE);
        write (p[1], msg2, MSGSIZE);
        write (p[1], msg3, MSGSIZE);
        wait ((int *)0); }
    if (pid == 0) {
        for (j = 0; j < 3; j++) {
            read (p[0], inbuf, MSGSIZE);
            printf ("%s\n", inbuf); } }
    exit (0); }
```

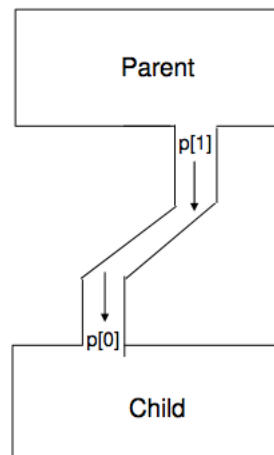
Haviland/Salama, UNIX System Programming, page 144/145 und Figure 6.2



5.2.3 Variante 3:

```
char *msg1 = "hello, world #1";
char *msg2 = "hello, world #2";
char *msg3 = "hello, world #3";
main () {
    char inbuf[MSGSIZE];
    int p[2], j, pid;
    if (pipe (p) < 0) {
        perror ("pipe call");
        exit (1); }
    if ((pid = fork()) < 0) {
        perror ("fork call");
        exit (2); }
    if (pid > 0) {
        close (p[0]);
        write (p[1], msg1, MSGSIZE);
        write (p[1], msg2, MSGSIZE);
        write (p[1], msg3, MSGSIZE);
        wait ((int *)0); }
    if (pid == 0) {
        close (p[1]);
        for (j = 0; j < 3; j++) {
            read (p[0], inbuf, MSGSIZE);
            printf ("%s\n", inbuf); } }
    exit (0); }
```

Haviland/Salama, UNIX System Programming, page 145/146 und Figure 6.3



5.3 Schliessen unbenutzter Pipe-Enden

1. Saubere Programmierung
2. Schutz gegen copy/paste Fehler bei der Software-Entwicklung
3. Auf älteren Unix-Systemen begrenzte Anzahl verfügbarer Dateideskriptoren pro Prozess
4. Das Schliessen des letzten Lese- oder Schreib-Endes einer Pipe popen() im Betrieb hat Auswirkungen auf die Prozesssynchronisation.

5.4 Pseudocode

Listing 11: pipe() Pseudocode

```
algorithm pipe
input: none
output: file descriptor, write file descriptor
```

```

{
5  assign new inode from pipe device (algorithm ialloc);
  /* Since System V R4 dedicated FIFO file system (fifofs) */
  allocate file table entry for reading, another for writing;
  initialize file table entries to point to new inode;
  allocate user file descriptor for reading, another for writing,
10  initialize to point to respective file table entries;
  set inode reference count to 2;
  initialize count of inode readers, writers to 1;
}

```

5.5 Dup Pipe Fork

Listing 12: Dup Pipe Fork

```

#include <string.h>
2 #include <stdio.h>

char string[] = "hello world";

main () {
7  int count, i;
  int to_par[2], to_chil[2];
  char buf[256];

  pipe (to_par);
12  pipe (to_chil);

  if (fork() == 0) {
    close (0); dup (to_chil[0]);
    close (1); dup (to_par[1]);
17  close (to_par[1]); close (to_chil[0]);
    close (to_par[0]); close (to_chil[1]);
    for (;;) {
      if ((count = read (0, buf, sizeof (buf))) == 0)
        exit (0);
22  else
      fprintf (stderr, "child read %s\n", buf);
      write (1, buf, count);
      fprintf (stderr, "child wrote %s\n", buf);
    }
27  }
  /* parent process executes here */
  close (1); dup (to_chil[1]);
  close (0); dup (to_par [0]);
  close (to_chil[1]); close (to_par[0]);
32  close (to_chil[0]); close (to_par[1]);

  for (i = 0; i < 15; i++) {
    write (1, string, strlen (string));
    fprintf (stderr, "parent wrote %s\n", buf);
37  read (0, buf, sizeof (buf));
    fprintf (stderr, "parent read %s\n", buf);
  }
}

```

5.6 popen() Bibliotheks- Funktion

Listing 13: popen() Beispiel

```

#include <stdio.h>

#define MAXLEN      255    /* maximum filename length */
#define MAXCMD      100    /* maximum length of command */
5 #define MAXLINE 100      /* maximum number of files */
#define ERROR      (-1)
#define SUCCESS     0

getlist (namepart, dirnames, maxnames)
10 char *namepart; /* additional part of ls command */
   char dirnames[][MAXLEN+1]; /* to hold file names */
   int  maxnames;    /* max. no. of file names */
{
   char *strcpy(), *strncat(), *fgets();
15 char cmd[MAXCMD+1], inline[MAXLEN+2];
   int  i;
   FILE *lsf, *popen();

   strcpy (cmd, "ls "); /* first build command */
20
   /* add additional part of command */
   if (namepart != NULL) strncat (cmd, namepart, MAXCMD - strlen (cmd));

   /* start up command */
25 if (( lsf = popen (cmd, "r")) == NULL) return (ERROR);

   for (i=0; i < maxnames; ++i) {
       /* read a line */
       if (fgets (inline, MAXLEN+2, lsf) == NULL) break;
30
       /* remove newline */
       if (inline[strlen (inline) -1] == '\n') inline [strlen (inline) -1] =
           '\0';

       strcpy (&dirnames[i][0], inline);
35 }
   if (i < maxnames) dirnames [i][0] = '\0';
   pclose (lsf);
   return (SUCCESS);
}
40
main () {
   char namebuf[MAXLINE][MAXLEN+1]; int i = 0;
   getlist ("*.c", namebuf, MAXLINE);
   while (namebuf[i][0] != '\0') printf ("%s\n", namebuf[i++]);
45 }

```