FHNW Informatik

SYSTEMPROGRAMMIERUNG

Dozent: Hannes Lubich

Autor: Christian Glatthard (chregi.glatthard@gmail.com)

3. Semester, HS 2012

Version: 1.2, 05.01.13 11:11:19

INHALTSVERZEICHNIS

1 DIE PROGRAMMIERSPRACHE C

C ist eine der ältesten Programmiersprachen. Entwickelt wurde sie in den frühen 1970er Jahren von Dennis Ritchie an den Bell Laboratories für die Systemprogrammierung des Betriebssystems Unix. Seitdem ist sie auf vielen Computersystemen verbreitet.

C ist eine imperative Programmiersprache, das heisst ein Entwickler schreibt im Code Schritt für Schritt was der Computer machen soll. C kennt noch keine Objekte.

Das Hauptanwendungsgebiet von C ist in der Systemprogrammierung einschliesslich der Erstellung von Betriebssystemen und in der Programmierung von eingebetteten Systemen (embedded systems).

C ermöglicht direkten Hardware und Speicherzugriff.

Mithilfe von Header Dateien können andere Module eingebunden und verwendet werden.

1.1 DATENTYPEN

C unterscheidet zwischen 5 verschiedenen Datentypen:

- Objekttypen (object types)
- Funktionstypen (function types)
- unvollständige Typen (incomplete types)
- Basisdatentypen (basic types)
- abgeleitete Typen (derived types)

1.1.1 BASISDATENTYPEN

Datentyp	Typ	Beschreibung
Void	Incomplete	
_Bool		
char, wchar_t		
Int		Ganzzahlige Werte
float, double, long		Gleitkommazahlen

signed char ≤ short int ≤ int ≤ long int ≤ long long int.

1.1.2 ABGELEITETE TYPEN

Abgeleitete Typen werden wie folgt aufgeteilt:

- Felder (array types)
- Zeiger (pointer types)
- Funktionen (function types)
- zusammengesetzte Typen (structure types und union types)

1.2 POINTER

Etwas spezielles in C sind die Pointer. Da diese nicht ganz ohne sind, widme ich ihnen einen eigenen kleinen Abschnitt.

1.2.1 EINFÜHRUNG - VARIABLEN

Um Pointer zu verstehen müssen wir uns zuerst überlegen was Variablen sind und was passiert wenn wir eine Variable definieren.

Eine Variable ist etwas mit einem Namen dessen Wert sich ändern kann. Bei der Deklaration einer Variable wird ein spezifischer Speicherblock für dessen Wert reserviert. Für Integers zum Beispiel werden auf modernen 32bit PCs 4 byte reserviert. Die Grösse eines Datentyps kann jedoch von Maschine zu Maschine anders sein.

Um die Grösse eines beliebigen Typs herauszufinden kann die Prozedur sizeof(type) abhelfen.

Wenn wir also eine Variable deklarieren, z.B.

```
int k;
```

wird der Compiler beim Durchlaufen dieser Zeile sofort einen 4 byte Bereich im Memory für k reservieren. Ausserdem fügt er k einer Symboltabelle hinzu, zusammen mit der relativen Memory-Adresse. Später wenn wir k einen Wert zuweisen

```
k = 2;
```

schreibt der Compiler an die für k reservierte Adresszeile den Wert 2. Eine Zuweisung in der Art von

```
a = k;
```

wird vom Compiler so verarbeitet, dass er sich den Wert von k holt und ihn an die für a reservierte Speicherstelle schreibt.

1.2.2 RVALUE UND LVALUE

Man spricht davon, dass Variablen einen **rvalue** und einen **lvalue** besitzen, je nachdem ob sie links oder rechts vom Gleichheitszeichen stehen.

Der **lvalue** (sprich wenn die Variable links vom Gleichheitszeichen steht, ihr also ein Wert zugeordnet wird) wird vom Compiler als die Adresse der Variable interpretiert. Der **rvalue** dagegen greift auf den Wert zu der an der Adresse der Variable gespeichert ist.

1.2.3 UMGANG MIT POINTERN

Nun sind Pointer eigentlich nichts anderes als Variablen welche als Wert einen **lvalue** besitzen, d.h. eine Speicheradresse. Um eine Variable als Pointer zu deklarieren wird dem Variablenamen ein * vorangestellt. Ausserdem geben wir dem Pointer einen Typ, dieser definiert was für ein Datentyp an der gespeicherten Adresse abgelegt werden wird. Das heisst im folgenden Beispiel, dass der Pointer ptr auf einen Integer-Wert zeigt. Pointer werden standardmässig als NULL-Pointer initialisiert, dies kann aber vom System abhängen.

```
int *ptr;
ptr = NULL;  /* Make it a NULL-pointer */
```

Wir wollen nun im Pointer ptr die Adresse einer Variablen k speichern. Um die Adresse von k (den **rvalue**) zu erhalten, stellen wir der Variablen ein & vor.

```
ptr = &k;
```

Ein Pointer enthält also als Wert eine Speicheradresse und „zeigt“ somit auf eine Stelle im Speicher.

Um an die Stelle im Speicher zu schreiben ist wieder der asterisk zu gebrauchen.

```
*ptr = 7;    /* writes the value 7 to the adress of the pointer */
```

Da ptr an die Adresse von k zeigt ist nun auch der Wert von k = 7.

1.3 SYNTAX

Konstante definieren	`const string[] x = „test\n“`
Variable definieren	`i: integer;`
Array definieren	`i[10]: integer;`
Struct / (Record) definieren	`record x { i: integer; x: character; }` `struct y { i: integer; x: character; }`
Variable vom Typ record / struct definieren	`record q_entry s_entry;` `struct r_entry t_entry;`
Struct verwenden	`s_entry.i = 2;`
Pointer definieren	`int *x;`
Pointer auf c zeigen lassen	`x = &c;`
8 an die entspr. Adresse schreib.	`*x = 8;`
Speicherstelle neben x	`*x+1;`
Adresse einer Variable	`&c;`
Prozeduraufruf	`procedure (1,2, „test“);` `procedure (&s_entry);`

If Else	`if (Bedingung) then {}`
	`else {}`
	`[ else if () {} ]`
	`endif;`

Switch case	`switch (Variable);`
	`case (Bedingung): {break;};`
	`default: break;`

While Schlaufe	`while (Bedingung) do {}`
For Schlaufe	`for (Initialwert; Maxwert; Veränderung) {}`
Do While Schlaufe	`do {} while (BedingungI`
Austritt aus Schlaufe	`Continue;`
Sprünge	`Goto`
Eingabe	`read(param)`
Ausgabe	`print(param)`
Header laden	`#include file.h`

Makro definieren	`#define „Text“ „Ersatztext“`
Ausnahmen	`#undef „Text“`
Bedingte Definition	`#if !defined (BLA) #define BLA „bla“ #endif`

1.3.1 PRINTF()

Datentyp	Platzhalter
Character	%c
Decimal	%d
Floats	%f
String	%s

Für weitere Beispiele lohnt sich ein Blick auf die Code Reference Card.

1.4 DATEIEN IN C

- Quellcode: *.c
- Deklarationen: *.h
- Übersetzte Libs: *.a
- Objektcode: *.o
- Ausführbare, gelinkte Datei: a.out

1.5 PROGRAMMIEREN MIT C

Um mit C zu programmieren benötigt man eigentlich nicht viel ausser einem Editor (ggf. mit Syntax Unterstützung), sowie einem Compiler (Open Source: GNU C Comiler [gcc]).

1.5.1 CODE TEMPLATE

```
#include <stdio.h>      /* Include Standard IO Library */

int main () {
   printf("hello world!\n");
}
```

1.5.2 PROGRAMM KOMPILIEREN

```
gcc –p <Name der kompilierten Datei> source.c
```

1.5.3 PROGRAMM STARTEN

```
./a.out          /* wenn nichts mit –o angegeben wurde */
```

1.6 WIESO C?

- Enge Beziehung zwischen Linux / Unix und C
- Systemkernschnittstelle von Unix / Linux ist in C definiert
- Kernel und Utilities sind grösstenteils in C
- maschinennah
- DIE Programmiersprache für Systemprogrammierung
- universelle Programmiersprache
- Zugriff auf Bits & Bytes
- erlaubt strukturiertes und modulares Programmieren
- allgemein anerkannt und hat sich durchgesetzt

1.7 FEHLERSUCHE & FEHLERBEHEBUNG

- strace (Linux) / truss (Solaris)

Dump aller System Calls, Parameter & Rückgabewerte

- bedingte fprintf Aufrufe → Standard Error Output:

```
/* # define DEBUG */
…
fprintf(stderr, „Standard Error, i=%d\n“, i);
fflush(stderr);
#endif
```

2 SINGLE UNIX SPECIFICATION (SUS)

Die Single UNIX Spezifikation wurde mit dem Hintergrundgedanken ins Leben gerufen ein einheitliches API für sämtliche UNIX Systeme zu definieren. Sie wurde in ihrer ursprünglichen Form von der „Open Group“ definiert. Wenn ein Betriebssystem die Spezifikationen erfüllt und allgemein verfügbare Applikationen darauf laufen, dann ist es ein offenes System.

Das grundsätzliche Ziel ist es, Portabilität für Programmquellcode zu garantieren.

2.1 WAS IST EIN API

Ein API (Application Program Interface) ist ein geschriebener Vertrag zwischen System- und Applikationsentwicklern. Ein API ist kein Stück Code, sondern ein Stück Papier in dem definiert wird was die beiden Entwicklergruppen zur Verfügung gestellt bekommen, respektive zur Verfügung stellen müssen.

2.2 UNIX – DIE MARKE

Heutzutage haben alle Vertreiber die Single UNIX Spezifikation implementiert. Die Marke UNIX wird mithilfe von ausführlichen Tests validiert und mit einer einzigartigen Vertreiber Garantie unterstützt: **The Open Brand**.

2.2.1 THE OPEN BRAND

– läuft unter Markenrecht

– kennzeichnet Produkte die garantiert die Open Systems Specification erfüllen

– der Vertreiber garantiert, dass jegliche nicht Erfüllung der Spezifikation innerhalb einer definierten Zeitspanne korrigiert wird

2.3 SINGLE UNIX SPECIFICATION 3

Die Version 3 der Spezifikation wurde 2001 veröffentlicht und wird von der Austin Group weitergepflegt, welche sich auch um die POSIX (Portable Operating System Interface) kümmern.

SUSv3 besteht aus ungefähr 3700 Seiten, welche thematisch in vier Hauptteile gegliedert sind:

Base Definitions (XBD)

Eine Liste von Definitionen und Konventionen welche in der Spezifikation verwendet werden, zusätzlich eine Liste von C Header Files welche von den Systemen zur Verfügung gestellt werden müssen. Alles in allem werden 84 Header Files aufgelistet.

Shell and Utilities (XCU)

Eine Liste von Utilities und eine Beschreibung der sh Shell. 160 Utilities insgesamt.

System Interfaces (XSH)

Enthält Spezifikationen von verschiedenen Funktionen die als System Calls oder Library Funktionen implementiert sind. 1123 System Interfaces sind spezifiziert.

Rationale (XRAT)

In diesem Teil wird der Hintergrund des Standards erklärt.

Die Standard Commandline und das Scripting Interface ist die POSIX Shell, eine Erweiterung der Bourne Shell, basierend auf einer frühen Version der Korn Shell.

Wichtig zu wissen ist, dass ein System nicht auf dem Source Code des ursprünglichen AT&T Unix basieren muss um die Spezifikation zu erfüllen. Z/OS zum Beispiel ist ein qualifiziertes Unix System obwohl es kein Zeile des AT&T Unix Quellcodes enthält.

Eine HTML Version der Spezifikation findet sich online: http://www.UNIX-systems.org/version3 .

2.3.1 PORTABLE OPERATING SYSTEM INTERFACE (POSIX)

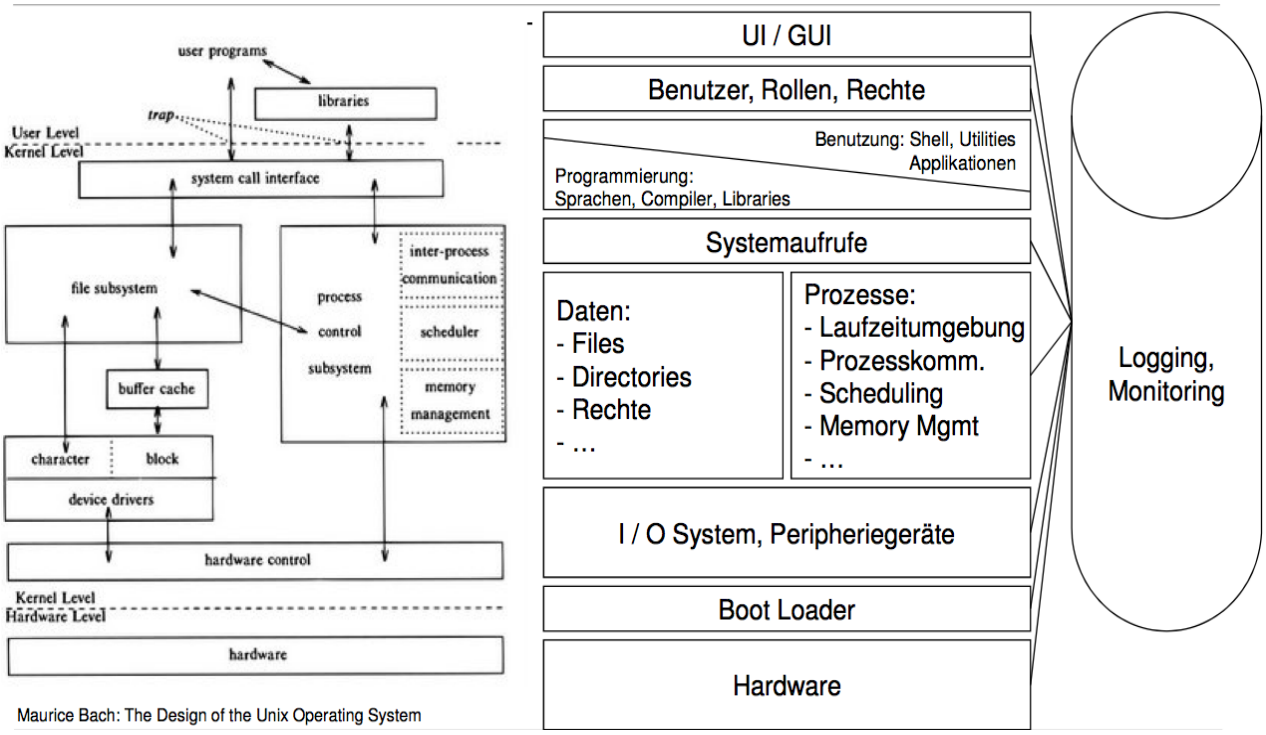
POSIX ist eine Familie von Standards die durch das IEEE spezifiziert wurde um Kompatibilität zwischen Betriebssystemen zu gewährleisten. Es definiert die API zusammen mit Command Line Shells und Utility Interfaces für Software Kompatibilität mit verschiedenen Unix Varianten.

