1 Überblick

Thema: "Parallele Programmierung in C unter Unix"

- Grundlegende Betriebssystem-Features für die Programmierung komplexer Programmsysteme am Beispiel von C und Unix
- Nicht Programmiersprache, nicht Algorithmen
- Warum Unix, nicht Windows:
  - Seit vielen Jahren etabliert und unverändert
  - Relativ einfach, wohldokumentiert, nahe an den allgemeinen Grundkonzepten
  - Zwischen verschiedenen Unix weitgehend portabel (Aufrufe standardisiert: POSIX, XOpen)
  - Unter Windows: Vergleichbare Grundkonzepte — ganz andere Aufrufe
- Warum C, nicht C++, Java, ...:
  - Konzepte in C++ noch nicht etabliert
  - Server traditionell in C geschrieben, nicht in C++

Inhalt:

1. Wiederholung: C Standard-Include-Files, C File I/O, ...
2. Parallelität auf Shell-Ebene: `&`, `|`, `jobs`, ...
3. `system` und `popen`
4. Überblick: Prozesse
5. `fork`, `exec`, `wait` usw.
6. Signal Handling
7. Unnamed Pipes und Named Pipes
8. Shared Memory
9. Semaphore
10. SysV Message Queues
11. `mmap`
12. `select`, Linux `/proc`, ...

(1–4 Vorbereitung / 5–9 "Harter Kern" / 10–12 Aufputz)

Beispiele:

Pipes:
In Unix parallel (im Unterschied zu Windows)!
Beispiel: Gnu C Compiler intern: Präprozessor, Compiler, Assembler

Server:

- Samba (1 Prozeß pro PC)
- Datenbanken (1 Prozeß pro User + Systemprozesse)
- SAP, Linkworks (interne Struktur aufzeichnen: "Arbeitsverteiler" + "Arbeiter")
- WWW-Server: Traditionell 1 Prozeß pro `GET` (analog: FTP, ...)

High Performance Computing:

- Physikalische Simulation (Wetter, Aerodynamik, Crashtests, ...)

- Computeranimation
- Such- und Optimierungsprogramme (Schachspiel)
- Paralleles Make

Discrete Event Simulation:
Tankstelle, Lift, ...

Voraussetzungen:
C und Linux wird vorausgesetzt.

2 C Library

Ohne Anspruch auf Vollständigkeit!!!

Include-Files für `#include`:

- Wo wird gesucht: Unterschied `" "` und `< >`
- Bitte für alle verwendeten Library-Funktionen: Header inkludieren!
- Auf Typen achten: `off_t`, `size_t`, `mode_t`, `pid_t`, ...!
- Auch Konstanten sind hilfreich, z. B. `PATH_MAX`!

Als Suchhilfe:

`stdio.h`	C-Library-File-I/O, `perror`, `system`, `popen`
`string.h`	String- und Mem-Funktionen
`ctype.h`	`isalpha`, ...
`sys/types.h`	Typdeklarationen
`errno.h`	`errno`, alle Werte und Makros dafür (`EPERM`, ...)
`stdlib.h`	Kreuz und quer: Makros, Typen, Funktionen:
	`malloc` & `free`, `exit` & `abort`, `ato`x
	`rand`, `system`, `getenv`, ...
	Merkhilfe: `stdlib.h` enthält *betriebssystem-unabhängige* Funktionen
`unistd.h`	Viele Unix-Systemfunktionen: `exec` & `fork` & `getpid`,
	`getenv`, `getcwd` & `chdir`, `sleep`
	Unix-System-File-I/O (`read`, ...), `pipe`, ...
	Merkhilfe: `unistd.h` enthält *Unix-spezifische* Funktionen
`time.h`	Zeitfunktionen (auch: `sys/time.h`, `sys/times.h`)
`fcntl.h`	`open`, `creat`, `fcntl`
`sys/stat.h`	`stat`, `fstat`, Konstanten und Makros dafür
`dirent.h`	Lesen von Directories
`assert.h`	`assert`
`pwd.h`	Passwort-Handling
`limits.h`	Konstanten: Limits, Min- / Max-Werte (z. B. `PATH_MAX`)
`math.h`	`sin`, ...
`setjmp.h`	`setjmp` & `longjmp` (igitt!)
`stdarg.h`	`va`-Makros (igitt!)
`signal.h`	Funktionen für Signal-Handling, Konstanten für Signale
`sys/wait.h`	`wait` und Verwandte
`sys/ipc.h`	Allg. Definitionen für Shm/Sem/Msg
`sys/msg.h`	Message Queues
`sys/sem.h`	Semaphore
`sys/shm.h`	Shared Memory

Unix-System-File-I/O:

- Type `int` ("Filedescriptor")
- Standardfiles: `0`, `1`, `2` (oder besser lesbar und portabel: `STDIN_FILENO`, `STDOUT_FILENO` und `STDERR_FILENO` aus `unistd.h`)
- Funktionen ohne `f`

- Returnwert `-1` für Fehler
- Direkte Systemaufrufe (Kernel, nicht Library)

`open(`*name*`, `*flags [, perm]*`)`	Öffnet File (legt ihn ev. neu an) `O_RDONLY`, `O_WRONLY`, `O_RDWR`, `O_APPEND`, `O_TRUNC`, `O_CREAT`, `O_EXCL`, `O_NONBLOCK`
`creat(`*name*`, `*perm*`)`	`open(`*name*`, O_WRONLY | O_CREAT | O_TRUNC, `*perm*`)`
`dup(`*fd*`)` `dup2(`*fd1*`, `*fd2*`)`	Öffnet offenen File auf anderem Descriptor
`close(`*fd*`)`	Schließt File
`read(`*fd*`, `*buf*`, `*n*`)`	Liest
`write(`*fd*`, `*buf*`, `*n*`)`	Schreibt *n* Bytes, unformatiert ohne Rücksicht auf `\n` und `\0`
`lseek(`*fd*`, `*pos*`, `*mode*`)`	Setzt Schreib/Leseposition

C-Library-File-I/O: (aus `stdio.h`)

- Type `FILE *` (intern `typedef struct ...`) ("File-Pointer")
- Standardfiles: `stdin`, `stdout`, `stderr`
- Funktionen `f...`
- Returnwert `EOF` oder `NULL` für Fehler (`EOF` verwenden, nicht `-1`!!!)
- Achtung: ***Gepuffert***
  - `stderr` am Terminal: Unbuffered
  - `stdin` und `stdout` am Terminal: Line buffered (I/O erst bei `\n` oder bei Read!)
  - Alles andere: Block buffered (I/O erst wenn 4 KB voll)
- In Library programmiert, ruft intern Unix-System-File-I/O auf!
- Nicht beides gemischt auf einem File verwenden!!!

