Systemprogrammierung

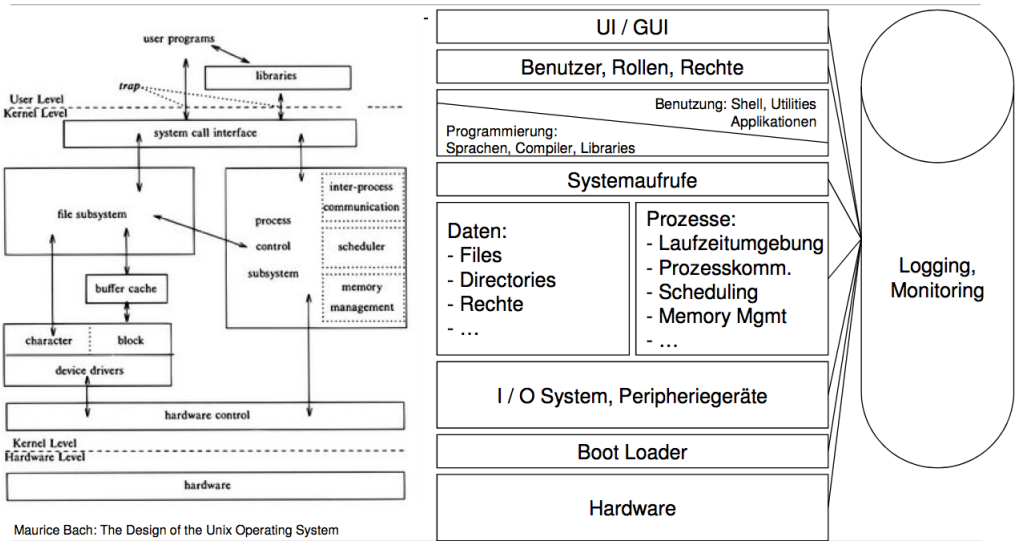
Jan Fässler

3. Semester (HS 2012)

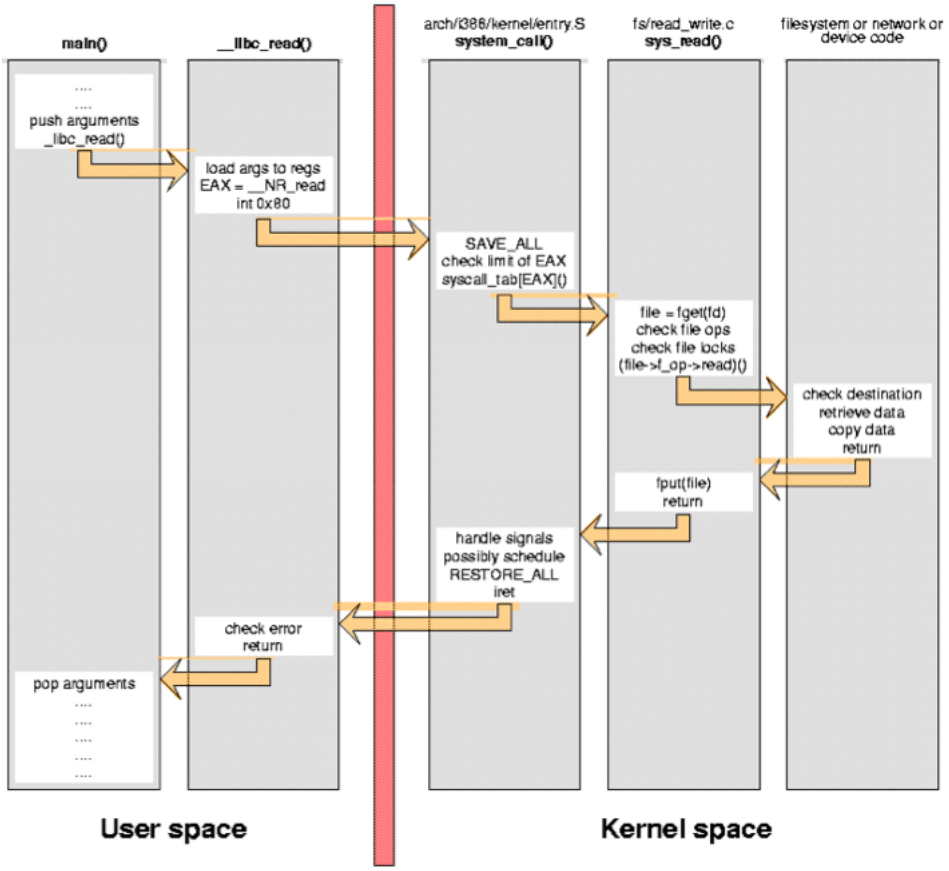
Inhaltsverzeichnis

1 Einleitung

1.1 UNIX-Aufbau

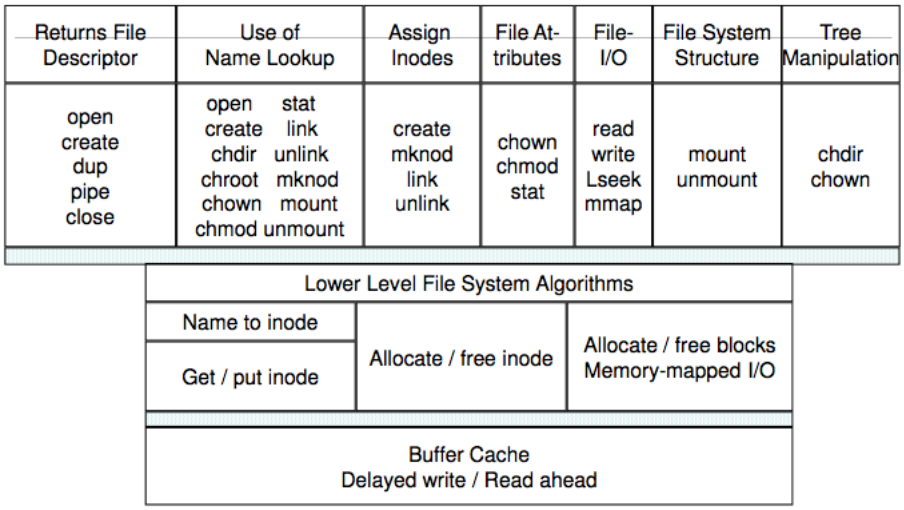


1.2 System Call Schnittstelle



2 Dateisystem

2.1 Übersicht



2.2 System Calls

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 Relevante Dateisystem Algorithmen

2.4.1 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.

2.4.2 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.

2.4.3 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.

2.4.4 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.

2.5 Allokation

2.5.1 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.

2.5.2 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).

2.6 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.

2.7 Geräte

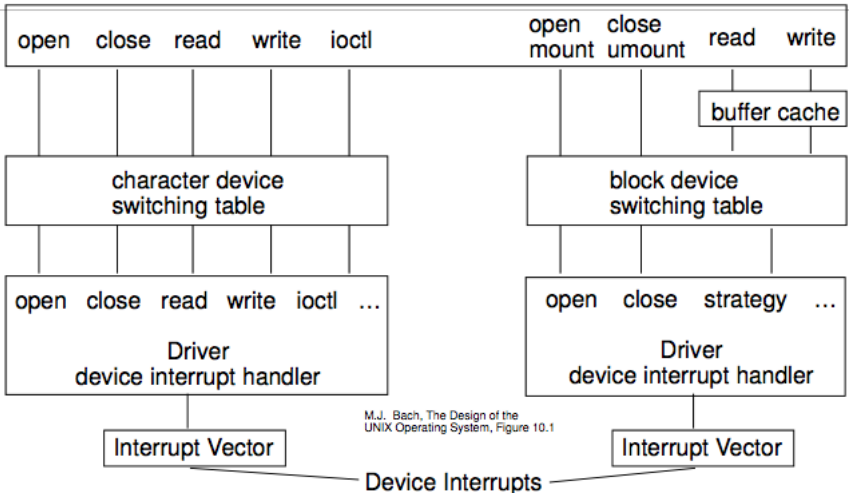
2.7.1 Einbindung

- 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)

2.7.2 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.

2.7.3 Gerätespezialdateien



2.8 Beispiele

2.8.1 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 */
        pDirEnt = readdir( pDIR );
        while ( pDirEnt != NULL ) {
            printf( "%s\n", pDirEnt->d_name );
            pDirEnt = readdir( pDIR );
        }
        /* Release the open directory */
        closedir( pDIR );
        return 0;
    }
```

2.8.2 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) {
41      write (fd, BUFFER, sizeof (BUFFER));
        write (fd, "\n", 1);
  }
  if (close (fd) == -1) {
     perror ("close failed");
46   return (-1);
  }
  }
```

---

3 Prozesse

3.1 Einleitung

3.1.1 Aufgaben an ein Prozesssteuerungssystem

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

3.1.2 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.