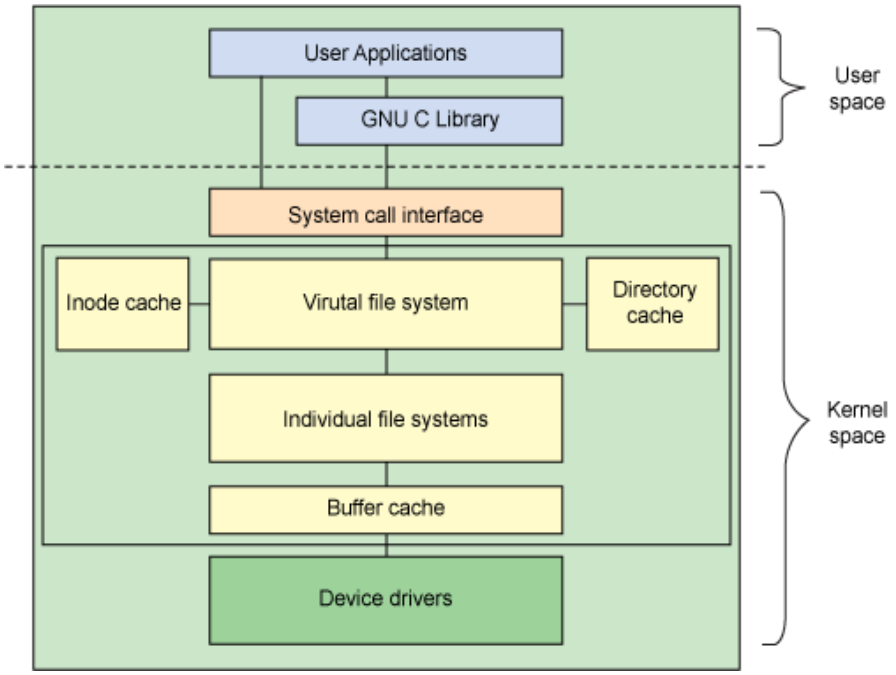
In POSIX enthalten ist ebenfalls der ANSI C Standard.

Für mehr Infos zu den einzelnen POSIX Standards hilft ein Blick auf Wikipedia: http://en.wikipedia.org/wiki/POSIX#Name

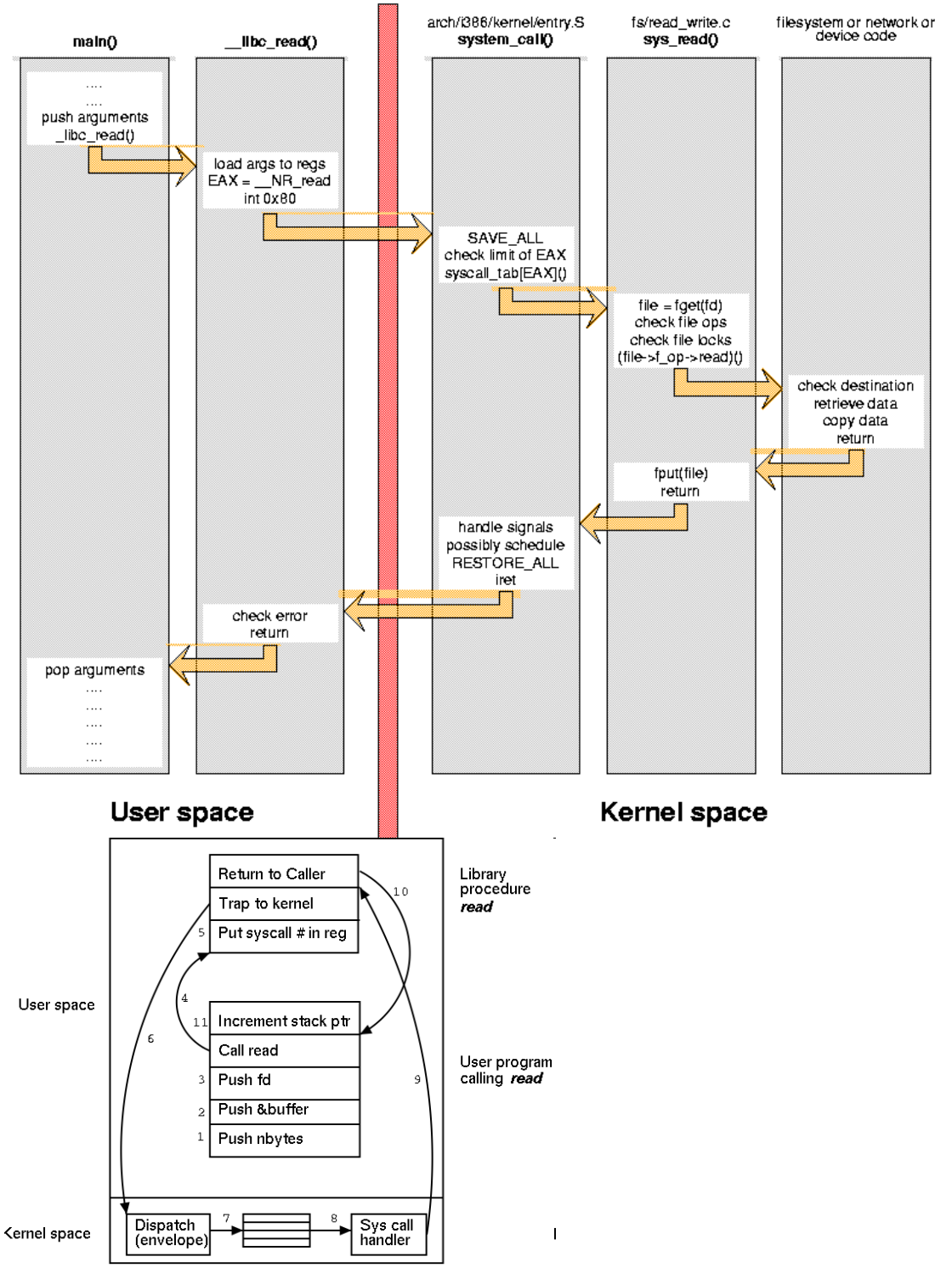
3 EINFÜHRUNG

3.1 UNIX AUFBAU





3.2 SYSTEMCALL SCHNITTSTELLE



3.2.1 SYSTEM- UND BENUTZERLIBRARIES

Befinden sich in /usr/lib/libc.a

Systemcall	Beschreibung
Printf(), fprintf()	
Perror()	Ausgabe der letzten System Fehlermeldung
Scanf()	
Fopen(), fclose()	File öffnen, schliessen
Getc(), putc()	
Opendir()	
Readdir()	
Writedir()	
Closedir()	

4 PROZESSE

Prozesse sind ein oder mehrere Programme die auf einem oder mehreren physischen oder virtuellen Prozessoren ausgeführt werden. Sie sind zu jeder Zeit mit „computational state“ verbunden (aktuell verwendete Variablen etc. im Programm). Prozesse werden vom Betriebssystem strikt überwacht und verwaltet (korrektes Verhalten im System, Ressourcen-Verbrauch, Synchronisation etc.)

4.1 ANFORDERUNGEN AN EIN PROZESS-STEUERSYSTEM

- Prozesse kreieren
- Prozesse starten
- Prozesse schedulen, Warteschlangen, Ressourcenverbrauch
- Prozesse stoppen, unterbrechen
- Prozesse terminieren (freiwillig / wegen Fehler)
- Prozess-Signalisierung und -Kommunikation (Prozess-Prozess & Kern-Prozess / Prozess-Kern)
- Faire Zuordnung von Hauptspeicher und anderen geteilten Ressourcen
- Ein- / Auslagerung von Prozessen bei vollem Speicher

- Prozesse und ihre Zustände anzeigen

4.2 KERNEL & USER MODE

Ein Prozess in UNIX hat genau zwei Ausführungsmodi:

- User Mode: Ausführung von normalem Programmcode
- Kernel Mode: Systemaufrufe (Systemcalls) oder Ausnahmenbehandlung (Exception Handling)

Der Modus kann durch einen Systemcall durch das Programm, eine Ausnahmesituation (Exception), oder durch asynchrone Events gewechselt werden.

Beide Modi haben separate Segmente und sind voneinander abgeschirmt.

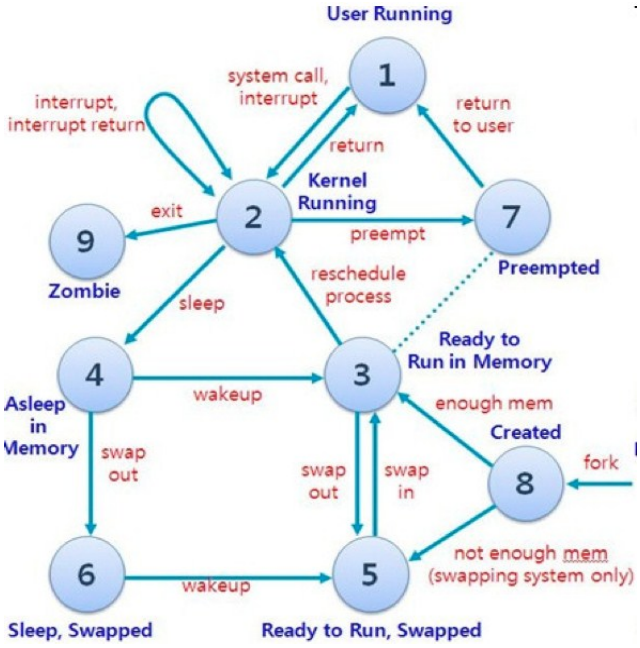
4.3 PROZESS-KONTEXT

Benutzer: Daten des Prozesses im zugewiesenen Adressraum

Hardware: Basis- und Grenzregister, Befehlszählregister, Akkumulator & Seitentabelle

System: Prozessnummer, geöffnete Dateien, Eltern- / Kindprozesse, Prioritäten

4.4 ZUSTÄNDE



4.5 SCHEDULING

– Vermeidung von Starvation und Deadlocks

– nötig durch Mehrbenutzersystem, Multitasking

4.5.1 PRIORISIERUNG

Unix verwendet 3 Prioritätsklassen (virtual scheduling):

– **Realtime:** Scheduling mit fixen Prioritätsklassen (wichtiger als Fairness)

– **System:** Geschlossene Scheduling-Klasse für Systemprozesse (inkl. Null Prozess oder init Prozess)

– **Time-shared:** faires Scheduling zwischen allen Benutzerprozesse

UNIX implementiert die FIFO (First in First out) und round-robin (Reihum Prinzip) Realtime Scheduling Klassen, in beiden Fällen hat jeder Prozess eine zusätzliche Priorität zu seiner Scheduling-Klasse.

– Der Scheduler führt den Prozess mit der höchsten Priorität aus (gibt es zwei mit derselben Priorität, wird der ausgeführt, welcher schon länger wartet)

– FIFO Prozesse werden ausgeführt bis sie entweder beenden oder blockieren

– Round-Robin Prozesse werden nach einer Weile preempted (vorweggenommen) und der Scheduling Queue hinten wieder angehängt. Auf diese Weise teilen sich round-robin mit gleicher Priorität automatsch fair ihre Zeit auf

Die Scheduling Klasse eines Prozesses entscheidet welcher Algorithmus angewendet wird.

Je länger ein Prozess läuft, umso niedriger ist seine Priorität. Ebenfalls sinkt die Priorität mit steigendem Ressourcenverbrauch.

4.6 SYSTEMCALLS

Systemcall	Beschreibung
Fork()	Prozess erzeugen durch Kopieren
Exit()	Prozess beenden mit Return-Wert
Exec()	Ausführen eines neuen Programms in vorhandener Prozesshülle
Wait()	Warten auf Prozesstermination (Kindprozesse)
Sleep()	Freiwilliges Schlafen

Kill()	Senden eines Signals (32 Signaltyoen)
Signal()	Signalbehandlung (behandeln / ignorieren)
Pipe()	
Socket()	
Getpid()	Prozess ID des aktuellen Prozesses

4.6.1 EXEC()

Von exec gibt es viele verschiedene Varianten: execl, execv, execle, execlp, execvp, exect. Die Unterschiede liegen dabei bei den möglichen Parametern. Bei allen muss jedoch der Pfad zum neuen Prozess angegeben werden. Wird in einem Programm der exec() Systemcall aufgerufen „stirbt“ somit der aktuelle Prozess und der neue wird in der alten Prozesshülle ausgeführt. Das heisst jeglicher Code nach einem exec Systemcall wird nicht mehr ausgeführt.

```
#include <stdio.h>
main() {
      (void) printf ('process started\n');
      execl('/bin/date', '/bin/date', (char *) 0);
      (void) printf ('This should not happen\n');
}
```

4.6.2 FORK()

Fork erstellt einen neuen Prozess als Kopie des aktuellen und gibt danach einen Return Code zurück. Der neue Prozess (Kind) erhält eine „0“. Der aufrufende Prozess (Eltern) erhält entweder eine „-1“ im Fehlerfall, oder einen Wert grösser als 0, welcher der Prozess-ID des Kindprozesses entspricht.

Der Kindprozess ist eine identische Kopie des Elternprozesses. Ausnahmen sind die Prozess-ID und der Ressourcenverbrauchszähler.

```
#include <stdio.h>
main () {
  switch (fork()) {
    case -1: (void) printf('fork failed\n');
      break;
    case 0: (void) printf ('child executing\n');
    default: (void) printf ('parent executing\n');
      break;
  }
```

```
}
```

4.6.3 EXIT()

Exit() terminiert einen Prozess und gibt dessen Ressourcen (Speicher, offene Dateien etc) frei. Nur der Eintrag in der Prozesstabelle bleibt als Zombie-Prozess bestehen, bis der Elternprozess den Rückgabewert abruft. Sämtliche Unterprozesse des terminierten Prozesses gehen an den init Prozess (pid 1) über.

4.6.3.1 ATEXIT()

In modernen UNIX Systemen gibt es die Möglichkeit benutzerdefinierte Exit-Funktionen (zum Aufräumen) mit der atexit() Funktion zu registrieren:

```
int atexit (void (*function) (void));
```

Beim Aufruf von exit werden alle zuvor mit atexit() registrierten Funktionen in umgekehrter Reihenfolge ausgeführt.

4.6.4 WAIT()

Der wait Systemcall suspendiert einen Prozess von der CPU bis ein Kindprozess terminiert oder eine Ausnahme signalisiert. Wait() liefert dem Elternprozess dann den Exit Code des Kindprozesses zurück, oder einen Indikator, warum der Kindprozess vom Betriebssystem beendet wurde.

Hat ein Prozess beim Aufruf von wait() keine Kindprozesse, wird er nicht schlafen gelegt. Je nach System liefert wait() einfach den Exit Code des ersten terminierten Kindprozesses oder erst den des letzten.

4.6.4.1 SIMPLES PROGRAMM MIT WAIT()

```
#include <stdio.h>
#include <sys/wait.h>
main () {
      union wait status;
      if (fork() == 0)
      (void) execl ("/bin/date", "date", (char *) 0);
```

```
      /* parent process */
      (void) printf ("parent awaiting term. of child\n");
      (void) wait (&status);
      (void) printf ("parent after noticing term. of child\n");
}
```

4.6.4.2 AUSWERTUNG DES EXIT CODES NACH WAIT()

```
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
  pid_t cpid, w;
  int status;
  cpid = fork();
  if (cpid == -1) { perror("fork"); exit(EXIT_FAILURE); }
  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 */
    w = waitpid(cpid, &status, WUNTRACED | WCONTINUED);
    if (w == -1) { perror("waitpid"); exit(EXIT_FAILURE); }
    if (WIFEXITED(status)) {
      printf("exited, status=%d\n", WEXITSTATUS(status));
    } 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)) {
      printf("continued\n"); }
    } while (!WIFEXITED(status) && !WIFSIGNALED(status));
    exit(EXIT_SUCCESS); } }
```

5 THREADS

5.1 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 unterstützen mehr als einen parallelen Ausführungspfad innerhalb eines Prozesses (multi-threading) → es muss kein Kontextwechsel vorgenommen werden um eine andere Aktivität zu starten.

5.2 MUTEX

Da es bei multi-thread Anwendungen zu Synchronisationsproblemen mit geteilten Ressourcen kommen kann, wurden Mutexe eingeführt um Variablen vorübergehend zu blockieren.