`fopen(`*name*`, `*type*`)`	Öffnet File (`r w a r+ w+ a+`)
`fdopen(`*fd*`, `*type*`)`	Öffnet File zu offenem Descriptor
`freopen(`*name*`, `*type*`, `*file*`)`	Öffnet File statt offenem File (für `stdin`, `stdout`, ...)
`fclose(`*file*`)`	Schließt File
`fflush(`*file*`)`	Schreibt Puffer
`getc(`*file*`)` `fgetc(`*file*`)`	1 Char lesen ***Achtung:*** Ergebnis ist `int` wegen `EOF`!!! `getc` ist Makro, `fgetc` ist Funktion
`getchar()`	`getc(stdin)`
`putc(`*char*`, `*file*`)` `fputc(`*char*`, `*file*`)`	1 Char schreiben `putc` ist Makro, `fputc` ist Funktion
`putchar(`*char*`)`	`putc(`*char*`, stdout)`
`fgets(`*str*`, `*len*`, `*file*`)`	String lesen (bis `\n`)
`gets(`*str*`)`	Dasselbe von `stdin` ***Achtung:*** Keine Längenprüfung!!! ***Achtung:*** `gets` verwirft das `\n`, `fgets` nicht!

`fputs(`*str*`,` *file*`)`	String schreiben
`puts(`*str*`)`	Dasselbe auf `stdout`
	Achtung: `puts` hängt `\n` an, `fputs` nicht!
`printf(`*format*`,` *arg* `...)`	Formatierte Ausgabe auf `stdout`
`fprintf(`*file*`,` *format*`,` *arg* `...)`	Formatierte Ausgabe auf *file*
`sprintf(`*str*`,` *format*`,` *arg* `...)`	Formatierte Ausgabe in *str*
	Achtung: Keine Längenprüfung! Keine Typprüfung!
	Keine Prüfung auf Anzahl der Argumente!
`scanf(`*format*`,` *ptr* `...)`	Formatiertes Lesen von `stdin`
`fscanf(`*file*`,` *format*`,` *ptr* `...)`	Formatiertes Lesen von *file*
`sscanf(`*str*`,` *format*`,` *ptr* `...)`	Formatiertes Lesen von *str*
	Achtung: Argumente müssen ***Pointer*** sein!
`fread(...)`	Blockweises unformatiertes Lesen
`fwrite(...)`	Blockweises unformatiertes Schreiben
`fseek(`*file*`,` *pos*`,` *mode*`)`	Aktuelle Position setzen
`rewind(`*file*`)`	Auf Fileanfang positionieren (Kein `f`!)
`ftell(`*file*`)`	Was ist die aktuelle Position?
	Alternativ: `fgetpos` und `fsetpos`
`ferror(`*file*`)`	Ist ein Fehler aufgetreten?
`feof(`*file*`)`	Ist End-of-File erreicht?
`clearerr(`*file*`)`	Setzt `ferror`- und `feof`-Flag zurück
`fileno(`*file*`)`	Zu *file* gehörender File-Descriptor

Andere File-Funktionen:

`remove(`*name*`)` `rmdir(`*name*`)`	Löschen von Files/Dir's
`mkdir(`*name*`,` *perm*`)`	Anlegen von Dirs
`rename(`*old*`,` *new*`)`	Umbenennen von Files/Dir's
`tmpfile()`	Kreiert und öffnet temporären File
`access(`*name*`,` *mode*`)`	Prüft Existenz und Zugriffsrechte
`stat(`*path*`,` *buf*`)`	Liest Status
	(Rechte, Länge, Owner, Filedatum, ...)
`fstat(`*fd*`,` *buf*`)`	Analog für *fd* (nicht *file*!)
`chmod(`*name*`,` *perm*`)`	Setzt Rechte

Andere Library-Funktionen:

String- und Mem-Funktionen:

`\0`-terminiert!

Achtung: Keine Längenprüfung!

- `strcat`, `strncat`, `strcpy`, `strncpy`: Strings anhängen / kopieren
- `strcmp`, `strncmp`: Strings nach ASCII-Wert vergleichen
- `strcoll`: Strings nach Locale vergleichen
- `strlen`: Länge eines Strings
- `strdup`: Kopie eines Strings via `malloc`
- `strchr`, `strrchr`, `strstr`: Suche in Strings (alt: `index`, `rindex`)
- `strspn`, `strcspn`, `strpbrk`, `strtok`, `strsep`: Parsen von Strings

- `atol`, `atof`, `strtol`, ...: Umwandlung von String in Zahl
- `memchr`, `memcmp`, `memcpy`, `memset`: Das gleiche für Speicherbereiche (Zähler statt `\0`)

Char-Funktionen:

- `isdigit`, `isalpha`, `islower`, `isupper`, `isalnum`, `isspace`, `iscntrl`, ...: Test von Char's (Locale!!!)
- `toupper`, `tolower`: Umwandlung groß/klein

Int-Funktionen:

- `rand`, `srand`: Zufallszahlen

Math-Funktionen:

- `sin`, `cos`, `tan`, ...
- `exp`, `log`, `log10`, `pow`, `sqrt`
- `fabs`, `floor`, `ceil`
- `drand48`, ...: Zufallszahlen

Error Handling: *Wir prüfen jeden nichttrivialen Library- und Kernel-Aufruf!*

- `extern int errno`: Fehlercode des letzten Library-Aufruf (oder 0, wenn alles ok) (aus `errno.h`) (*Achtung:* Wird automatisch gesetzt, aber nicht zurückgesetzt!)
- `EPERM`, `ENOENT`, ...: Konstanten dafür
- `perror(`*str*`)`: Gibt *str* und Fehlertext zu `errno` auf `stderr` aus. Empfohlen!!! (Einheitlichkeit, Lokalisierung, ...) Noch besser: `fprintf` auf `stderr` mit `argv[0]` und `strerror(errno)` (und ev. Name des Files o. ä.)! Übliches Format: *Programmname*: *Operation*: *strerror-text*
- `exit(`*rc*`)`: Ende des Programms mit angegebenem Returncode
- `atexit(`*func*`)`, `on_exit(`*func*`,` *arg*`)`: Installation von Exithandlern (Funktionen, die bei Programmende automatisch ausgeführt werden)
- `abort()`: Harter Programmabbruch mit `core`
- `assert(`*expr*`)`: Prüft *expr*, sonst Abbruch mit Zeilennummer (aus `assert.h`)
- `setjmp`, `longjmp`: (Finger weg!)