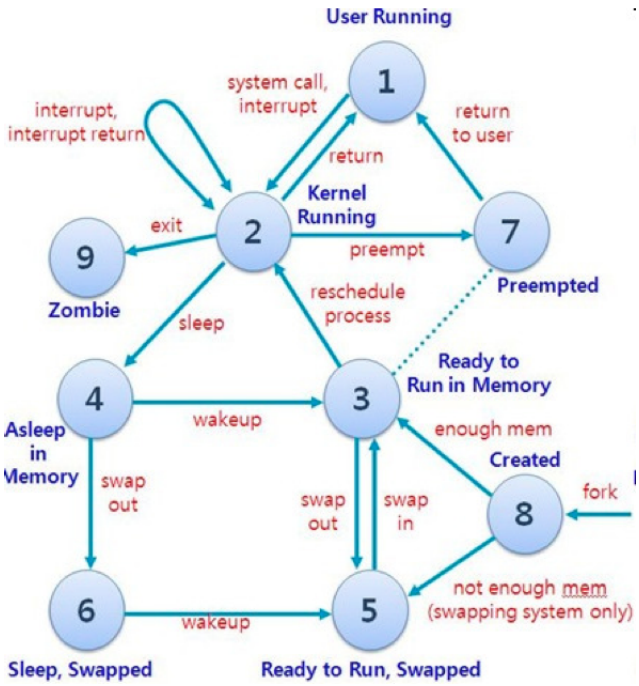
3.1.3 Prozesskontext

Benutzer Daten des Prozesses im zugewiesenen Adressraum

Hardware Basis- und Grenzregister, Befehlszählregister, Akkumulator & Seitentabelle

System Prozessnummer, geöffneten Dateien, Eltern- oder Kindprozesse & Prioritäten

3.1.4 Zustände



3.2 System Calls

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

3.2.1 nötige System Calls für Multiuser-Betriebssystem

File System: open(), read(), write(), close(), unlink(), creat()

Prozess Steuer: fork(), sleep(), exec(), kill(), exit(), wait()

Inter Prozess Kommunikation: pipe(), sync(), socket()

Memory Management: malloc(), free()

3.2.2 exec()

Ausführen eines neuen Programms in einer vorhandenen Prozesshülle

Listing 3: exec() Beispiel

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

3.2.3 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 Prozesserzeugung in anderen Betriebssystemen keine Parameter.

Listing 4: fork() Beispiel

```
2 switch (fork()) {
    case -1:(void) printf ("fork failed\n");
      break;
    case 0: (void) printf ("child executing\n");
```

```
      break;
7   default: (void) printf ("parent executing\n");
    break;
  }
```

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

3.2.5 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 zuru?ck.

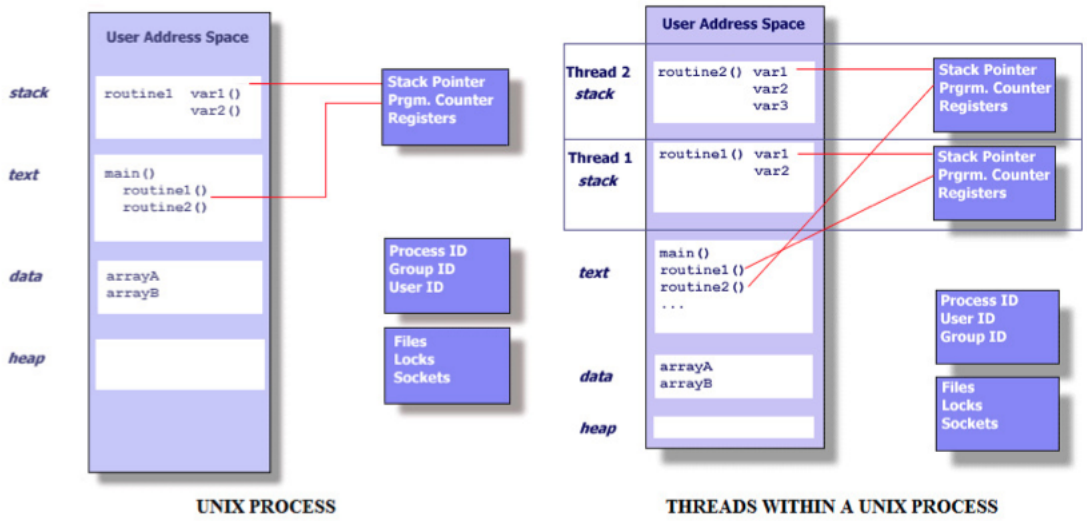
Listing 5: wait() Beispiel

```
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 Threads

4.1 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 parallellen Ausführungspfad innerhalb eines Prozesses (multi-threading) → es muss kein Kontextwechsel vorgenommen werden, um eine andere Aktivität zu starten.



4.2 System Calls

pthread_create() Erzeugung eines Threads

pthread_exit() Selbst-Termination eines Threads

pthread_cancel() Fremd-Termination eines Threads

pthread_join() Wartena uf 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 Initialisierungsroutinge

4.3 Beispiel

Listing 6: Threads

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