Wird nun ein Thread gestartet welcher mit einer geteilten Variable arbeiten muss, kann er diese locken (sperren). Dadurch müssen sämtliche anderen Threads warten bis der blockierende Thread den Lock entfernt und sie die Variable verwenden können.

Typen: fast (Default), recursive, non-recursive

5.3 SCHEDULING

- **SCHED_FIFO:** First in First Out, Ablaufbereiter Thread mit höcchster Priorität läuft bis zur Terminierung, dann nächster usw.
- **SCHED_RR:** Round Robin, ablaufbereite Threads mit gleicher höchster Priorität laufen mit Time Slice
- **SCHED_FG_NP:** alle ablaufbereiten Threads laufen mit Time Slice, Länge des Time Slice entspricht Priotität
- **SCHED_BG_NP:** wie SCHED_FG_NP, jedoch können Hintergrund-Threads durch SCHED_FIFO oder SCHED_RR verdrängt werden und erhalten weniger Ausführungszeit als SCHED_FG_NP

5.3.1 PROGRAMMBEISPIEL MUTEX

```
/* http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
void *functionC();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int counter = 0;
main() {
  int rc1, rc2;
  pthread_t thread1, thread2;
  /* Create independent threads each of which will execute functionC */
  if( (rc1=pthread_create( &thread1, NULL, &functionC, NULL)) ) {
    printf("Thread creation failed: %d\n", rc1); }
  if( (rc2=pthread_create( &thread2, NULL, &functionC, NULL)) ) {
    printf("Thread creation failed: %d\n", rc2); }
  /* Wait till threads are complete before main continues. Unless  we */
  /* wait we run the risk of executing an exit which will terminate
*/
  /* the process and all threads before the threads have completed.
*/
  pthread_join( thread1, NULL);
  pthread_join( thread2, NULL);
  exit(0); }
  void *functionC() {
  pthread_mutex_lock( &mutex1 );
  counter++;
  printf("Counter value: %d\n",counter);
  pthread_mutex_unlock( &mutex1 ); }
```

5.4 SYSTEMCALLS

Pthread_create()	Erzeugung eines Threads
pthread_exit()	Selbst-Termination eines Threads
pthread_cancel()	Fremd-Termination und Exit-Code 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-ID's
pthread_once()	Ausführung einer Initialisierungs-Routine
pthread_mutexattr_init()	
pthread_mutexattr_destroy()	
pthread_mutex_lock()	
pthread_mutexattr_trylock()	
pthread_mutex_unlock()	
pthread_mutex_init()	

pthread_mutex_destroy()	
pthread_attr_setsched()	Scheduling Methode setzen
pthread_attr_setprio()	Priorität für Scheduling setzen

6 DATEISYSTEM

6.1 ANFORDERUNGEN

- Konsistente, permanente und strukturierte Ablage von Dateien (Hierarchie, Extensions..)
- Zugriffssicherheit (Rechte / Rollen, Attribute [read-only, hidden, …])
- Basisoperationen (Benutzer-Navigation, Datei-Manipulationen)
- Programmierbarer Zugriff (API?)
- Mehrbenutzer/-prozessfähigkeit
- Performance & Standardisierung

6.2 AUFBAU

Disk

Boot Block	Partition	Partition

Partition

Block Group	Block Group	Block Group	Block Group	Block Group

Block Group

Superblock	Group Descr.	Block Bitmap	Inode Bitmap	Inode Table	Data Blocks

Superblock

- Anzahl inodes & Datenblöcke
- Adresse des 1. Datenblocks
- Anzahl freie Blöcke & Inodes
- Grösse eines Datenblocks
- Blöcke / inodes pro Gruppe
- Anzahl Bytes pro Inode

Block Group Descriptor

- Adressen des Block & Inode Bitmaps
- Adresse der Inode Table
- Anzahl freie Blocks & Anzahl freie Inodes

Block Bitmap

Gibt für jeden Data Block der Gruppe an ob er noch frei ist oder nicht. Jedes Bit der Block Bitmap steht für einen Data Block. 1 bedeutet „used“ und 0 bedeutet „free/available“

Inode Bitmap

Analog zum Block Bitmap, einfach für die Inodes in der Inode Table.

Inode Table

Jede Inode repräsentiert 1 File des Filesystems.

Data Blocks

Enthalten die wirklichen Daten.

6.2.1 INODE (INDEX NODE)

Die Inode ist ein fundamentales Konzept in der Welt der UNIX Filesysteme. Jedes Objekt des Filesystems wird durch eine Inode repräsentiert. Jede Inode ist identifizerbar durch eine (innerhalb des Filesystems) eindeutige inode Nummer. Jedes File unter UNIX hat folgende Attribute:

- **type** – Filetyp (-, l, d, b, c, p)

- **owner** - Besitzer
- **group** – Besitzergruppe
- **perms** – Berechtigungen (z.B. rwxr-xr-x)
- **file accessed** – Datum des letzten Zugriffs auf die Datei
- **content modified** – Datum der letzten Änderung am Inhalt der Datei
- **inode modified** – Datum der letzten Änderung an der Inode
- **number of links** – Anzahl Hard-Links auf die Datei im Filesystem (min. 1, ansonsten ist die Datei nicht im FS ersichtlich)
- **size** – Grösse der Datei in Byte
- **datablock information** – Adresse der Datei im Speicher

6.2.2 INODE ANZEIGEN

Um die Inode Nummer einer Datei anzuzeigen kan ls -i verwendet werden.

Ausserdem kann man mit dem Befehl „stat“ sämtliche Inode Attribute einer Datei anzeigen lassen.

6.3 INODE ALLOZIIEREN

- Dateisystem feststellen
- Superblock locken (Schlafen falls besetzt)
- nächsten Inode aus „Free List* holen
- leer? → neu auffüllen

6.4 DATENBLOCK ALLOZIIEREN

- Dateisystem feststellen
- Superblock locken (Schlafen falls besetzt)
- nächsten freien Datenblock aus der „Free list“ der Block Bitmap holen
- leer? → Schlafen bis Datenblock frei

6.5 PERIPHERIEGERÄTE

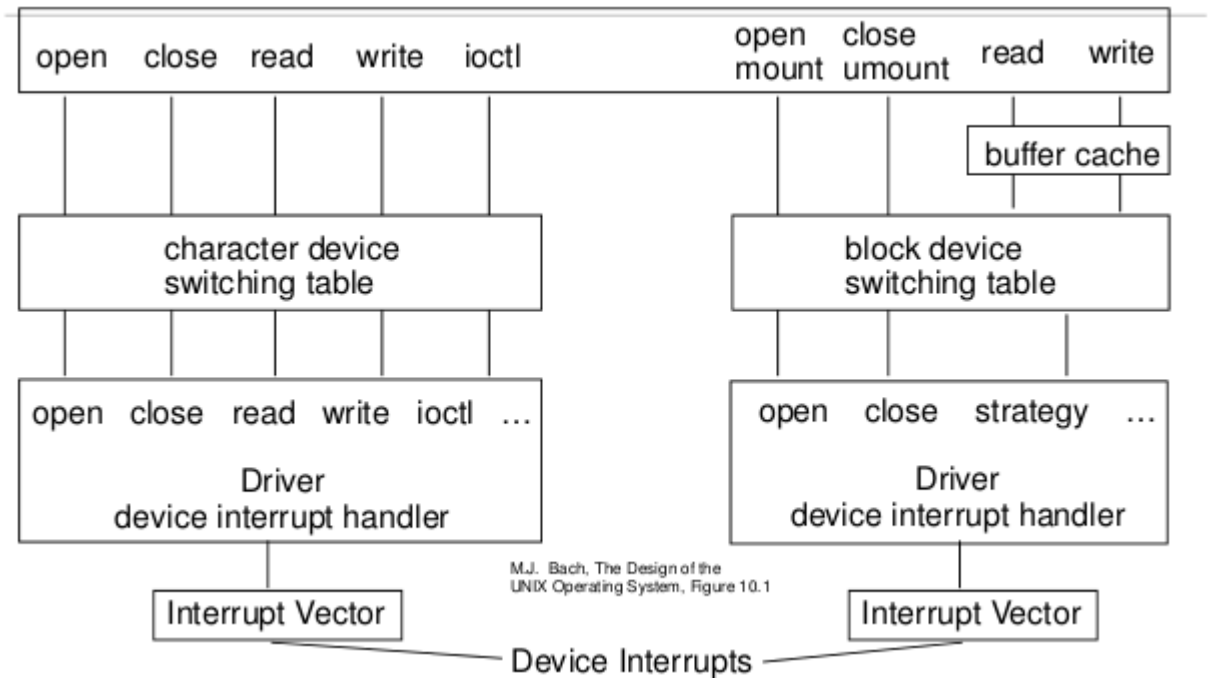
Sämtliche Peripheriegeräte werde als Gerätespezialdateien im /dev Dateisystem mit einer einheitlichen Schnittstelle eingebunden.

- Major / Minor Device Nummer
- Zugriffsrechte auf Geräte sind relevant
- block- oder zeichenweiser Zugriff
- Echte Geräte und Pseudo-Geräte (z.b. virtuelle Terminals tty, Netzwerkprotokolle oder (/dev/null)

6.5.1 GERÄTETREIBER

Gerätetreiber sind die einzige Schnittstelle über die ein Prozess mit Geräten kommunizieren kann. Sie sind Teil des kernel-Codes. Sie werden entweder statisch beim Systemstart oder zur Laufzeit als Module geladen (insmod / rmmod).

6.5.2 ZUGRIFF ÜBER GERÄTESPEZIALDATEIEN



6.6 LIBRARY CALLS

Library Call	Beschreibung
Opendir	
Readdir	
Closedir	
Rewinddir	

6.7 SYSTEMCALLS

Systemcall	Beschreibung
Creat()	
Mknod()	
Open()	
Close()	
Unlink()	
Read()	
Write()	
Seek(), lseek()	
Ioctl()	
Dup()	
Chown, chmod, umask	
Chdir()	
Mount(), umount()	
Namei()	Pfadnamen in inode Nummer
Fstat()	
Getcwd()	Get current working directory

6.8 BTRFS – DAS DATEISYSTEM DER ZUKUNFT?

Btrfs (auch B-Tree oder Better-Filesystem genannt) gibt es seit 2008. Obwohl es noch nicht stabil ist, ist es bereits Bestandteil des Linux Kernels. Es wird bereits seit 2010 von MeeGo, sowie in der Enterprise Version 6 des Red Hat Linux verwendet. Das Filesystem verwaltet sämtliche Files sowie den Festplattenplatz in B-Baumstrukturen. Verwaltete Dateien und Volumes können einzeln bis zu 16 Ebyte gross sein (1'000'000 TB). Es unterstützt Snapshots und Prüfsummen.

7 VIRTUAL MEMORY (VM)

Wenn im Bezug auf UNIX über „Memory“ oder „Speicher“ gesprochen wird, meint man damit normalerweise die Virtual Memory und nicht einfach den RAM Speicher. Diese besteht erstreckt sich sowohl über den physikalischen Speicher, sowie über den Swap Space.

– Erweiterung des Hauptspeichers pro System (mehr Prozesse als Speicher) / pro Prozess (einzelner Prozess grösser als verfügbarer Speicher)

– Organisation des Hauptseichers in gleich grosse, einheitlich adressierbare Einheiten (pages)

– benötigt Hardware-unterstützte Abbildung zwischen physischen und virtuellen Adressen

– statische Tabelle

– Assoziationstabelle / Cache

– dynamische Auflösung wenn eine Adresse referenziert wird (Basis Register)

7.1.1 VIRTUAL MEMORY HANDLER

Der Virtual Memory Handler versucht stetig einen gewissen Bereich freien RAM zu halten. Der exakte Algorithmus dazu unterscheidet sich von System zu System, jedoch sind grundsätzlich 3 Variablen definiert um seine Aktivität zu steuern:

Variable	Beschreibung
Lotsfree	Ist der Pool an freiem Speicher grösser als dieser Wert, ist der VM Handler idle. Sobald der Pool kleiner wird, beginnt er damit nicht mehr verwendete Pages auf die Free List zu setzen.
Desfree	Unter diesem Limit beginnt der Handler damit Pages die zwar unter Verwendung stehen, jedoch schon eine längere Zeit nicht verwendet wurden aufzuräumen. Je mehr der Wert überschritten wird, umso mehr Aufwand wird dafür betrieben.
Minfree	Wenn der Pool kleiner wird als dieser Wert, beginnt der Handler mit Trashing und wenn alles nichts mehr nützt mit desperation swapping.

7.1.2 DEMAND PAGING

Demand Paging ist eine Technik die es erlaubt Prozesse zu laden die grösser sind als der verfügbare Hauptspeicher. Voraussetzungen sind Hardware welche seitenorientiertes Speichermanagement unterstützt, sowie wiederaufsetzbare CPU Instruktionen.

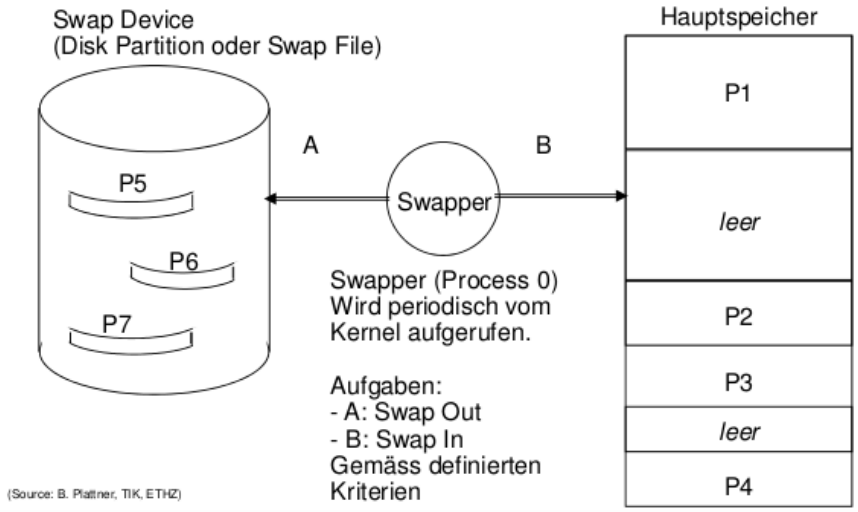
In der Regel müssen nur etwa 10-15% des Programmcodes zu jeder beliebigen Zeit im Hauptspeicher liegen, das ist das „working set“.

Zugriff auf eine Seite ausserhalb des „working set“ führt zu einem „page fault“ woraufhin die entsprechende Seite nachgeladen wird.