System-Funktionen:

- `malloc`, `free`: Dynamische Speicherverwaltung
- `getcwd`, `chdir`
- `getenv`: Lesen von Environment-Variablen
- `getopt`: Lesen von Commandline-Flags
- `sleep(`*n*`)`: Tut *n* Sekunden lang nichts (intern `alarm`)

Datum und Zeit:

- `time`: Liefert aktuelle Zeit (Sekunden seit 1970)
- `ctime`, `asctime`, `localtime`, `strftime`, ...: Zeit formatieren
- `mktime`: Die Gegenrichtung
- `clock_t`, `time_t`, `struct tm`: Typen dafür

3 `system` und `popen`

`int system(const char *`*command*`)`
Führt *command* als eigenen Prozeß aus

command wird von einer Shell ausgeführt (wie vom Terminal):
Pipes, Redirect, Wildcards, Backquotes, ... sind erlaubt
command kann auch ein selbstgeschriebenes Programm/Script sein
(Achtung auf `PATH`, damit es gefunden wird!)
Kehrt erst zurück, wenn *command* endet (daher nicht echt parallel!)
Return-Wert: Exit-Status von *command*
(codiert wie bei `wait`, siehe `man 2 wait`,
mit Makros `WIFEXITED`, `WEXITSTATUS` usw. aus `#include <sys/wait.h>` auswerten!)
bzw. ein negativer Wert, wenn der Shell-Aufruf schiefgeht (Grund in `errno`!)

Wenn *command* auf `&` endet, wird es als echt paralleler Prozeß ausgeführt: `system` kehrt sofort zurück, *command* läuft parallel weiter. Das ist nicht empfohlen: Man kann nicht herausfinden, wann und mit welchem Exit-Status *command* geendet hat oder ob es noch läuft, und man kann es auch nicht gezielt beenden.

command erbt unter anderem:

- Das Environment
- Offene Files (vor allem `stdin`, `stdout`, `stderr`)
- Current Directory
- Rechte
- Signal Settings "Ignore"
- `umask`

Achtung:

- Wenn *command* kein fixer String ist: Sicherheit! Gut prüfen!!! Beispiel Webserver!
  (Sonderzeichen entfernen, Filenamen quoten, ...)
- Ebenfalls aus Sicherheitsgründen empfohlen: Absoluter Pfad für *command*
- `system` verträgt sich schlecht mit `wait` und Handling von `SIGCHLD`
  (macht intern `fork` / `exec` / `wait`)!
- `system` und `popen` sind ineffizient (es wird eine `bash` gestartet!)!

Übliche Tricks:

- *command* zusammenbasteln (Filenamen, Mail-Adressen, ...)
- Vorher Environment bereinigen oder anreichern
- Vorher `stdin` / `stdout` / `stderr` umleiten (oder Redirect im *command* verwenden)
- `system` als "Wrapper" um Set-Userid-Shellscripts

`FILE *popen(const char *`*command*`, const char *`*rw*`)`

Startet *command* wie `system`, aber parallel.
Liefert einen offenen File (`FILE *`) als Ergebnis
(oder `NULL`, wenn was schiefgeht)

Wenn *rw* `"r"` ist:
File ist zum Lesen geöffnet

und mit `stdout` von *command* verbunden
(*command* ist vorderes Ende einer Pipe, eigenes Programm hinteres,
d. h. das eigene Programm liest den Output von *command*)
`stdin` von *command* ist mein `stdin`

Wenn *rw* `"w"` ist:
File ist zum Schreiben geöffnet
und mit `stdin` von *command* verbunden
(*command* ist hinteres Ende einer Pipe, eigenes Programm vorderes,
d. h. das eigene Programm schreibt in den Input von *command*)
`stdout` von *command* ist mein `stdout`

Schließen des Files: `int pclose(FILE *`*file*`)` (*nicht* `fclose`!)
Wartet, bis *command* endet!
Ergebnis: Exit-Status von *command* (wie oben) oder `-1`.

Achtung:

- Bei Mode `"w"`: Wenn notwendig `fflush` verwenden!
- Sonst siehe `system`!

Wichtigste Anwendung:

- Drucken aus Unix-Programmen (z. B. `netscape`, `acroread`, ...):
  `printfile=popen("lp ...", "w")`
  (was auf `printfile` geschrieben wird, landet am Drucker)
- Analog für Mail schicken aus einer Anwendung heraus,
  Anzeigen des Output mit `more`, ...

4 Prozesse, `fork`

Prozesse allgemein:

- Was ist ein Prozeß?
  - Prozeß = in Ausführung befindliches Programm
    (Code + Daten) (läuft oder wartet)
  - Prozeß = Verwaltungseinheit des Betriebssystems für CPU-Zuteilung, Speicherverwaltung, Rechte, offene Files, ...

  Daher:

  - Ein Programm wird erst mit dem Starten ein Prozeß (oder mehrere), ein Programmtext oder Executable File an sich ist noch kein Prozeß.
  - Jeder Prozeß befindet sich zu jedem Zeitpunkt an genau einer Codestelle genau eines Programmes, aber ein Programm kann zu einem Zeitpunkt von mehreren Prozessen ausgeführt werden (entweder, weil es mehrmals unabhängig voneinander gestartet wurde, oder weil es sich nach dem Start intern in mehrere gleichzeitige Abläufe aufgespalten hat). Jeder Prozeß hat dabei seine eigene aktuelle Codestelle im Programm, unabhängig von den anderen.