```
    for(i=0; i < NTHREADS; i++) {
       pthread_create( &thread_id[i], NULL, thread_function, NULL );
    }
    for(j=0; j < NTHREADS; j++) {
       pthread_join( thread_id[j], NULL);
    }
    /* 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);
}
void *thread_function(void *dummyPtr) {
    printf("Thread number %ld\n", pthread_self());
    pthread_mutex_lock( &mutex1 );
    counter++;
    pthread_mutex_unlock( &mutex1 );
}
```

5 Pipes

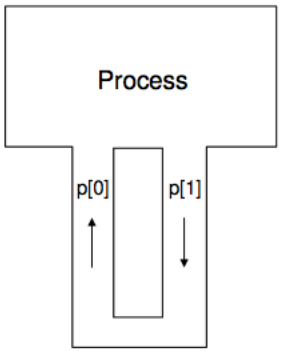
5.1 Einleitung

- 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 Kompatibilita?t mit Standard-I/O mit *read()* and *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:

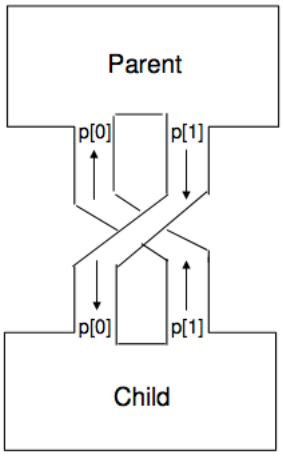
```
#include <stdio.h>
#define MSGSIZE 16
char *msg1 = "hello, world #1";
char *msg2 = "hello, world #2";
char *msg3 = "hello, world #3";

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:

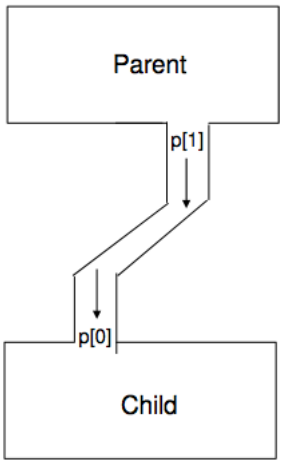
```
#include <stdio.h>
#define MSGSIZE 16
char *msg1 = "hello, world #1";
char *msg2 = "hello, world #2";
char *msg3 = "hello, world #3";
main () {
     char inbuf[MSGSIZE];
     int p[2], j, pid;
     if (pipe (p) < 0) {
          perror ("pipe call");
          exit (1); }
     if ((pid = fork()) < 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 7: pipe() Pseudocode

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

```
{
  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,
  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 8: Dup Pipe Fork

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

char string[] = "hello world";

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

   pipe (to_par);
   pipe (to_chil);

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

5.6 popen() Bibliotheks- Funktion

Listing 9: popen() Beispiel

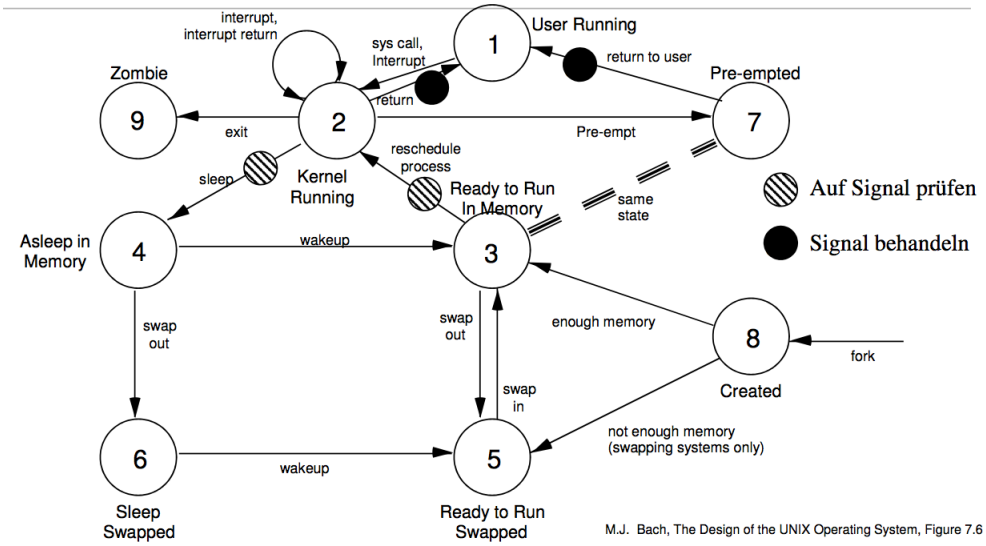
```
#include <stdio.h>

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

getlist (namepart, dirnames, maxnames)
  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();
  char cmd[MAXCMD+1], inline[MAXLEN+2];
  int  i;
  FILE *lsf, *popen();

  strcpy (cmd, "ls "); /* first build command */

  /* add additional part of command */
  if (namepart != NULL) strncat (cmd, namepart, MAXCMD - strlen (cmd));

  /* start up command */
  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;

      /* remove newline */
      if (inline[strlen (inline) -1] == '\n') inline [strlen (inline) -1] =
          '\0';

      strcpy (&dirnames[i][0], inline);
  }
  if (i < maxnames) dirnames [i][0] = '\0';
  pclose (lsf);
  return (SUCCESS);
}

main () {
  char  namebuf[MAXLINE][MAXLEN+1]; int i = 0;
  getlist ("*.c", namebuf, MAXLINE);
  while (namebuf[i][0] != '\0') printf ("%s\n", namebuf[i++]);
}
```