Demand Paging wird optimiert durch:

- „Reference Bit“ → ermöglicht nicht-lineares „working set“
- „Age bits“ → verbleiende Zeit einer Seite im „working set“

7.2 SWAPPING



7.3 SYSTEMCALLS

Systemcall	Beschreibung
Malloc()	Memory allozieren
Free()	Memory freigeben

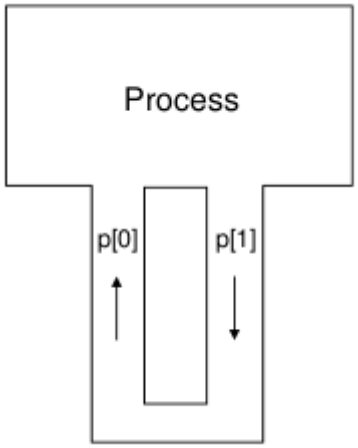
8 PIPES

- Unidirektionaler Kommunikationskanal zwischen verwandten Prozessen (Prozesse mit direkter / indirekter (vererbter) Eltern-Kind Beziehung nach einem fork())
- direkt nutzbar für Input / Output Umleitung in der Shell
- repräsentiert durch 2 Dateideskriptoren innerhalb des Prozesses umd 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

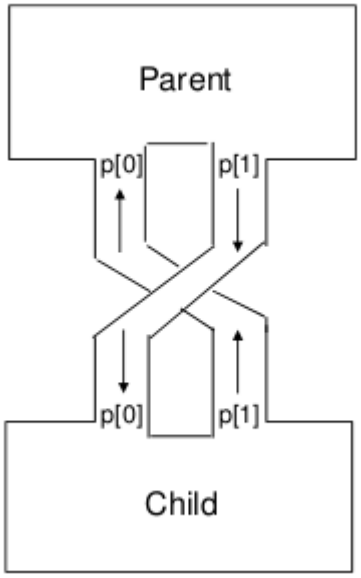
8.1 PROGRAMMBEISPIEL 1

```
#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);
}
```



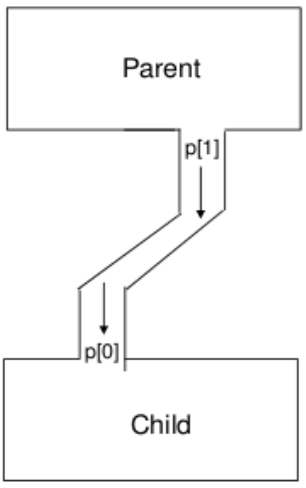
8.2 PROGRAMMBEISPIEL 2

```
#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); }
```



8.3 PROGRAMMBEISPIEL 3

```
#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) {
```

```
    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); }
```

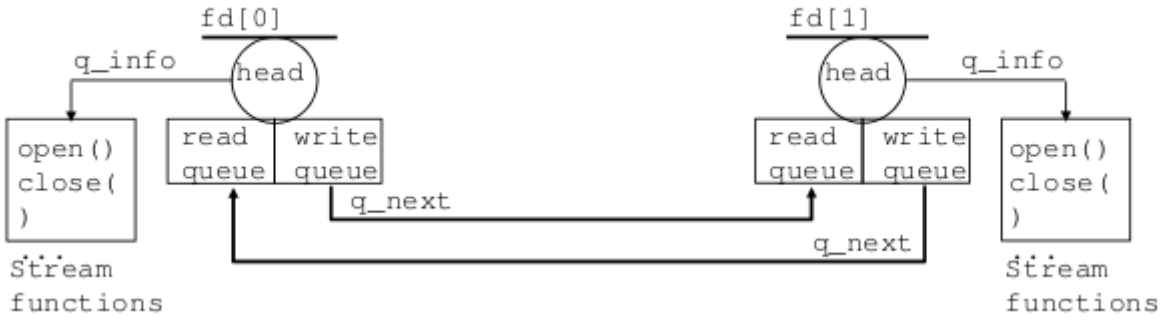
8.4 SCHLIESSEN UNBENUTZTER PIPE-ENDEN

Das Schliessen unbenutzter Pipe-Enden wie im Programmbeispiel 3 hat verschiedene Gründe, und sollte deshalb stets gemacht werden:

1. Saubere Programmierung
2. Schutz gegen Copy-Paste Fehler
3. Auf alten UNIX Systemen nur begrenzte Anzahl verfügbarer Dateideskriptoren pro Prozess
4. Schliessen des letzten Lese- oder Schreib-Endes einer Pipe im Betrieb hat Auswirkungen auf die Prozesssynchronisation

8.5 DATENSTRUKTUR

In den aktuellen UNIX Systemen werden für Pipes und Named Pipes dieselben Mechanismen gebraucht. Daten werden in einer Pipe in STREAMS Warteschlangen, bzw. Unix-Type Sockets gespeichert.



8.6 NAMED PIPES (PERSISTENTE PIPES)

Named Pipes haben den Filetyp p im Inode.

```
$ /etc/mknod channel p
$ ls -l channel
prw-rw-r-- 1      chregistaff 0 Sep 1 15:45 channel
$ cat < channel &
102
$ ls -l >> channel; wait
total 0
prw-rw-r-- 1      chregistaff 0 Sep 1 15:45 channel
$ rm channel
```

8.6.1 PROGRAMMIEREN MIT NAMED PIPES

```
char string[] = "hello";
#include <fcntl.h>
main (ac, av) int ac; char *av; {
      char buf[256];
      int fd;
      /* create named pipe with read/write
      permission for all users */
      mknod ("fifo", 010777, 0);
      if (ac == 2)
        fd = open ("fifo", O_WRONLY);
      else
        fd = open ("fifo", O_RDONLY);
      for (;;)
        if (ac == 2)
```

```
          write (fd, string, 6);
        else
          read (fd, buf, 6);
}
```

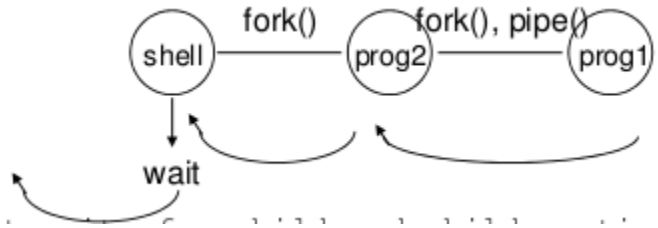
8.7 DUP()

dup() überschreibt den erstem freien Dateideskriptor in der User File Descriptor Tabelle eines Prozesses.

dup2() überschreibt den als 2. Argument genannten Dateideskriptor mit einer Kopie des als 1. Argument genannten Dateideskriptors (ob frei oder nicht).

8.8 BENUTZUNG VON PIPES IN DER SHELL

$ prog1 | prog2 führt zu folgendem Vorgang:



8.9 SYSTEMCALLS

pipe(), pipe2()	Erstellt eine Pipe, einen unidirektionalen Daten-Channel welcher für Interprozess Kommunikation verwendet werden kann.
popen(), pclose()	Prozess öffnen / schliessen indem eine Pipe erstellt, danach geforkt und die Shell aufgerufen wird. Per Definition ist eine Pipe unidirektional, deshalb kann nur Read oder Write als Typ angegeben werden.

9 SIGNALE

- Ursprünglich entwickelt um Ausnahmesituationen und Fehlverhalten von Prozessen anzuzeigen die zu einer Prozessbeendigung durch den Kernel führen.

- Später erweitert für asynchrone Prozesskommunikation und Anzeige von nicht kritischen Bedingungen (z.B. Terminieren eines Kindprozesses, Starten / Stoppen eines Prozesses für das Debugging, I/O auf einem Dateideskriptor usw.)

- Benutzt vom Kernel (als Sender), vom Benutzer (als Sender, z.B. Ctrl-C) und von Prozessen (als Sender mittels kill() oder als Empfänger)

9.1 POSIX.1-1990 SIGNALTYPEN

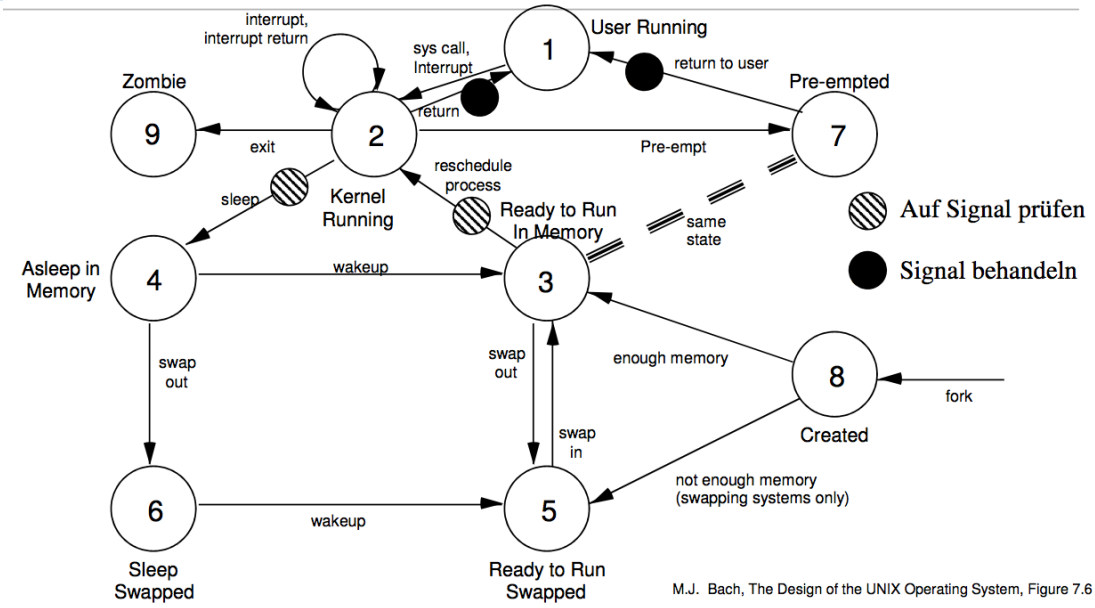
Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at tty
SIGTTIN	21,21,26	Stop	tty input for background process

SIGTTOU 22, 22, 27 Stop tty output for background process

SIGKILL und SIGSTOP können nicht blockiert oder ignoriert werden. Die Default Aktion für alle Signale ist die Prozesstermination. Einige Ausnahmen haben jedoch auch Ignorieren oder Stoppen als Standard.

Da die Signalnummern nicht einheitlich über alle Unix Systeme verwendet werden, ist es besser die Mnemonics anstelle der Nummern zu verwenden (SIGSTOP, SIGKILL ..).

9.2 SIGNALE UND PROZESSE



9.3 SYSTEMCALLS

signal(signum, handler)	Der Handler kann eine Funktion, oder auch SIG_IGN (Ignore) sein. Setzt den neuen Signal Handler und setzt den alten zurück
kill(signum)	Sendet ein Signal
alarm(seconds)	Löst ein Alarmsignal aus nach einer bestimmten Anzahl Sekunden
Setjmp(jmp_buf)	Sichert den Stack Kontext des Programms, gibt 0 zurück
Longjmp(jmp_buf, int)	Springt an den zuvor definierten jmp_buf und tut so als hätte das Setjmp an dieser Stelle den Integer im zweiten Parameter zurückgegebe n (0 ist nicht mögich)
sigwait(sigtype)	Auf bestimmtes Signal warten

9.4 SIGNALE UND FORK() / EXEC()

- Mit fork() kreierte Prozesse erben sämtliche Signale vom Elternprozess (behandelte und nicht behandelte Signale)
- Mit exec() werden sämtliche Signal Handler wieder zurückgesetzt da der gesamte Inhalt der Prozesshülle (Textsegment) ersetzt wird

9.5 PROGRAMMBEISPIEL – ELTERN- / KIND-PROZESS SIGNALISIERUNG

```
#include <signal.h>
main (argc, argv) int argc; char *argv[]; {
      int i, ret_val, ret_code;
      if (argc > 1)
        signal (SIGCLD, SIG_IGN);
      for (i = 0; i < 5; i++) {
        printf (“fork run %d\n”, i);
        if (fork() == 0) {
          /* child process here */
          printf (“child proc %x\n”, getpid());
          exit (i);
        }
      }
      ret_val = wait (&ret_code);
      printf (“wait ret_val %x ret_code %x\n”, ret_val, ret_code);
}
```

9.6 PROGRAMMBEISPIEL – WARTEN AUF ALLE KINDPROZESSE

```
#include <signal.h>
main (argc, argv) int argc; char *argv[]; {
      int i, ret_val, ret_code;
      if (argc > 1)
        signal (SIGCLD, SIG_IGN);
      for (i = 0; i < 5; i++) {
        printf “fork run %d\n”, i);
        if (fork() == 0) {
          /* child process here */
          printf (“child proc %x\n”, getpid());
          exit (i);
        }
      }
      while ((ret_val = wait (&ret_code)) >= 0)
       printf (“wait ret_val %x ret_code %x\n”, ret_val, ret_code);
}
```

9.7 PROGRAMMBEISPIEL – ALARM

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

#define TIMEOUT    5
#define TRUE 1
#define FALSE      0

static int timed_out;
int catch();
void (*was)();

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

      timed_out = FALSE;

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

      /* turn alarm off */
      alarm(0);

      /* if timed_out is TRUE, then the action took too long */
      if (timed_out == TRUE)
         /* warn the user or do whatever else is needed */
           printf ("Time overrun\n");
      else {
         /* restore old signal action */
         signal (SIGALRM, was);
           printf ("Just in time\n");
        }
}

catch (sig) int sig; {
      /* set timeout flag */
      timed_out = TRUE;
      /* reinitialise signal handler action */
      signal (SIGALRM, catch);
}
```

9.8 PROGRAMMBEISPIEL – SETJMP

```
#include <signal.h>
#include <setjmp.h>
#include <stdio.h>

jmp_buf position;
int goback ();

main () {

      setjmp (position);
      /* set signal action correctly */
      signal (SIGINT, goback);
      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);
      /* jump back to saved position */
      longjmp (position, 1);
}