  - Prozesse können auf Mehrprozessor-Maschinen echt gleichzeitig (parallel) ausgeführt werden, auf Einprozessor-Maschinen werden sie durch das Betriebssystem gesteuert reihum stückerlweise nacheinander ausgeführt.

- Prozesse versus Threads:

  Prozesse sind die "Maximalvariante" paralleler Abläufe: Jeder Prozeß hat seinen eigenen Speicher, seine eigenen offenen Files, seine eigenen Rechte, sein eigenes Environment und Working Directory, seine eigenen Timer, usw.. Die Erzeugung eines neuen Prozesses und das Umschalten zwischen bestehenden Prozessen ist daher relativ langsam und aufwändig, und in vielen Fällen wäre soviel Aufwand gar nicht notwendig.

  Daher wurden in den letzten Jahren in vielen Betriebssystemen zusätzlich sogenannte "Threads" als "Minimalvariante" paralleler Abläufe eingeführt (im Englischen oft als "lightweight processes" bezeichnet): Ein Thread hat nur einen eigenen Programmzeiger (für die aktuelle Codestelle) und einen eigenen Stack (für Funktionsaufrufe), für alles andere wird der Vaterprozeß mitbenutzt: Alle Threads eines Prozesses greifen auf dieselben globalen Daten und dynamischen Speicherstrukturen zu, und sie nutzen dieselben offenen Files, dieselben Rechte und dasselbe Environment wie der Vater. Mit anderen Worten: Threads sind eigenständige, parallele Abläufe innerhalb ein und desselben Prozesses, aber sie sind keine vollwertige, selbstständige Verwaltungseinheit aus der Sicht des Betriebssystems, sondern Teil des Vaterprozesses: Ein Thread für sich allein hat weder eigene Rechte, noch eigenen Speicher, noch ein Environment, usw.. Dafür lassen sich Threads innerhalb eines Prozesses sehr effizient erzeugen und umschalten.

  Threads werden hier nicht weiter behandelt: Sie sind (auf der Ebene des Unix-Kernels und der C-Library) noch nicht hinreichend einheitlich implementiert und portabel verwendbar (bei Java gehören sie hingegen sehr wohl zum Standard).

- **Terminologie:**
  Im Deutschen: *Vaterprozeß*, *Sohnprozeß*
  Im Englischen: *parent process*, *child process*

- Starten von Prozessen und Programmen in Unix:
  In zwei Schritten: `fork` und `exec`

  `fork`**:** Erzeugt einen neuen Prozeß. Der neue Prozeß führt das *gleiche* Programm aus wie der Aufrufende.
  `exec`**:** Startet im aktuellen Prozeß ein neues Programm statt dem gerade laufenden Programm. Der Prozeß bleibt dabei *gleich* (gleiche Pid), es wird *kein* neuer Prozeß erzeugt.

- Prozeßkennung in Unix: **"Pid"** ("process identifier")
  Positive, ganze Zahl größer 1, Typ `pid_t` (`typedef short` oder `int`)
  (0 ist ein Kernel-Prozeß (`sched`, `idle` oder `swapper`), 1 ist `init`, negative Werte haben Sonderbedeutung).
  Wird vom Kernel beim Prozeß-Start zufällig zugeteilt.
  Bleibt für die Laufzeit des Prozesses gleich und eindeutig.

- Weitere Eigenschaften eines Prozesses: Parent Pid, Eigentümer, Session, Process Group, Controlling Terminal, ...

`pid_t fork(void)`**:** Klont den aktuellen Prozeß:
Erzeugt einen neuen Prozeß als exakte Kopie des aufrufenden Prozesses. Beide haben im Moment des `fork` die gleichen Daten und stehen an der gleichen Stelle im Programm (kehren

gerade aus dem `fork` zurück), arbeiten aber von diesem Moment an jeder für sich unabhängig voneinander weiter.

Kehrt *zweimal* zurück: Einmal im Vater, einmal im neu erzeugten Sohn. Returnwert:

- *Im Vater:* Pid des neuen Sohnprozesses, `-1` bei Fehler (siehe `man`-Page: Die wichtigsten sind Speicher voll, Prozeßlimit pro User / Prozeßlimit systemweit erreicht).
- *Im Sohn:* `0`

Typische Programmstruktur daher:

```
#include <unistd.h>
#include <sys/types.h>

pid_t pid;

pid = fork();
if (pid < 0) {
   /* fork schiefgegangen, kein Sohn-Prozess, Fehlerbehandlung, exit */
}
if (pid == 0) {
   /* Code fuer den Sohn-Prozess */
}
else {
   /* Code fuer den Vater-Prozess */
   /* pid == Prozessnummer des neuen Sohn-Prozesses */
}
```

Achtung: `pid` speichern, der Vaterprozeß hat keine Möglichkeit, nachträglich die Pid eines seiner Söhne herauszubekommen!

Kopiert bzw. vererbt werden:

- Der Programmcode (wird geshared, nicht kopiert).
- Stack, Daten und Heap: Der Sohn hat zum Zeitpunkt des `fork` die gleichen Variablenwerte und die gleichen dynamischen Datenstrukturen wie der Vater.
  (meist intern als COW (copy-on-write) implementiert)
  Achtung: Auch die Puffer der Stdio-Library werden vererbt (und dann doppelt geschrieben bzw. gelesen)!
- Alle offenen File-Descriptoren (und damit auch umgeleitete Standard-Files) und die aktuelle Position in offenen Files.
  Wenn beide Prozesse gleichzeitig in denselben ***vorher offenen*** File sequentiell schreiben, landen die Daten in zufälliger Reihenfolge nacheinander in der Datei oder am Terminal (sie überschreiben ihren Output nicht gegenseitig).
  Wenn beide Prozesse gleichzeitig sequentiell aus demselben ***vorher offenen*** File oder vom Terminal lesen, werden die vorhandenen Daten zufällig auf jeweils einen der Prozesse aufgeteilt (dieselben Daten werden nur einmal, nicht doppelt gelesen).
  Öffnen beide Prozesse denselben File ***nach*** dem `fork`, lesen beide unabhängig voneinander den kompletten File bzw. schreiben unabhängig voneinander dieselben Daten (der Kernel verwaltet dann ***zwei*** getrennte File-Positionen).
- Das Environment, das Current Working Directory und das Current Root Directory.
- Alle Rechte (Userid, Group-Id, ...), die `umask` und die `ulimit`-Werte.
- Die Signal-Einstellungen (Signal-Handler, Signal-Maske).
- Das Controlling Terminal.