6 Signale

6.1 Einleitung

Ursprünglich entwickelt, um Ausnahmesituationen und Fehlverhalten von Prozessen anzuzeigen (Division durch Null, Zugriffsschutzverletzungen usw.), die zu einer Prozessbeendigung durch den Kernel führen. Später erweitert, um asynchrone Prozesskommunikation zu erlauben, und um nicht-kritische Bedingungen (z.B. Terminieren eines Kindprozesses, Starten/Stoppen eines Prozesses für das Debugging, I/O auf einem Datei- deskriptor usw.) anzuzeigen. Mechanismus wird benutzt vom Kernel (als Sender), vom Benutzer (als Sender, z.B. Cntl-C) und von Prozessen (als Sender mittels kill() oder als Empfänger).

6.2 Signale und Prozesse



6.3 System Calls

kill(signum) Sendet ein Signal

signal(signum,handler) Setzt einen neuen Signal Handler und gibt den alten zurück.

alarm(seconds) Setzt ein Alarmsignal nach einer bestimmten Anzahl Sekunden ab.

6.4 Eltern- / Kind-Prozess Signalisierung

Wenn ein Prozess *fork()* aufruft, bleiben alle schon definierten Signalaktionen (Default, Ignore, Handler) im Eltern- und im Kindprozess intakt. Wenn ein Prozess mit pendenten, aber noch nicht behandelten Signalen *fork()* aufruft, erbt der Kindprozess die pendenten Signale (d.h. Signalduplikation). Wenn ein Prozess *exec()* aufruft, werden alle Signal Handler wieder auf ihre Default-Aktion zurückgesetzt, da die Adresse des Signal Handlers im Prozessadressraum auf anderen Code zeigt und der Rumpf der Handling Prozedur im Textsegment deallozііert / überschrieben wurde.

6.5 Alarm Signal

Listing 10: Alarm Signal

```
main () {
  was = signal (SIGALRM, catch); /* catch SIGALRM / save previous action */
  timed_out = FALSE;

  alarm (TIMEOUT); /* set the alarm clock */
  printf ("in time-critical section\n"); /* do something time critical */
  sleep (3); /* set to value < 5 to be OK, or > 5 to provoke overrun */
  alarm(0); /* turn alarm off */

  /* if timed_out is TRUE, then the action took too long */
  if (timed_out == TRUE) printf ("Time overrun\n");
  else {
     signal (SIGALRM, was); /* restore old signal action */
       printf ("Just in time\n");
    }
}
catch (sig) int sig; {
  timed_out = TRUE;         /* set timeout flag */
  signal (SIGALRM, catch); /* reinitialise signal handler action */
}
```

6.6 Jumping

Listing 11: Alarm Signal

```
main () {
  setjmp (position);
  signal (SIGINT, goback); /* set signal action correctly */
  printf ("Signal set back to goback\n");
  display_main_menu();
}
goback (signo) int signo; {
  signal (SIGINT, SIG_IGN);
  printf ("\nInterrupted by signal %d\n", signo);
    fflush (stdout);
  longjmp (position, 1); /* jump back to saved position */
}
display_main_menu () {
  fflush (stdout);
  printf ("Waiting in main menu: ");
  fflush (stdout);
    pause();
}
```