display_main_menu () {
      fflush (stdout);
      printf ("Waiting in main menu: ");
      fflush (stdout);
        pause();
}
```

10 UNIX SOCKETS

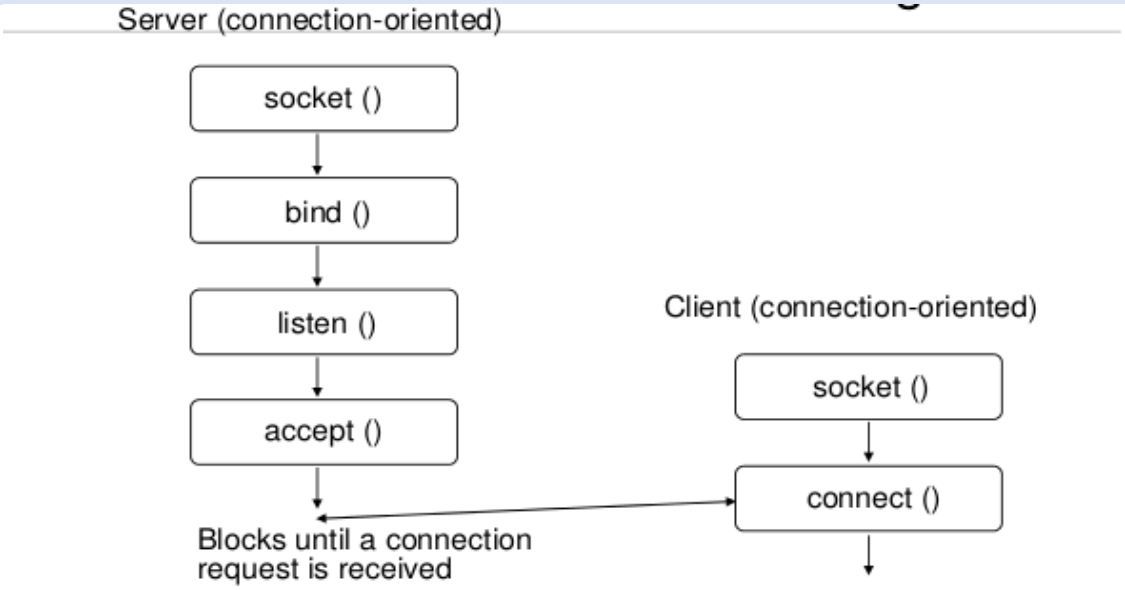
Die Entwicklung der Socket Schnittstelle wurde von der US Regierung finanziert und an der University of California at Berkeley durchgeführt. Ziel war es eine offene, generische Programmierschnittstelle für Interprozesskommunikation (lokal & über verschiedene Netzwerke) für UNIX zu realisieren. Sockets sind eine Analogie zu Postfächern oder Telefonsteckdosen. Sie erlauben es dem Nutzer sich mit dem Netzwerk zu verbinden. Ein Socket ist dabei Endpunkt einer Kommunikationsverbindung.

Anforderungen:

- Transparenz (bezüglich Netzwerkverhalten etc)

- Effizienz (um die Programmierer zu überzeugen)
- Kompatibilität (mit UNIX Standard I/O)
- Unterstützung verschiedener Netzwerkprotokolle → communication domains
- Klassifikation von Kommunikationseigenschaften → communication types
- Benennung und Adressierung von Kommunikationsendpunkten → name binding
- Einheitliche Abstraktion für Kommunikationsendpunkte in einem Programm / Prozess → sockets

10.1 HERSTELLEN EINER SOCKET VERBINDUNG



10.2 SENDEN / EMPFANGEN VON DATEN

Um möglichst viele Domänen (Unix, Internet ..) und Protokolle (TCP, UDP) zu unterstützen, gibt es verschiedene Möglichkeiten für Senden / Empfangen von Daten.

- **r / w:** Rückwärtskompatibilität zu UNIX Standard I/0, ermöglichen von Pipes mit dup()
- **send / recv:** zusätzliche Funktionalität und Parametrisierung, z.B. MSG_PEEK, MSG_OOB, nutzt TCP, Pakete geordnet
- **sendto / recvfrom:** erlaubt Senden ganzer Datenpartitionen (Datagramme) ohne Verbindung (UDP), ungeordnet, keine Fehlerprüfung, zwei Buffer nötig (Sender & Empfänger)

- **sendmsg / recvmsg:** nicht Bytestrom-orientiert, Applikation wird erst benachrichtigt wenn ganze Nachricht angekommen ist

In den meisten UNIX Varianten werden alle 4 Systemcall Paare auf eine einzige interne Funktion abgebildet.

10.3 BLOCKERENDES VERSUS SELEKTIVES WARTEN AUF SOCKET I/O

- accept() blockiert den aufrufenden Prozess solange, bis ein Verbindungsaufbauwunsch am Socket signalisiert wird.

- select() erlaubt nicht blockierendes Warten auf neue Verbindungen bzw. nur kurze Blockaden (z.B. über SIGALARM).

10.4 SCHLIESSEN EINER VERBINDUNG

- **shutdown()** signalisiert Verbindungsabbruch-Anforderung zur anderen Seite und wartet danach auf Bestätigung. Vor der Bestätigung eintreffende Daten werden noch bestätigt, jedoch nicht an die Applikation weitergegeben. Shutdown kann einseitig oder beidseitig erfolgen.

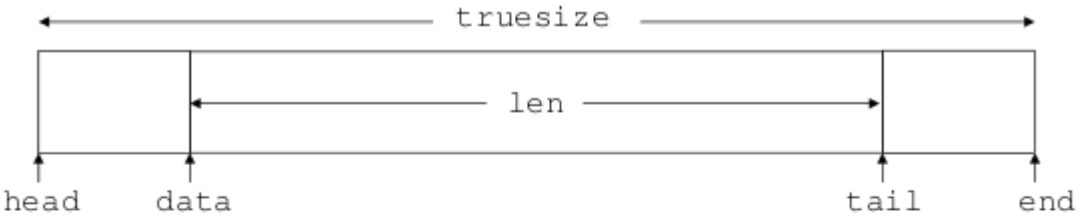
- **close()** deallozііert lokale Socket Datenstruktur und zugeordnete Ressourcen (Puffer)

- **exit()** terminiert das Programm ohne explizites shutdown / close, überlässt Aufräumarbeiten dem OS und der Gegenpartei.

- **Provider Abort** überlässt das Aufräumen dem OS beider Gegenparteien

10.5 SPEICHERVERWALTUNG AUF SOCKET SCHICHT

Daten auf einem Socket werden in sogenannten mbufs gespeichert, welche jeweils auf den folgenden mbuf verweisen (also sozusagen eine LinkedList). Alle mbuf zusammen sind der mbuf Pool.



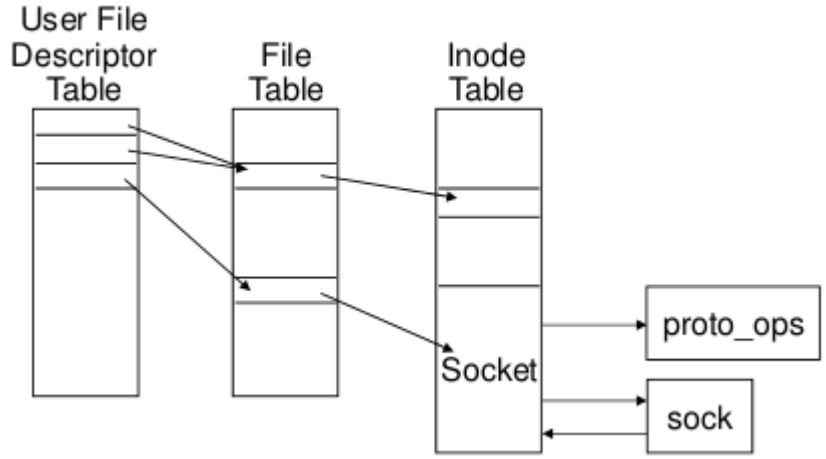

```
struct sk_buff {
        struct sk_buf *next;        /* Next buffer is list */
        struct sock   *sk;             /* Socket we are owned by */
        unsigned long len;             /* Length of actual data */
        unsigned int  truesize;        /* Buffer size */
        unsigned char * head;          /* Head of buffer */
        unsigned char *data;        /* Data head pointer */
        unsigned char *tail;        /* Tail pointer */
        unsigned char *end;            /* End pointer */
}
```

J. O'Gorman, Operating System with Linux, page 316/317

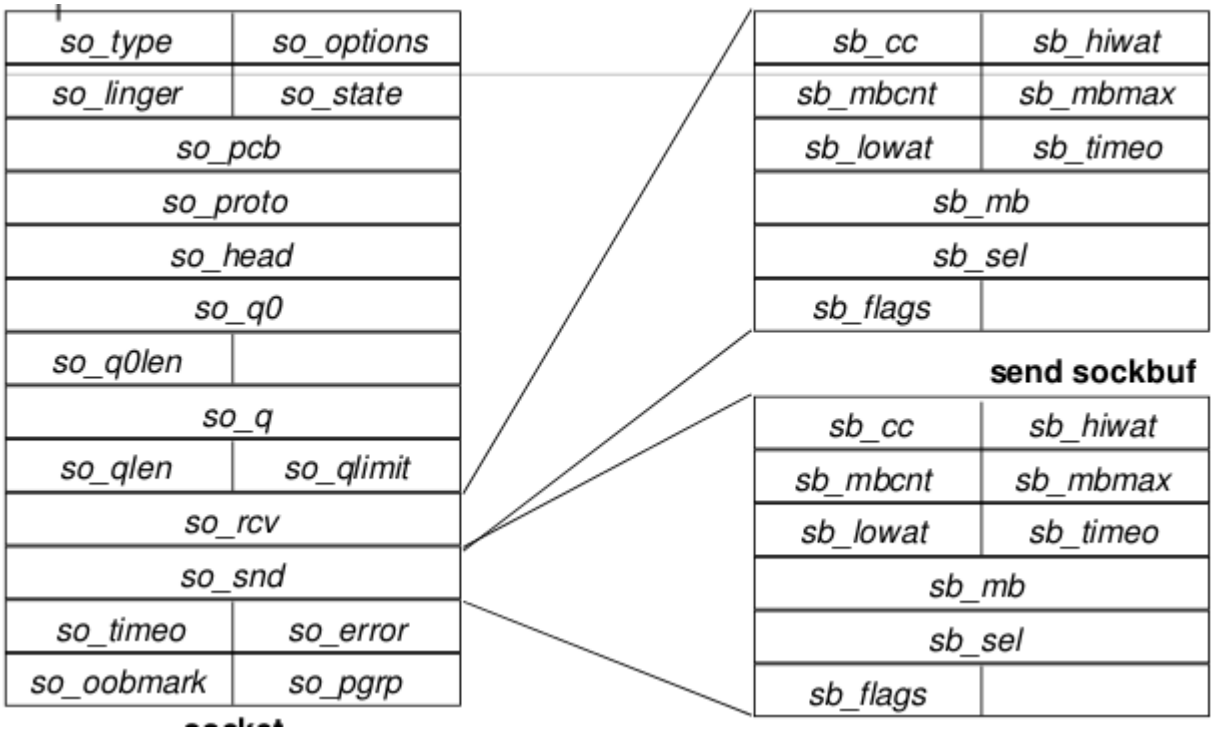
10.6 UNTERSTÜTZTE PROTOCOL / DOMAIN FAMILIES

PF_UNSPEC	Unspecified
PF_Unix	Local communication, e.g. pipes
PF INET	DARPA Internet (TCP / IP)
PF IMPLINK	1822 Input Message Processor link layer
PF PUP	Old Xerox network
PF CHAOS	MIT Chaos network
PF NS	Xerox Network System (XNS) architecture
PF NBS	National Bureau of Standards (NBS) network
PF ECMA	European Computer Manufacturers network
PF DATAKIT	AT&T Datakit network
PF CCITT	CCITT protocols, e.g. X.25
PF SNA	IBM System Network Architecture (SNA)
PF DFCnet	DEC network
PF DLI	Direct link interface
PF LAT	Local-area network terminal interface
PF_HYLINK	Network Systems Corporation Hyperchannel (raw)
PF_APPLETALK	Apple Talk Network
PF_IPX	Novell IPX

10.7 PROZESSZUGANG ZU SOCKETS

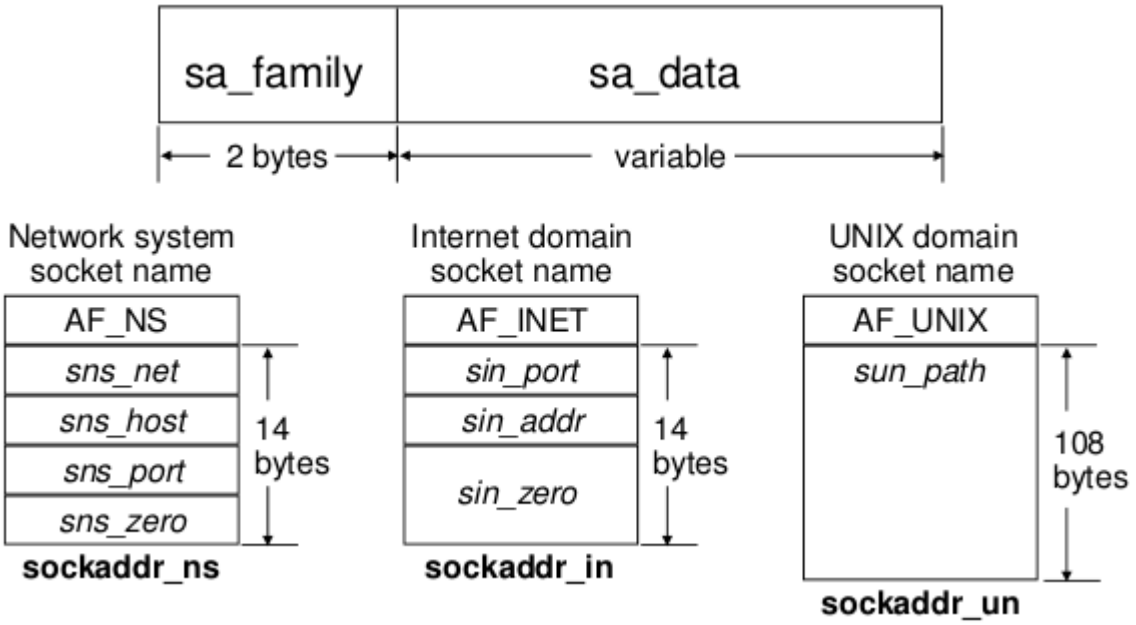


10.8 DATENSTRUKTUR



Socket

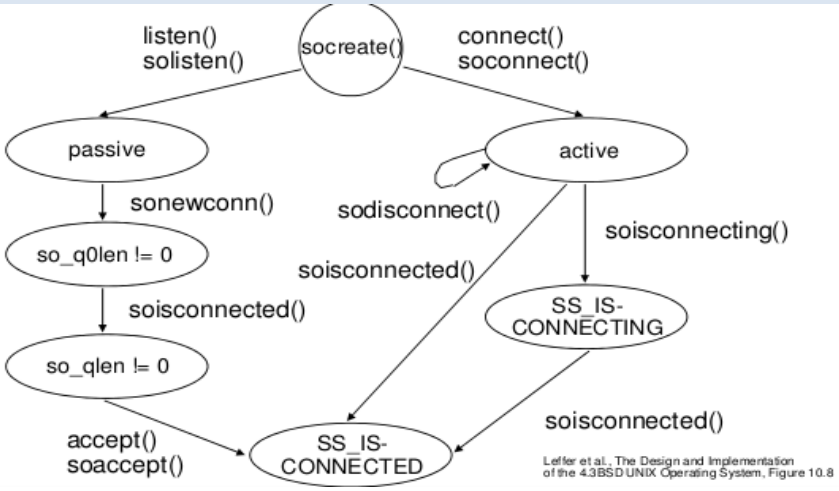
10.9 SOCKET ADRESSEN



10.10 UNTERSTÜTZTE SOCKET TYPEN

Name	Typ	Eigenschaften
SOCK_STREAM	Stream	reliable, sequenced data transfer, may support out-of-band data
SOCK_DGRAM	Datagram	Unreliable, unsequenced data transfer, with message boundaries preserved
SOCK_SEQPACKET	Seqeunced packet	Reliable, sequenced data transfer with message boundaries preserved
SOCK_RAW	raw	Direct access to the underlying communication protocols

10.11 SOCKET VERBINDUNGSSCHEMA



10.11.1 ABLAUF BEI STREAM SOCKETS

Client seitig:

1. Socket erstellen
2. erstellten Socket mit der Server-Adresse verbinden, von welcher Daten angefordert werden sollen
3. senden und empfangen von Daten
4. evtl. Socket herunterfahren (shutdown())
5. Verbindung trennen, Socket schliessen

Server-seitig:

1. Server Socket erstellen
2. binden des Sockets an eine Adresse (Port), über welche Anfragen akzeptiert werden
3. auf Anfragen warten
4. Anfrage akzeptieren und damit neues Socket Paar für diesen Client erstellen
5. bearbeiten der Client-Anfrage auf dem neuen Client-Socket
6. Client-Socket wieder schliessen

10.11.2 ABLAUF BEI DATAGRAM SOCKETS

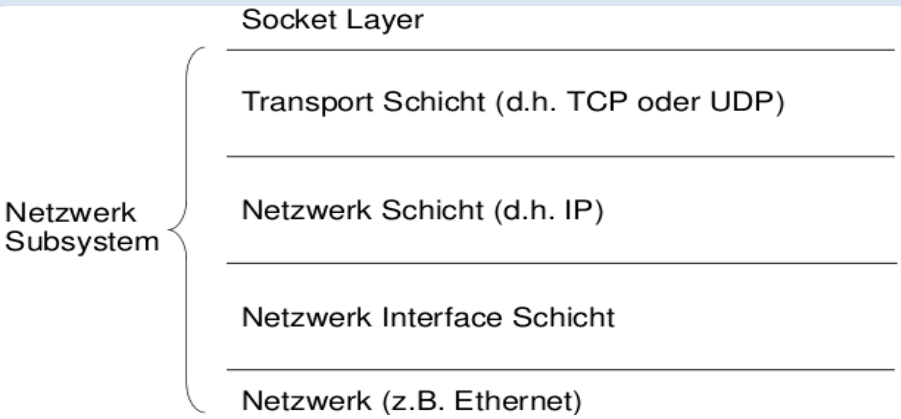
Client seitig:

1. Socket erstellen
2. An Adresse senden

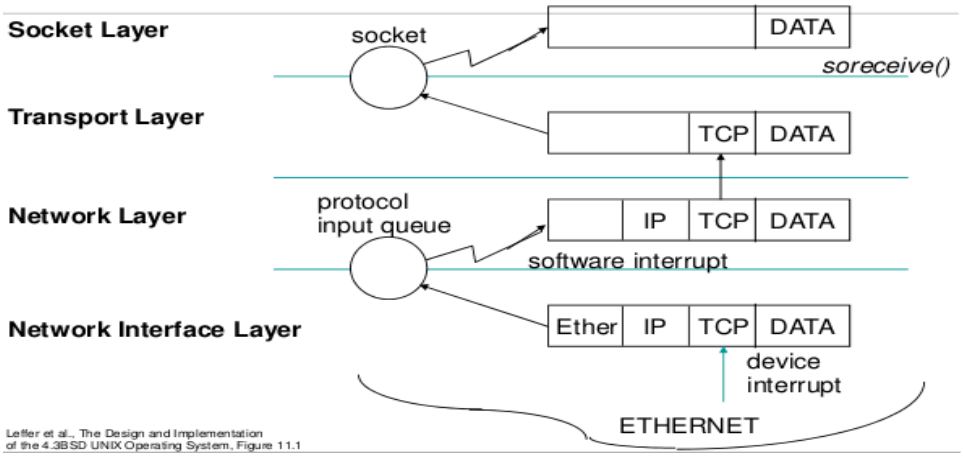
Server-seitig:

1. Socket erstellen
2. Socket binden
3. auf Pakete warten

10.12 SCHICHTENSTRUKTUR DER SOCKET SCHNITTSTELLE



10.12.1 DATENFLUSS DURCH DIE SCHICHTEN



10.13 SYSTEMCALLS

socket(domain, type, prot)	Socket erstellen
bind(sockfd, addr, addrlen)	Nach dem Erstellen eines Sockets mit socket() existiert dieser bloss als Name aber hat noch keine Adresse. Bind vernküpft ihn mit einer Adresse
listen(sockfd, backlog)	Markiert den Socket (sockfd) als passiven Socket, d.h. als Socket der einkommende Verbindungen mit accept entgegennehmen kann
accept(sockfd, addr, addrlen, flags)unique	Wird mit Verbindungsbasierten Socket Typen (SOCK_STREAM, SOCK_SEQPACKET) verwendet. Entpackt den ersten Connection Request aus der Queue der zu bearbeitenden Verbindungen für den „hörenden“ Socket sockfd. Kreiert einen neuen verbundenen Socket und gibt einen neuen File Descriptor zurück, welcher auf diesen verweist. Der ursprüngliche Socket sockfd wird nicht verändert. Der neu erstellte Socket ist nicht im listening state.
connect(sockfd, addr, addrlen)	Verbindet den Socket zur Adresse

11 INTERPROCESS COMMUNICATION (IPC)

11.1 ZENTRALE KONZEPTE

Es gibt in System V 3 verschiedene IPC Mechanismen, für deren Verwaltung der Kernel je eine separate Tabelle pflegt:

- Message Queues
- Shared Memory
- Semaphore

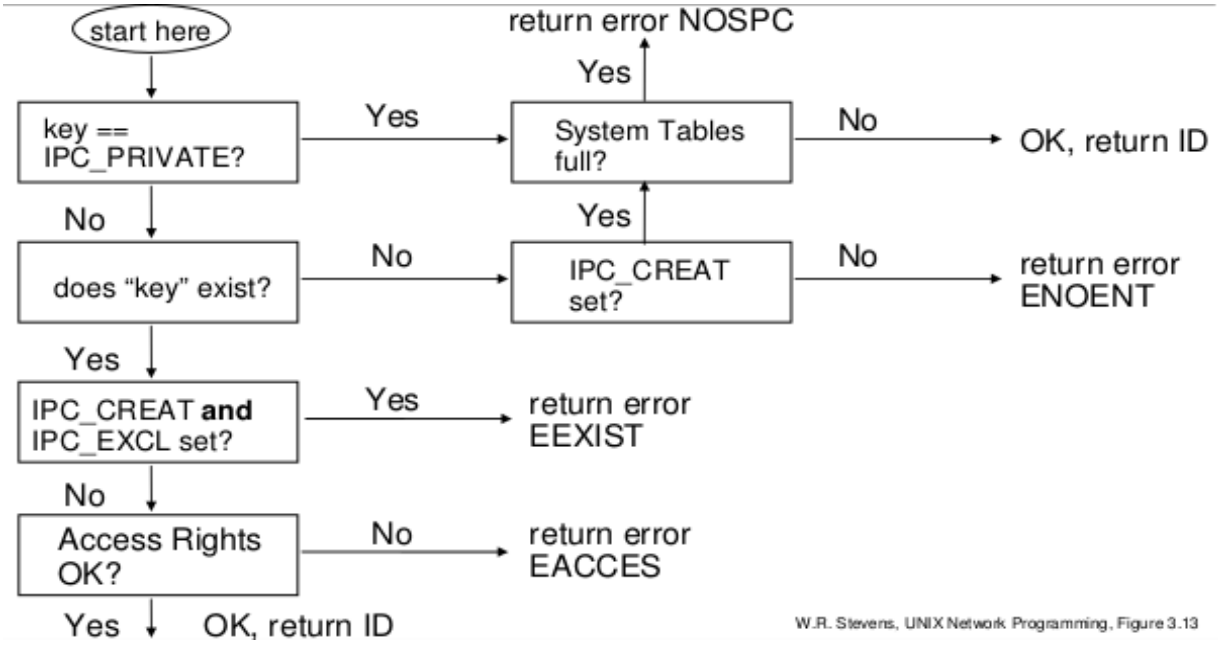
Für die Verwaltung aller 3 Mechanismen werden dieselben Prozeduren verwendet.

Der Zugriff erfolgt über numerischen Schlüssel.

Es gibt keine Registratur für verwendete / reservierte Schlüssel → Kollisionen möglich.

System V IPC Objekte sind NICHT kompatibel mit Standard I/O basierter Prozesskommunikation (Dateien, Sockets, Pipes, Geräte)

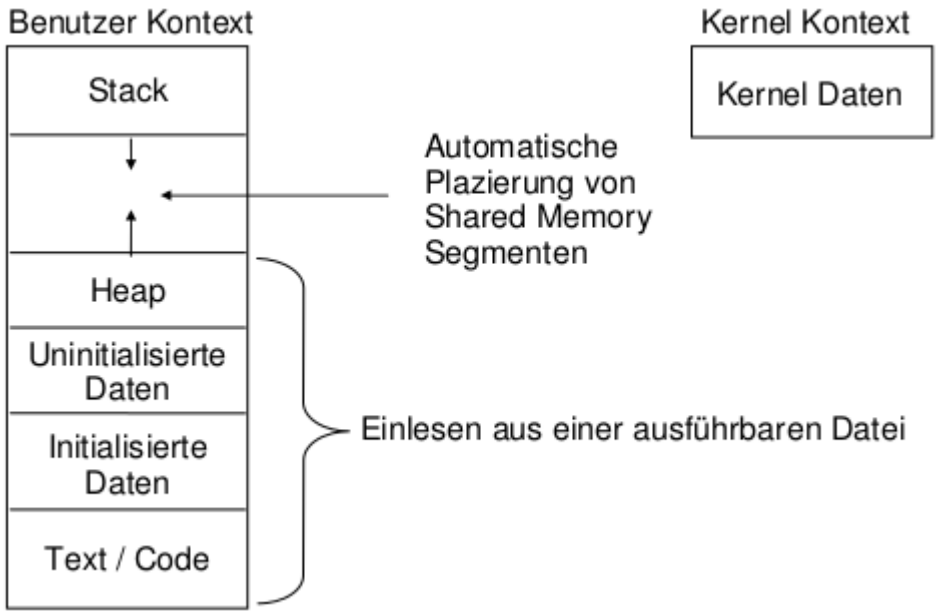
11.2 VERWALTUNG VON IPC OBJEKTEN



11.3 SHARED MEMORY

- Erlaubt gemeinsame Nutzung von Hauptspeicher-Seiten zwischen verschiedenen Prozessen
- Verwendung eines dedizierten Segment Typs der auf normaler Unix Speicherverwaltung basiert
- Nutzung zwischen nicht-verwnandten Prozessen möglich
- limitiert auf lokales System
- schnellste der IPC Mechanismen
- Synchronisation durch Semaphore, Signalbit oder allg. Signale

11.3.1 BEISPIEL-ADRESSRAUM EINES PROZESSES MIT SHARED MEMORY



11.3.2 ERSTELLEN EINES SEGMENTS (SHMGET)

Für das Allozieren eines Segments werden folgende Schritte durchgeführt:

1. Remove Region from linked list of free regions
2. assign region type

3. assign region inode pointer
4. if (inode pointer not null) increment inode reference count
5. place region on linked list of active regions
6. return (locked region)

11.3.2.1 CODEBEISPIEL