- Alle Shared-Memory-Segmente, die gerade im Vaterprozeß attached sind.

In allen diesen Punkten gilt:
Änderungen in einem Prozeß nach dem `fork` *wirken sich nicht auf den anderen Prozeß aus!*

Verschieden sind:

- Returnwert von `fork`.
- Pid und PPid des Prozesses.
- Prozeß-Laufzeit-Uhr (beim Sohn mit 0 initialisiert).
- `alarm`-Timer (beim Sohn abgeschaltet).
- Blockierte Signale (nicht an den Sohn vererbt).
- Record Locking (nicht an den Sohn vererbt).

Alternativen zu `fork`, je nach Unix: `vfork` oder `clone`:
Kopieren weniger, sind daher schneller.

`pid_t getpid(void)`, `pid_t getppid(void)` :
Liefern als Ergebnis die eigene Pid bzw. die Pid des Vaterprozesses.
Keine Fehlerbedingungen.
Achtung: `getppid` liefert `1`, wenn der Vater nicht mehr lebt!
(siehe nächste Stunde)

5 Ende von Prozessen, `wait` und Verwandte

Wie endet ein Prozeß?

1a) Er erreicht das Ende von `main`, oder macht ein `return` aus `main`.

1b) Er ruft `exit` auf.

2a) Er wird durch ein Signal beendet.

2b) Er ruft `abort` auf, oder ein `assert` schlägt fehl.

Was ist ein Signal?

Ein (meist nicht vorhersagbares, von außen ausgelöstes) Ereignis, das den normalen Programmablauf (wo immer er gerade ist) unterbricht.

Wenn ein Signal eintrifft, gibt es mehrere Möglichkeiten:

- Das Signal bricht das Programm ab (oder — bei den Job-Control-Signalen — versetzt das Programm in den Status "stopped").
- Das Signal wird ignoriert, das Programm macht normal weiter.
- Das Signal wird vom Programm abgefangen und bewirkt die Ausführung einer bestimmten Funktion (eines *Signal-Handlers*) im Programm (in der nächsten Stunde behandelt).
- Das Signal ist gerade blockiert und wird erst später wirksam (hier gar nicht behandelt).

Die vollständige Liste der Signalnamen und -nummern steht in `sys/signal.h` (auf Linux: `asm/signal.h`, siehe auch `man 7 signal`). Die wichtigsten:

Name	*Nr*	*Default-Aktion*	*übliche Ursache*
SIGINT	2	Abbruch	Interrupt vom Terminal (`Ctrl-C`)
SIGQUIT	3	Abbruch, `core`	Quit vom Terminal (`Ctrl-\`)
SIGTERM	15	Abbruch	Programmabbruch (z. B. mit `kill`)
SIGKILL	9	Abbruch	Garantierter Programmabbruch (nicht abfangbar)
SIGHUP	1	Abbruch	Verbindung zum Terminal unterbrochen (bei Daemons meist zum Neu-Initialisieren)
SIGSEGV	11	Abbruch, `core`	Schutzverletzung ("Urwald-Pointer")
SIGABRT	6	Abbruch, `core`	`abort` und `assert`
SIGILL	4	Abbruch, `core`	Illegal Instruction (Programm "im Urwald")
SIGFPE	8	Abbruch, `core`	Division durch 0, Overflow, ...
SIGBUS	10	Abbruch, `core`	z. B. Unaligned Pointer
SIGSYS	12	Abbruch, `core`	Fehlerhafter Syscall
SIGPIPE	13	Abbruch	Schreiben in Pipe ohne Leser ("broken pipe")
SIGALRM	14	Abbruch	`alarm`-Timer abgelaufen
SIGCHLD	18	Ignoriert	Ende eines Sohn-Prozesses
SIGWINCH	20	Ignoriert	Terminal-Größe wurde geändert
SIGUSR1	16	Abbruch	Frei verwendbar
SIGUSR2	17	Abbruch	Frei verwendbar
SIGSTOP	24	Prozeßstop	Prozeßstop (nicht abfangbar)
SIGTSTP	25	Prozeßstop	Terminal-Prozeßstop (`Ctrl-Z`)
SIGTTIN	26	Prozeßstop	Terminal-Read im Hintergrund
SIGTTOU	27	Prozeßstop	Terminal-Write im Hintergrund
SIGCONT	23	Ignoriert	Prozeßfortsetzung nach Stop

(die Nummern sind nicht auf allen Unix-Systemen gleich!)

Die meisten Signale werden nur dem "betroffenen" Prozeß geschickt, aber *nicht* seinen Söhnen. Vom Terminal ausgelöste Signale (`SIGINT`, `SIGQUIT`, `SIGHUP`, `SIGTSTP`, `SIGWINCH`) werden *allen* Prozessen geschickt, die in diesem Terminal gerade im Vordergrund arbeiten.

Was passiert, wenn ein Prozeß endet?

Wenn der Vater noch lebt und schon wartet: Der beendete Prozeß verschwindet sofort aus der Prozeßliste, im Vater kehrt die Funktion `wait` mit dem Exitstatus des beendeten Prozesses zurück.

Wenn der Vater noch lebt und noch nicht wartet: Der beendete Prozeß wird ein Zombie-Prozeß (mit der Parent Pid seines Vaters).

- Wenn der Vater später einmal `wait` aufruft, verschwindet der Zombie aus der Prozeßliste, und das `wait` im Vater kehrt sofort mit dem Exitstatus des Zombies zurück (wie im ersten Fall).
- Wenn der Vater endet, ohne auf seinen Sohn zu warten, erbt der `init`-Prozeß den Zombie: Die Parent Pid des Zombies wird auf 1 gesetzt, der Zombie wird irgendwann vom `init`-Prozeß beseitigt (wie im nächsten Fall).