```
# include < sys / types .h > /* supplies key_t */
# include < sys / ipc .h >
/* supplies ipc_perm */
# include < sys / shm .h >
/* supplies structures and macros for
* shared mem . data structures etc . */

int size , permflags , shm_id ;
key_t key ;
...
shm_id = shmget ( key , size , permflags ) ;
```

11.3.3 EINBINDEN EINES SEGMENTS (SHMAT)

Wenn ein Segment eingebunden wird, werden folgende Schritte durchgeführt:

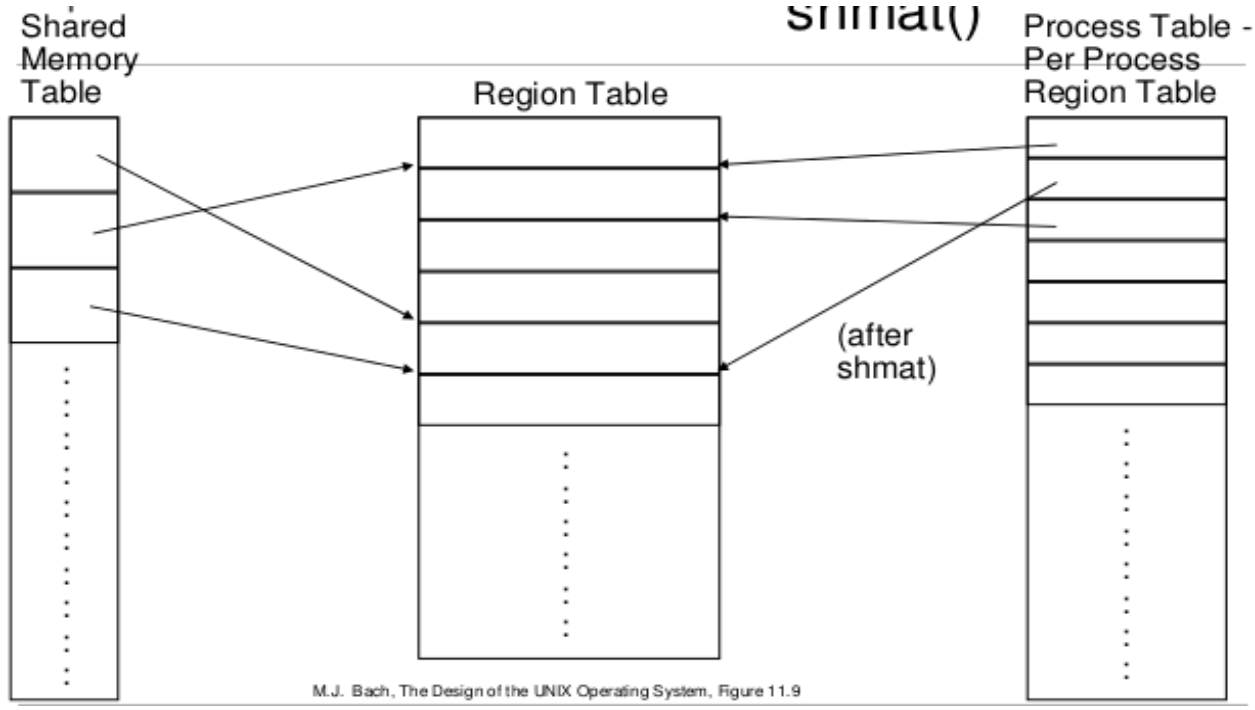
1. check validity of descriptor, permissions
2. if (user specified virtual address)
    a. round off virtual address, as specified by flags
    b. check legality of virtual address, size of region
3. else
    a. kernel picks virtual address: error if none available
4. attach region to process address space (algorithm attachreg)
5. if (region being attached for the first time)
    a. allocate page tables, memory for region (algorithm growreg)

6. return (virtual address where attached)

11.3.3.1 CODEBEISPIEL

```
# include < sys / types .h > /* supplies key_t */
# include < sys / ipc .h >
/* supplies ipc_perm */
# include < sys / shm .h >
/* supplies structures and macros for
* shared mem . data structures etc . */
int shm_id , shmflags ;
char * memptr , * daddr , * shmat () ;
...
memptr = shmat ( shm_id , daddr , shmflags ) ;
```

11.3.3.2 DATENSTRUKTUREN NACH SHMAT()



11.3.4 ENTFERNEN EINES SEGMENTS (SHMDT)

Wenn ein Segment entfernt wird, werden folgende Schritte durchgeführt:

1. get auxiliary memory management tables for process

2. release as appropriate
3. decrement process size
4. decrement region reference count
5. if (region count is 0 and region not sticky bit)
    a. free region (algorithm freereg)
6. else
    a. free inode lock, if applicable (inode associated with region)
    b. free region lock

11.3.4.1 CODEBEISPIEL

```
int
retval ;
char
* memptr ;
...
retval = shmdt ( memptr ) ;
```

11.3.5 SYSTEMCALLS ÜBERSICHT

shmget(key, size, shmflag)	Gibt ID des Shared Memory Segments entsprechenden
shmat(shmid, shmaddr, shmflg)	Attach: Fügt das Shared Memory Segment zum Adressspace des aufrufenden Prozesses hinzu
shmdt(shmaddr)	Detach: Entfernt das Shared Memory Segment vom Adressspace des aufrufenden Prozesses
shmctl(shmid,cmd,*buf)	Führt cmd auf dem Shared Memory Segment aus

11.3.6 BEISPIELPROGRAMM – IPC MIT SHARED MEMORY

```
#include <stdio.h>
#include <sys/types.h>   /* supplies key_t */
#include <sys/ipc.h>     /* supplies ipc_perm */
#include <sys/shm.h>     /* supplies structures and macros for
                  * shared mem. data structures etc. */
```

```
#define SHMKEY    75
```

```
#define K   1024
#define MAXCNT    10

int shmid;
main() {
    int i, *pint;
    int pid;
    char *addr1, *addr2;
    struct shmid_ds shm_status;
    extern cleanup();
    extern void shmstat_print();
    for (i=0; i<32; i++)
        signal (i, cleanup);
    shmid = shmget(SHMKEY, 128 * K, 0777|IPC_CREAT);
    shmctl(shmid, IPC_STAT, &shm_status);
    shmstat_print(SHMKEY, shmid, &shm_status);

    pid = fork();

    if (pid > 0) {
        printf("\nParent process started. \n");

        addr1 = shmat(shmid, 0, 0);
        printf ("\n      addr1 0x%x \n", addr1);


      /* Fill the first 10 Bits of Memory */
        pint = (int *) addr1;
        for (i = 0; i<MAXCNT; i++)
          *pint++ = i;
        pint = (int *) addr1;
        *pint = MAXCNT;

        shmdt(addr1);


        shmctl (shmid, IPC_STAT, &shm_status);
        shmstat_print(SHMKEY, shmid, &shm_status);
      wait();
      printf("\nParent process finished. \n");
      exit(0);
    }
    else if (pid == 0) {

        printf("\nChild process started\n");
```

```
        addr2 = shmat(shmid, 0, 0);
        printf ("\n      addr2 0x%x \n", addr2);
```