Wenn der Vater nicht mehr lebt: Der beendete Prozeß wird ein Zombie mit Parent Pid 1. Der `init`-Prozeß (Prozeß 1) sollte alle Zombies mit Parent Pid 1 in regelmäßigen Abständen aufräumen, d. h. diese sollten von selbst (längstens nach ein paar Minuten) aus der Prozeßliste verschwinden (wenn ein Zombie mit Parent Pid 1 stundenlang überbleibt, stimmt irgendetwas mit dem `init`-Prozeß nicht).

Außerdem bekommt der Vater beim Ende eines Sohnes das Signal `SIGCHLD`, das allerdings per Default ignoriert wird.

Was passiert, wenn der Vaterprozeß eines Prozesses endet?

Die Parent Pid des Sohnes wird auf 1 gesetzt (egal, ob der Sohn noch lebt oder schon Zombie ist). Sonst bekommt der Sohn vom Tod des Vaters nichts mit.

Was ist ein Zombie-Prozeß (auch: Defunct-Prozeß)?

Ein Zombie-Prozeß ist ein Prozeß, der geendet hat, ohne daß sein Vater bisher für ihn `wait` oder `waitpid` aufgerufen hat. Mit anderen Worten: Zwischen dem Zeitpunkt, wo ein Prozeß endet, und dem Zeitpunkt, wo sich jemand seinen Exitstatus abholt (und der Prozeß damit endgültig und restlos verschwindet), existiert der Prozeß als Zombie.

Ein Zombie-Prozeß führt kein Programm mehr aus; als Programm wird im `ps` bei den meisten Unixes `<defunct>` angezeigt, als Ausführungsstatus `Z`. Er kann auch nicht mehr gekillt werden (ist schon tot!). Ein Zombie-Prozeß belegt keinerlei Ressoucen außer seinem Eintrag in der Prozeßliste. Dieser Eintrag ist dazu da, Pid und Exitstatus des Prozesses zu speichern, bis der Vater ihn abfragt, und bis dahin zu verhindern, daß die gleiche Pid nocheinmal vergeben wird.

Bei Programmen, die lange laufen und viele Sohnprozesse erzeugen, muß man darauf achten, daß diese nicht alle zeitlebens als Zombies überbleiben, weil sonst rasch die Prozeßzahl-Limits erreicht werden.

Die Funktion `wait`: `pid_t wait(int *`*statptr*`)`

Liefert den Pid (als Returnwert) und den Exitstatus (in der `int`-Variable, deren Adresse als *statptr* übergeben wurde; darf `NULL` sein) eines beendeten Sohnprozesses zurück. Wenn man am Exitstatus nicht interessiert ist, kann man `NULL` als *statptr* übergeben.

Mögliche Fälle:

1. Der Prozeß hat Söhne, und mindestens einer davon hat schon geendet (d. h. ist ein Zombie): `wait` kehrt sofort mit Pid und Exitstatus eines der Zombies zurück.
2. Der Prozeß hat Söhne, aber diese sind alle noch aktiv: `wait` wartet, bis einer der Söhne endet, und kehrt dann mit dessen Pid und Exitstatus zurück.
3. Der Prozeß hat gar keine Söhne, oder das Warten wurde durch ein Signal unterbrochen (bei `waitpid` auch: Der angegebene Prozeß existiert nicht oder ist kein Sohn). In diesem Fall wird `-1` als Pid zurückgegeben und `errno` gesetzt (`ECHILD` oder `EINTR`).

Die Funktion `waitpid`: `pid_t waitpid(pid_t` *pid*`, int *`*statptr*`, int` *options*`)`

Wie `wait`, aber

- In *pid* kann man die Pid eines bestimmten Prozesses mitgeben, auf den gewartet werden soll. `-1` hat den gleichen Effekt wie ein normales `wait` (wartet auf den nächstbesten Sohn), die anderen Sonderfälle (`0`, `<-1`) behandeln wir nicht.
- In *options* kann man Optionen angeben (sonst `0` übergeben). Für uns ist nur `WNOHANG` wichtig: In diesem Fall wartet `waitpid` nicht, wenn es keinen beendeten Prozeß gibt, sondern kehrt sofort mit `0` als Ergebnis zurück.

Achtung: Man kann von jedem Sohnprozeß nur ***einmal*** den Exitstatus mit `wait` oder `waitpid` abrufen. Dadurch wird der Eintrag in der Prozeßtabelle gelöscht, danach gibt es im System keinerlei Informationen mehr über den Prozeß, seine Pid ist wieder frei für neue Prozesse!

Der Exitstatus: Der von `wait` und `waitpid` gelieferte Wert ist codiert (so wie der von `system` und `pclose`) und muß mit den Makros aus `sys/wait.h` getestet und entschlüsselt werden. Es ist immer eines der folgenden drei Makros true:

`WIFEXITED(`*status*`):` Der Prozeß endete normal (durch Return aus `main` oder durch den Aufruf von `exit(`*n*`)`). In diesem Fall liefert `WEXITSTATUS(`*status*`)` den Returncode *n*.

`WIFSIGNALED(`*status*`):` Der Prozeß endete durch ein Signal (oder durch Aufruf von `abort`). In diesem Fall liefert `WTERMSIG(`*status*`)` die Nummer des Signals, das den Prozeß gekillt hat.

`WIFSTOPPED(`*status*`):` Der Prozeß endete noch gar nicht, sondern wurde nur gestoppt (diesen Sonderfall behandeln wir nicht). In diesem Fall liefert `WSTOPSIG(`*status*`)` die Nummer des Signals, das den Prozeß gestoppt hat.

Typische Programmstruktur daher:

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t pid;
int exitstat;

pid=wait(&exitstat);
if (pid>0) { /* wait erfolgreich */
  /* exitstat enthaelt Status von geendetem Sohnprozess pid */
  if (WIFEXITED(exitstat)) { /* Sohn endete normal */
    rc=WEXITSTATUS(exitstat);
  }
  else if (WIFSIGNALED(exitstat)) { /* Sohn starb durch ein Signal */
    sig=WTERMSIG(exitstat);
  }
  else { /* WIFSTOPPED, sollte bei uns nicht vorkommen */
  }
}
else if (pid==0) { /* nicht bei wait, nur bei waitpid mit WNOHANG: */
  /* Gerade kein beendeter Sohn vorhanden */
}
else { /* pid==-1, Fehler im wait */
}
```

Weitere Funktionen: Je nach Unix gibt es auch noch die Funktionen `wait2`, `wait3` und `wait4` mit zusätzlichen Optionen und Informationen (siehe `man`-Pages).

6 sigaction, kill, alarm, pause

Signal-Handler: Ein ***Signal-Handler*** ist eine Funktion, die dann ausgeführt wird, wenn ein bestimmtes Signal eintrifft, unabhängig davon, wo das Programm gerade ist. Wenn der Signal-Handler das Programm nicht z. B. mit `exit` beendet oder mit `longjmp` an einer bestimmten Stelle fortsetzt, macht es nach Ende des Signal-Handlers dort weiter, wo es vorher war (Signal-Handler sind die wichtigste und die einzig "moralisch vertretbare" Anwendung von `longjmp`).

Achtung: Wenn das Programm beim Eintreffen eines Signals gerade in bestimmten Systemaufrufen hing (wichtigste Beispiele: `read`, `write`, `wait`), so setzt es nach der Behandlung des Signals ***nicht*** den Systemaufruf fort. Stattdessen kehrt der Systemaufruf sofort mit einem Fehler (`-1`) und `errno` gleich `EINTR` zurück[1]! (sinnvollerweise nochmal aufrufen)

Ein Signal-Handler wird als normale Funktion mit einem `int`-Argument und Returntyp `void` definiert, beim Aufruf wird automatisch die Nummer des auslösenden Signals als Argument übergeben (so kann man denselben Signal-Handler für mehrere Signale einrichten und trotzdem unterscheiden, welches Signal gekommen ist).

Typische Fälle für Signal-Handler sind:

- Aufräum-Funktionen, die noch temporäre Dateien wegräumen, schöne Fehlerberichte schreiben, Daten aus dem Hauptspeicher in eine Datei retten, Sohn-Prozesse beenden, ein `ROLLBACK` auf die Datenbank absetzen, usw., bevor das Programm wirklich abbricht[2].
- Handler für `SIGCHLD`, die nur dann ein `wait` aufrufen, wenn wirklich ein Prozeß geendet hat, damit der Vater nicht ständig in einem `wait` hängen muß.
- Bei System-Programmen (`init`, `inetd`, `cron`, ...) ein Handler für `SIGHUP`, der u. a. bewirkt, daß die Konfigurationsdateien frisch gelesen und die Logfiles frisch geschrieben werden.
- Ein Handler für `SIGALRM`, um Timeouts zu implementieren oder die Dauer irgendwelcher Operationen zu limitieren.
- Beliebige Synchronisation und Kommunikation zwischen mehreren Prozessen eines Programmes mit den frei verwendbaren Signalen `SIGUSR1` und `SIGUSR2`.

Achtung:
Variablen, die sowohl im Signal-Handler als auch im Rest des Programmes verwendet werden, sind *global* zu definieren. Variablen, die nur im Signal-Handler verwendet werden, aber zwischen den Aufrufen ihren Wert behalten sollen, müssen `static` sein!

Die Funktion `sigaction`:

```
int sigaction(int sig,
    const struct sigaction *new_sighandler,
    struct sigaction *old_sighandler)