```
        pint = (int *) addr2;
      /* Wait for memory to be filled */
      while (*pint == 0) {
          sleep(2);
      }


      /* Read out the memory and print it to the screen */
        printf ("\n");
        for (i=0; i < MAXCNT; i++)
          printf ("      index %d\tvalue %d\n", i, *pint++);

    shmctl (shmid, IPC_STAT, &shm_status);
      shmstat_print(SHMKEY, shmid, &shm_status);

        shmdt(addr2);

        shmctl(shmid, IPC_STAT, &shm_status);
        shmstat_print(SHMKEY, shmid, &shm_status);
      cleanup();
      printf("\nChild process finished.\n");
      exit(0);
    }

 else {
      cleanup();
      printf("\nfork failed\n");
      exit(-1);
   }

}

cleanup () {
    shmctl(shmid, IPC_RMID, 0);
}


shmstat_print (mkey, shmid, sstat)
    key_t mkey; int shmid;
    struct shmid_ds *sstat; {
    char *ctime();
    char *atime;
    printf ("\nKey %d, shmid %d, ", mkey, shmid);
    printf ("Last change %s", ctime (&(sstat->shm_ctime)));
    printf ("Access rights %o\n", sstat->shm_perm.mode);
```

```
    printf ("uid of creator %d, gid of creator %d, ",
    sstat->shm_perm.cuid, sstat->shm_perm.cgid);
    printf ("uid of owner %d, gid of owner %d\n",
    sstat->shm_perm.uid, sstat->shm_perm.cgid);
    printf ("size of segment %d, pid of last semop %d\n",
    sstat->shm_segsz, sstat->shm_lpid);
    printf ("pid of creator %d, ", sstat->shm_cpid);
    printf ("region attach counter %d\n", sstat->shm_nattch);
    atime= ctime (&(sstat->shm_atime));
    atime[strlen(atime)-1] = '\0';
    printf ("Last shmat %s, ", atime);
    printf ("Last shmdt %s", ctime (&(sstat->shm_dtime)));
}
```

11.4 SEMAPHOR

Ein Semaphor ist zwar kein Mechanismus für die Datenübertragung, er kann jedoch hilfreich sein für die Synchronisierung der Prozesse. Vereinfacht gesprochen ist ein Semaphor ein Zähler, dessen Wert entscheidet, ob eine Ressource exklusiv nutzbar ist oder nicht. Ein Semaphor besteht aus einem Zähler und einer Warteschlange für die Aufnahme blockierter Prozesse:

```
struct Semaphor {
   int zaehler;
   Queue queue; /* Warteschlange */
};
```

Sowie aus 3 Funktionen:

- Initialisierungsfunktion
- **P()** - prolaag: prüfen, dabei wird der Semaphor dekrementiert, falls er danach noch grösser 0 ist, setzt der Prozess seine Arbeit fort, ansonsten wartet er.
- **V()** - verhoog; erhöhen, dabei wird der Semaphor wieder inkrementiert und so für eventuell wartende Prozesse freigegeben

Da Edsger W. Dijkstra – der Erfinder der Semaphore dieser Art – ein Holländer war, sind diese beiden Funktionen leider nicht so einfach zu merken.

11.4.1 NUTZUNG FÜR GEGENSEITIGEN AUSSCHLUSS

```
s: semaphore (1);

P1 : process                          P2 : process
      ...                                   ...
      P(s);                                 P(s);
      ... -- critical section               ... - critical section
      V(s);                                 V(s);
      ...                                   ...
end  process                          end  process
```

11.4.2 NUTZUNG FÜR PROZESS-SYNCHRONISATION

```
s: semaphore (0);

P1 : process                          P2 : process
      ...                                   ...
      V(s); -- signal event                 P(s); -- wait for event
      ...                                   ...
end  process                          end  process
```

11.4.3 ADDITIVE SEMAPHORE

```
exclusion : add_semaphore (N);

      type reader = process
      ...
      P (exclusion, 1);
           ... - read within critical section
      V (exclusion, 1);
      end process;

      type writer = process
      ...
      P (exclusion, N);
           ... -- write within critical section
      V (exclusion, N);
      end process;
```

11.4.4 SEMAPHORE IN UNIX

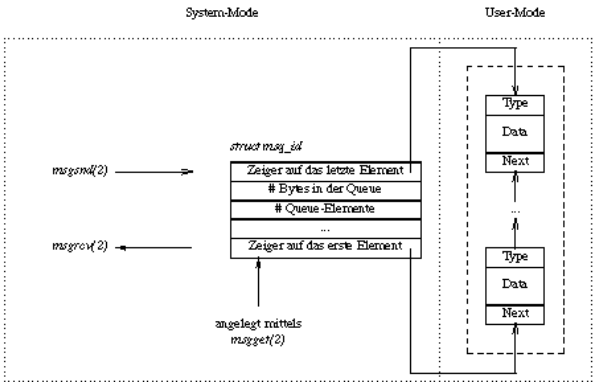
```
P(sem)
      if (sem != 0)
         decrement sem value by one
      else
         wait until sem becomes non-zero


V (sem)
      if (queue of waiting processes not empty)
         restart first process in wait queue
      else
         increment sem value by one
```

11.5 MESSAGE QUEUE

Message Queues dienen zum Versenden von Datenmengen von einem Server an einen Client. Eine Message Queue ist eine verkettete Liste von Listenelementen (messages) in der Form

```
struct msg_type {
      long type;
      char data[LENGTH];
}
```

wobei die Liste vom Listenkopf (Message Queue) aus kontrolliert wird. Mit msgsend() werden Elemente an die Schlange angefügt, mit msgrcv() werden sie wieder entfernt. Es können beliebig strukturierte Daten ausgetauscht werden. Auch die Message Queue erlaubt Kommunikation zwischen nicht verwandten Prozessen, jedoch ebenfalls mit der Einschränkung, dass es nur auf dem lokalen System funktioniert.



11.5.1 ERZEUGEN EINER MESSAGE QUEUE (MSGGET)

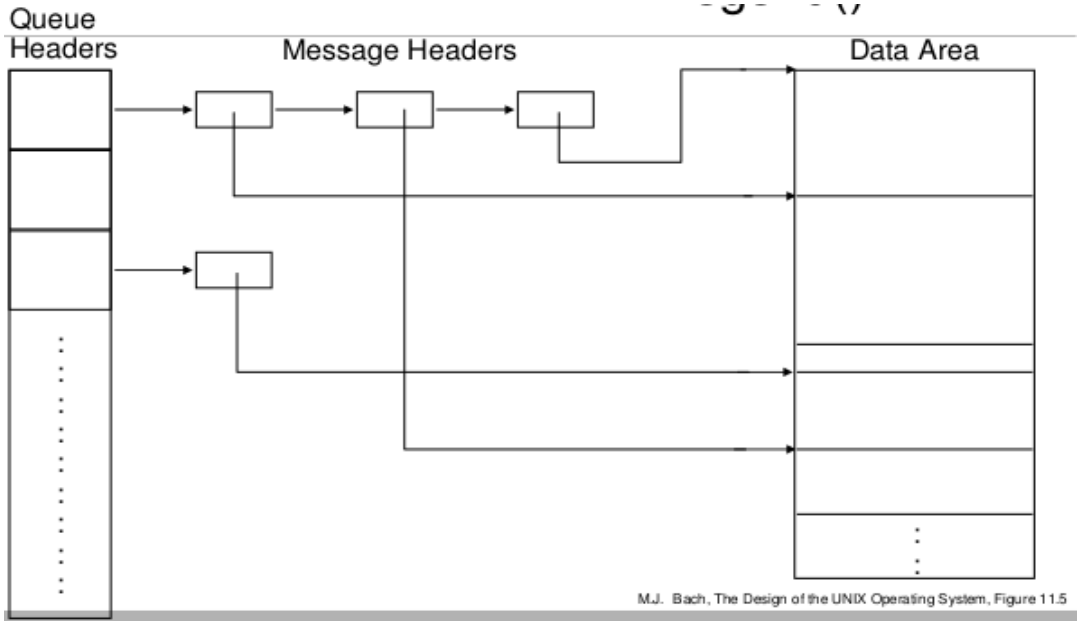
Durch den Aufruf von msget() besorgt sich ein Prozess den Zugriff auf eine message queue id. Mit msgctl() kann diese gelöscht, gelesen, modifiziert, gesperrt oder entsperrt werden.

```
#include <sys/types.h> /* supplies key_t */
#include <sys/ipc.h>
/* supplies ipc_perm */
#include <sys/msg.h>
/* supplies structures and macros
* for Message Queues, Messages etc. */
int msg_qid, permflags;
key_t key;
...
msg_qid = msgget (key, permflags);
```

11.5.2 SENDEN EINER NACHRICHT (MSGSND)

```
#include <sys/types.h> /* supplies key_t */
#include <sys/ipc.h>
/* supplies ipc_perm */
#include <sys/msg.h>
/* supplies structures and macros
* for Message Queues, Messages etc. */
int msg_qid, size, flags, retval;
struct my_msg {
long mtype;
char mtext[SOMEVALUE];
} message;
...
retval = msgsnd (msg_qid, &message, size, flags);
```

11.5.3 DATENSTRUKTUREN NACH MSGSND()



11.5.4 EMPFANG EINER NACHRICHT (MSGRCV)

```
#include <sys/types.h> /* supplies key_t */
#include <sys/ipc.h>
/* supplies ipc_perm */
#include <sys/msg.h>
/* supplies structures and macros
* for Message Queues, Messages etc. */
int msg_qid, size, flags, retval;
struct my_msg {
long mtype;
char mtext[SOMEVALUE];
} message;
long msg_type;
...
retval = msgrcv (msg_qid, &message, size, msg_type, flags);
```

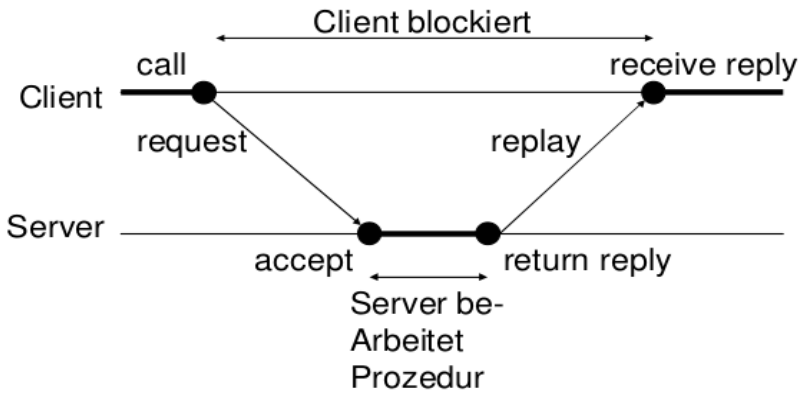
11.5.5 MESSAGE QUEUE KONTROLLIEREN

```
#include <sys/types.h> /* supplies key_t */
#include <sys/ipc.h>
/* supplies ipc_perm */
#include <sys/msg.h>
/* supplies structures and macros
* for Message Queues, Messages etc. */
int msg_qid, command, retval;
```

```
struct msqid_ds msq_stat; /* mess. queue id data structure */
...
retval = msgctl (msg_qid, command, &msq_stat);
```

12 REMOTE PROCEDURE CALLS (RPC)

RPC ist eine Technik zur Realisierung von Interprozesskommunikation zwischen verschiedenen Computern. Es gibt viele verschiedene Implementierungen welche meist nicht untereinander kompatibel sind. Die am weitesten verbreitete ist ONC RPC (oft auch nach dem Erfinder einfach Sun RPC genannt).

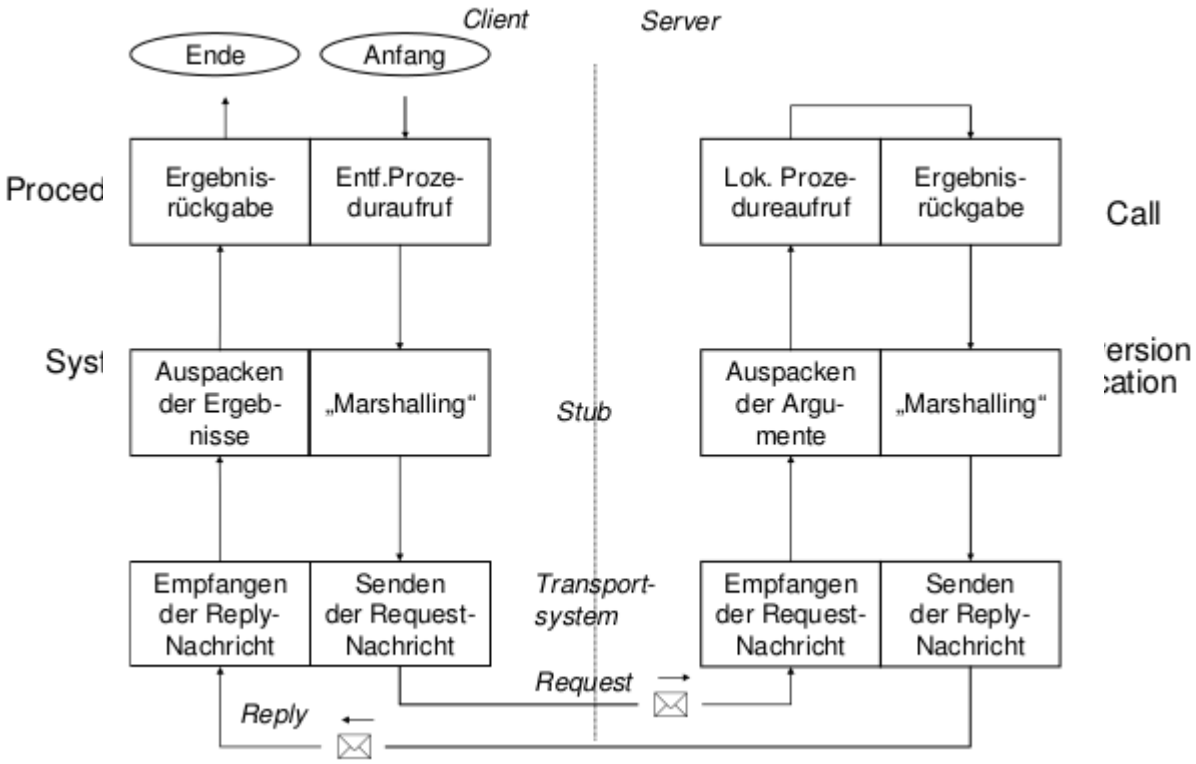


12.1 FUNKTIONSWEISE

Der Client ruft eine entfernte Prozedur auf einem Server auf. Die zur Bearbeitung benötigten Parameter werden an die Client-Stub Funktionsstelle des RPC Systems geschickt. Die Stub verpackt alle Funktionsparameter in eine komplexe Datenstruktur im NDR Format (Network Data Representation) – dieser Vorgang wird auch „Marshalling“ genannt.

Danach beauftragt die Stub das System mit der Übertragung der Nachricht an den Server. Der Client wartet nun auf eine Antwort vom Server und ist blockiert. Bekommt die Client-Stub eine Nachricht vom Server zurück, wird diese dekodiert („unmarshalling“) und an die Nutzer-Applikation zurückgegeben.

Wenn der Server eine Nachricht erhält, wird diese vom Betriebssystem an die Server-Stub weitergeleitet. Dort werden die Parameter entpackt („unmarshalling“) und die entsprechende Prozedur aufgerufen. Das zurücksenden der Ergebnisse ist äquivalent.



12.2 PARAMETERÜBERGABE

– Sun RPC erlaubt nur einen (strukturierten) Argument- oder Resultatwert

– Komplexe Argumente oder Resultate müssen in ein struct verpackt werden

12.3 SERVER – CLIENT BINDUNG

• Finden eines Servers im Netz

• Finden des gewünschten Service auf dem Server im Netz

• Sun RPC benutzt die Standard-Unix-Methode für das Finden von Servern im Internet (**gethostbyname**, DNS), einige Implementationen unterstützen auch die **clnt_broadcast()** Bibliotheksfunktion um Server zu finden

• Alle Serverprogramme, Programmversionen und angebotenen Prozeduren werden mit eindeutigen Nummern gekennzeichnet

• Ein Prozess kann eine oder mehrere Prozeduren anbieten

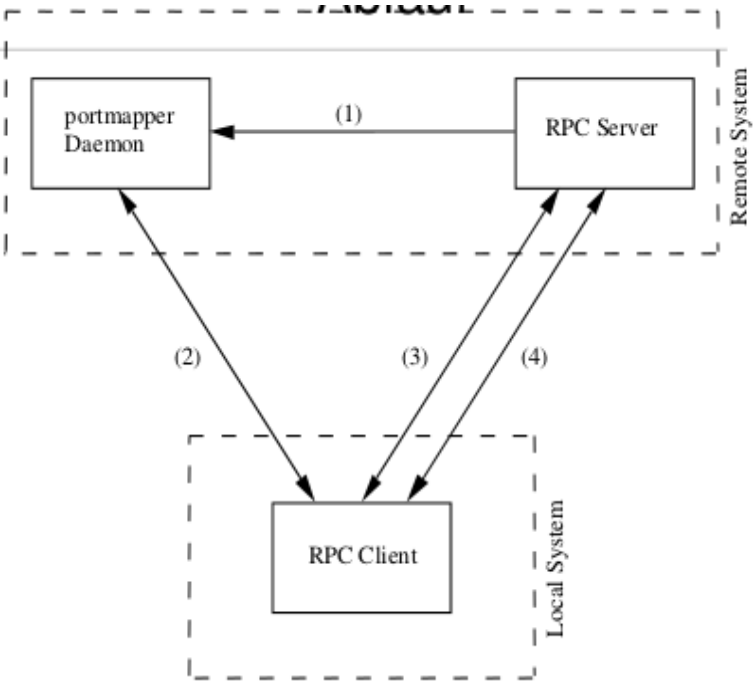
- Der **portmapper** Prozess (Linux: **rpcbind**) auf **Port 111** auf jedem Serversystem, dient als zentrale, lokale Registratur für verfügbare RPC-Dienste. Einzelne Server-Prozesse werden entweder manuell oder über den inetd Serverprozess gestartet.

12.4 RCP PORTMAPPING ABLAUF

1. Server erstellt Socket und registriert Programmnummer, Versionsnummer und Portnummer beim Portmapper

2. Klient kontaktiert Portmapper und fragt nach Programm-, Versions- und Prozedurnummer. Falls lokal bekannt, sendet Portmapper die Portnummer zurück.

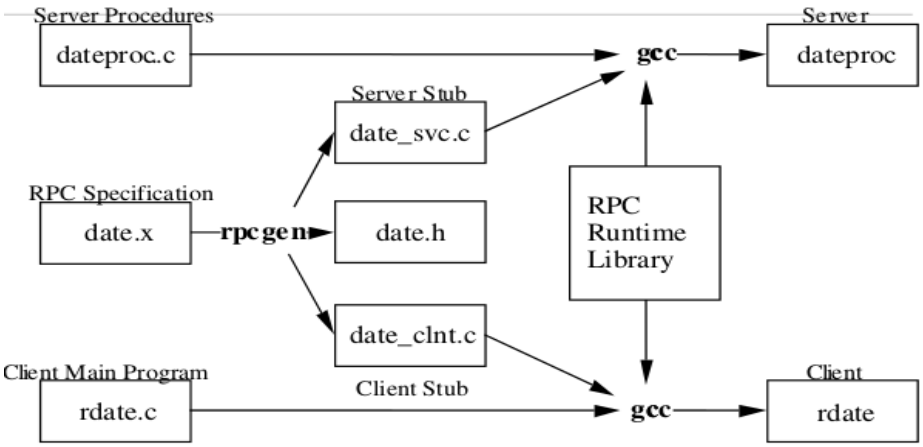
3. & 4. Klient kann gewünschte Prozedur direkt bei Server aufrufen



12.5 AUSWAHL DES TRANSPORTPROTOKOLLS

- RPC unabhängig von spezifischen Transportdiensten bzw. -protokollen
- Abbildungen auf übliche Transportprotokolle werden angeboten
- Sun RPC unterstützt Nutzung von TCP und UDP
- Bei Verwendung von UDP ist die Grösse von Argument und Resultat auf je 8192 Byte begrenzt.

12.6 RPC PROGRAMMIERBEISPIEL



12.7 REGISTRATION DER PROGRAMMNUMMER

- Verteilte Verwaltung des Namensraums
- Siehe /etc/rpc

- 0x00000000 – 0x1FFFFFFF: verwaltet durch ONC (Open Network Computing)
- 0x20000000 – 0x3FFFFFFF: Benutzerdefiniert
- 0x40000000 – 0x5FFFFFFF: Dynamisch zugeordnet
- 0x60000000 – 0xFFFFFFFF: Reserviert für künftige Nutzung

12.8 AUSNAHMEBEHANDLUNG

– Sun RPC verwendet automatisches Neusenden von Anfragen für UDP und erkennt verlorene Verbindungen in TCP (keepalive)

– es gibt verschiedene Szenieren die Fehler verursachen können (Server down, Client down)

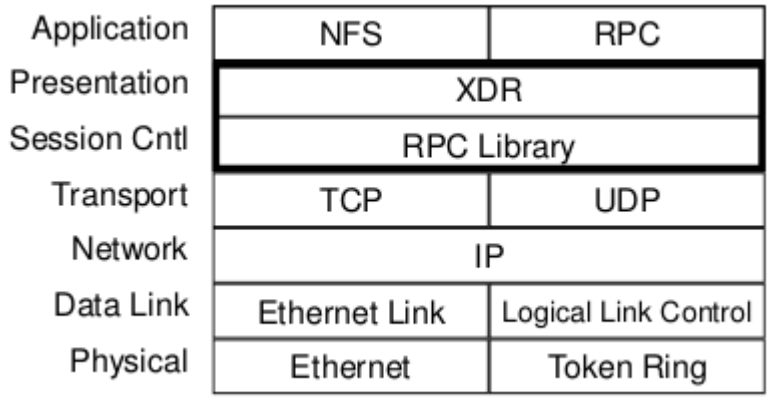
– Server unterhält RPC Cache mit Prozessaufruf und zurückgesendeten Resultaten

– Sun RPC verwendet eindeutige Transaktionsnummern für jeden RPC

12.8.1 AUFRUFSEMANTIK

Es gibt die Möglichkeit einem RPC mitzuteilen wie oft er im Fehlerfall aufgeruft werden soll:

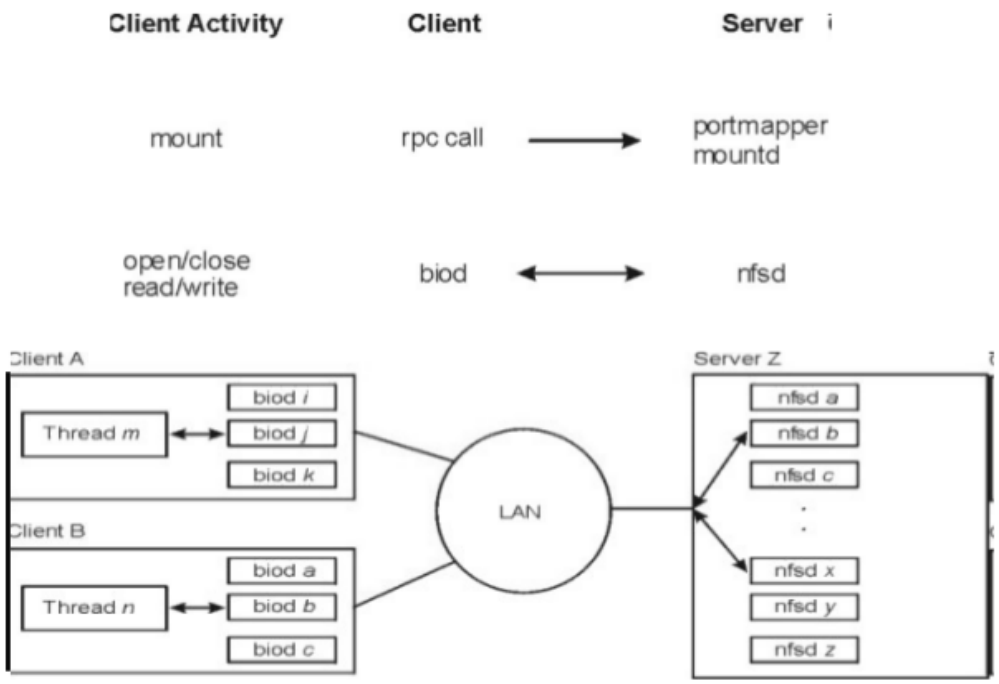
- genau einmal – genau einmal ausführen
- höchstens einmal – wurde der RPC komplett übermittelt aber bei Ausführung unterbrochen → noch einmal ausführen, wurde RPC nicht komplett übermittelt → nicht mehr ausführen
- mindestens einmal – Request solange wiederholen bis akzeptiert



12.9 SCHICHTENMODELL VON RPC

13 NETWORK FILE SYSTEM (NFS)

Das Network File System ist das Unix Pendant zum SMB auf Windows und dafür zuständig dass Rechner auf die Filesysteme anderer Rechner zugreifen können. NFS läuft über RPC.


Client Activity
Client
Server
mount
rpc call
portmapper
mountd
open/close
read/write
biod
nfsd
Client A
Thread m
biod i
biod j
biod k
Client B
Thread n
biod a
biod b
biod c
LAN
Server Z
nfsd a
nfsd b
nfsd c
nfsd x
nfsd y
nfsd z

14 GLOSSAR

14.1 ALLOKATION

In der Informatik ist damit die Reservierung von Hauptspeicher oder anderer Ressourcen zur eigenen Verwendung gemeint.

14.2 B-BAUM

Ein B-Baum (englisch B-Tree) ist eine Daten- / Indexstruktur welche häufig in Datenbanken und Dateisystemen eingesetzt wird. Ein B-Baum ist immer vollständig balanciert. Daten sind sortiert und nach Schlüsseln gespeichert. Einfügen, Suchen und Löschen von Daten ist in amortisiert logarithmischer Zeit möglich. Anders als viele Suchbäume wachsen und schrumpfen B-Bäume von den Blättern hin zur Wurzel.

14.3 BUFFER CACHE

Der Buffer Cache ist dafür da die Lese- und Schreibperformance zwischen dem Filesystem und den verschiedenen Geräten zu erhöhen.

14.4 FILE DESCRIPTOR (FD)

In der Systemprogrammierung ist ein File Descriptor ein abstrakter Indikator für den Zugriff auf ein File. Der Begriff stammt aus dem POSIX Standard. Ein File Descriptor ist vom Typ C Integer. Es gibt 3 Standard POSIX File Deskriptoren analog zu den 3 Standard Streams welche jeder Prozess haben sollte:

- 0 → Standard Input (stdin)
- 1 → Standard Output (stdout)
- 2 → Standard Error (stderr)

14.5 INODE (INDEX NODE)

Jedes Objekt eines UNIX Filesystems wird durch eine inode repräsentiert. Es enthält Informationen über ein reguläres File oder ein anderes Objekt im Filesystem.

14.6 PAGING

Das Vorgehen um eine Page des physischen Speichers vom oder zum Swap Device zu transferieren. Eine Page ist normalerweise in der Grössenordnung von 4 KB.

14.7 PHYSICAL MEMORY (RAM)

Als physikalischer Speicher wird der RAM Speicher bezeichnet welcher auf dem Motherboard vorhanden ist. Es beinhaltet NICHT den CPU Cache, RAM der Grafikkarte oder anderen Speicher von Controller Karten etc.

14.8 SWAPPING

Das Vorgehen ein gesamtes Programm (Code & Daten) zwischen dem physischen Speicher und dem Swap Space zu verschieben. In der Regel gibt es nur sehr wenige UNIX Systeme die dieses Vorgehen effektiv nutzen.

14.8.1 DESPERATION SWAPPING

Ist wenn der VM-Handler keine Pages zum Swap Out identifizeren kann und deshalb mit Umsortieren beginnt um ganze Programme aus zu swappen. Wird nur in wirklich vmkritischen Situationen ausgeführt.

14.9 SUPERBLOCK

Jedes Filesystem hat einen Superblock der folgende Informationen enthält:

- Filesystem Typ
- Grösse
- Status
- Information über weitere Metadaten

Wenn der Superblock eines Dateisystems verloren geht kann das entsprechende Filesystem nicht mehr gelesen werden. Deshalb gibt es jeweils Backups davon an verschiedenen Stellen des FS.

Superblockpositionen anzeigen: `dumpe2fs /dev/hda3 | grep -i superblock`

14.10 SWAP SPACE (AKA PAGING SPACE, AKA PAGING AREA)

Der Platz auf einer Disk der genutzt wird um den physikalischen Speicher eines Systems zu erweitern. Dadurch wird es möglich auf einem System mit 64MB Ram ein 100MB Programm zu starten. Mit seltenen Ausnahmen sollten alle UNIX Systeme mindestens einen Swap Space haben.

14.11 TRASHING

Als Trashing wird der Vorgang bezeichnet wenn das System gezwungen ist mehr Zeit und Leistung für Memory Management aufzubringen als dafür Anwendungen auszuführen. Der Begriff wird leider ziemlich weitläufig und oft auch falsch verwendet.

14.12 VIRTUAL MEMORY

Der gesamte Speicherplatz der dem UNIX Kernel zur Verfügung steht. Dies beinhaltet den physikalischen Speicher, sowie alle Swap Devices.

- B-Baum, http://de.wikipedia.org/wiki/B-Baum
- C (Programmiersprache), http://de.wikipedia.org/wiki/C_%28Programmiersprache%29
- Embedded System, http://de.wikipedia.org/wiki/Eingebettetes_System
- Imperative Programmierung, http://de.wikipedia.org/wiki/Imperative_Programmierung
- POSIX, http://en.wikipedia.org/wiki/POSIX
- SUS, http://en.wikipedia.org/wiki/Single_UNIX_Specification

15 QUELLEN

– Sämtliche Folien und zusätzlichen Materialien (Beispielprogramme, Weblinks, PDFs) von Prof. Dr. Hannes Lubich aus dem Modul Systemprogrammierung der Fachhochschule Nordwestschweiz

– syspr Zusammenfassungen von Jan Fässler https://github.com/janfaessler/FHNW-Informatik-Zusammenfassungen/tree/master/syspr

– Wikipedia

– What is a Pointer? http://pw1.netcom.com/~tjensen/ptr/ch1x.htm

– UNIX FAQ, Virtual Memory http://216.147.18.102/unixfaq/explain_vm.shtml

– Understanding UNIX / Linux Files System http://www.cyberciti.biz/tips/understanding-unixlinux-file-system-part-i.html

– The Linux System Administrator's Guide http://www.tldp.org/LDP/sag/html/index.html

– The Second Extended File System www.nongnu.org/ext2-doc/ext2.html

– Anatomy of the Linux File System http://www.ibm.com/developerworks/library/l-linux-filesystem/

– How to Use C Mutex Lock Examples for Linux Thread Synchronization http://www.thegeekstuff.com/2012/05/c-mutex-examples/

– Pipes versus FIFOs http://www.informatik.uni-osnabrueck.de/um/95/95.2/ipc/ipc.html