struct sigaction {
  void (*sa_handler)(int);
  sigset_t sa_mask;
  int sa_flags;
};
```

(und ev. noch weitere Felder)

Die Funktion `sigaction` sagt dem System, daß es die Funktion *new_sighandler*`.sa_handler` jetzt für diesen Prozeß als Signal-Handler für das Signal *sig* eintragen soll (ist *new_sighandler*

[1] Das Verhalten ist nicht bei allen Unix-Varianten gleich, bei manchen tritt dieser Fall gar nicht ein, bei manchen wird er in der C-Library abgefangen und schlägt nicht bis zum Anwendungsprogramm durch.

[2] Ein Trick, damit das Programm trotz Signal-Handler mit einem Exit-Status endet, der einen Programmabbruch durch ein Signal anzeigt: Am Ende des Signal-Handlers das Handling des Signals auf Default zurücksetzen und ein `kill` auf sich selbst statt einem `exit` machen!

NULL, bleibt die Einstellung für das Signal *sig* unverändert). In *old_sighandler* wird (falls nicht NULL) die alte Einstellung für dieses Signal zurückgegeben.

sigaction liefert als Ergebnis 0 bei Erfolg und -1 bei Fehler.

Sonderfälle für sa_handler (vordefinierte Konstanten aus <signal.h>):

- SIG_IGN: Ignoriere das Signal *sig*.
- SIG_DFL: Richte für das Signal *sig* wieder die Default-Aktion ein (für die meisten Signale: Programm-Abbruch).

sig kann nicht SIGKILL oder SIGSTOP sein, für diese beiden Signale kann man keinen Signal-Handler einrichten.

Ein mit sigaction eingerichteter Signal-Handler bleibt eingerichtet, bis man ihn mit einem neuerlichen Aufruf von sigaction ändert.

Wenn ein Signal die Ausführung eines Signal-Handlers auslöst, ist dieses Signal automatisch blockiert, bis der Signal-Handler fertig ist. Es ist also sichergestellt, daß der Signal-Handler nicht durch das gleiche Signal während der Ausführung geschachtelt aufgerufen wird.

Im Feld sa_mask kann man eine Menge von Signalen angeben, die während der Ausführung des Signal-Handlers zusätzlich zu blockieren sind.

Wir initialisieren dieses Feld mit sigemptyset(&(*new_sighandler*.sa_mask)) auf die leere Menge.

Im Feld sa_flags können diverse Optionen für das Handling dieses Signals gesetzt werden (siehe man-Page von sigaction); wir setzen es normalerweise auf 0. Interessant (aber leider nicht auf allen Unix-Systemen implementiert) ist das Flag SA_RESTART, das bewirkt, daß durch das betreffende Signal unterbrochene Systemaufrufe nicht mit dem Fehler EINTR enden, sondern nach dem Signal-Handler fortgesetzt werden.

Weiters kann man in sa_flags eine andere Behandlung des Endens von Sohn-Prozessen definieren, die keine Zombies mehr erzeugt (aber auch kein Abfragen des Exit-Status mit wait mehr erlaubt).

Bei einem fork werden die mit sigaction eingerichteten Signal-Handler vererbt, bei einem exec bleiben die SIG_IGN-Einstellungen erhalten, alle anderen eingerichteten Signal-Handler werden wieder durch die Default-Aktion ersetzt.

Anmerkung:
Die "klassische" Funktion zum Einrichten von Signal-Handlern ist signal, sie hat aber einige Fehler:

- signal kennt keine Maske und keine Flags. Es schützt nicht davor, daß die Ausführung des Signal-Handlers vom gleichen oder anderen Signalen unterbrochen wird.
- signal setzt den Signal-Handler in jedem Fall, man kann ihn nicht nur abfragen.
- Ein mit signal gesetzter Signal-Handler ist "one-Shot", er wird beim Eintreffen des Signals auf die Default-Aktion zurückgesetzt. Kommt das Signal gleich nocheinmal, bevor man den Signal-Handler wieder neu gesetzt hat, wird das Programm abgebrochen.

Es entwickelten sich daher in den einzelnen Unix-Varianten mehrere verschiedene "verbesserte" Versionen von signal; schließlich wurde versucht, sigaction zu standardisieren (wobei sigaction durch das Angeben einiger Flags signal emulieren kann). Es gibt aber noch andere Funktionen für den gleichen Zweck (z. B. sigvec) und viele hier nicht besprochenen Erweiterungen und zusätzliche Funktionen. Unter Linux bietet der Header-File signal.h einen Überblick, was es im Bereich des Signal-Handlings an Funktionalität gibt.

Die Funktion `kill`: `int kill(pid_t` *pid*`, int` *sig*`)`

Schickt dem Prozeß *pid* das Signal *sig*.

Man darf allen eigenen Prozessen (gleicher User wie der aufrufende Prozeß) ein Signal schicken, egal, ob sie meine Söhne sind oder nicht.

Returnwert: `0` bei Erfolg, `-1` bei Fehler. Typische Fehler (in `errno`):

- `ESRCH`: Es gibt gar keinen Prozeß *pid*.
- `EPERM`: Den Prozeß *pid* gibt es, aber er gehört nicht mir.

Sonderfall: Ist *sig* gleich `0`, so wird nur geprüft, ob es den Prozeß *pid* überhaupt gibt und ob man ihm ein Signal schicken dürfte, ohne daß dem Prozeß tatsächlich irgendetwas geschieht.

Sonderfall: Ist *pid* gleich `0`, so wird das Signal allen Prozessen der aktuellen Prozeßgruppe geschickt (typischerweise dem von der Shell aus gestarteten Prozeß mit dem Hauptprogramm und allen seinen Söhnen, Enkeln, usw.). Ist *pid* gleich `-1`, so wird das Signal allen Prozessen des aktuellen Users geschickt. Andere negative Werte für *pid* behandeln wir hier nicht.

Die Funktion `alarm`: `unsigned int alarm(unsigned int` *sec*`)`

Setzt den Alarm-Timer des aufrufenden Prozesses auf *sec*: Nach *sec* Sekunden bekommt der aufrufende Prozeß des Signal `SIGALRM`.

Der Alarm-Timer merkt sich immer nur den letzten Aufruf von `alarm`, nicht mehrere. Ist *sec* gleich `0`, wird ein eventuell gesetzter Alarm-Timer wieder abgeschaltet.

War der Alarm-Timer vor dem Aufruf schon gesetzt, liefert `alarm` als Returnwert die Anzahl der Sekunden, die es noch bis zum ursprünglich gesetzten Alarm gedauert hätte, sonst liefert es `0`. Keine Fehlerfälle.

`sleep` verwendet intern `alarm`; man sollte in einem Programm immer nur einen der beiden Mechanismen nutzen, da sie sich gegenseitig stören.

Es gibt neben dem Alarm-Timer noch weitere Timer, auch mit Milli- oder Mikrosekunden-Auflösung, und auch solche, die die verbrauchte CPU-Zeit und nicht die Echtzeit messen. Dementsprechend gibt es auch Funktionen ähnlich `sleep` mit Mikro- oder Nanosekunden-Auflösung.

Die Funktion `pause`: `int pause(void)`

Blockiert den Prozeß, bis ein Signal eintrifft: Der Aufruf von `pause` kehrt erst dann zurück, wenn ein Signal gekommen ist und der dazugehörige Signal-Handler fertig ausgeführt wurde. Beendet das Signal den Prozeß, kehrt `pause` gar nicht zurück. Der Returnwert ist immer `-1`, `errno` ist `EINTR`.

`alarm` und `pause` kommen aus `unistd.h`, die restlichen Signal-Funktionen aus `signal.h`.

Signalmengen:

Eine Signalmenge ist eine Menge von Null oder mehr Signalen, realisiert üblicherweise als Bitmaske. Signalmengen werden als Signalmasken zum Blockieren von Signalen verwendet: Jeder Prozeß hat eine Signalmaske, die dort enthaltenen Signale sind blockiert. Kommt ein solches Signal, wird es nicht sofort wirksam, sondern erst dann, wenn die Blockierung wieder aufgehoben wird (d. h. eine Signalmaske ohne dieses Signal gesetzt wird)[3].

Signalmengen haben den vordefinierten Typ `sigset_t` aus `signal.h`. Folgende Funktionen manipulieren Signalmengen:

[3] Ein ignoriertes Signal ist hingegen verloren, es wird nie mehr wirksam.

`int sigemptyset(sigset_t *set)` initialisiert *set* auf die leere Signalmenge.
`int sigfillset(sigset_t *set)` initialisiert *set* auf die Signalmenge mit allen Signalen.
`int sigaddset(sigset_t *set, int signr)` fügt das Signal *signr* zur Signalmenge *set* dazu.
`int sigdelset(sigset_t *set, int signr)` entfernt das Signal *signr* aus der Signalmenge *set*.
`int sigismember(const sigset_t *set, int signr)` prüft, ob das Signal *sig* in der Signalmenge *set* enthalten ist.

`sigismember` liefert ein boolsches Ergebnis, die anderen Funktionen liefern `0` bei Erfolg und `-1` bei Fehler.

Folgende Funktionen stehen im Zusammenhang mit der Signalmaske eines Prozesses (Details siehe `man`-Pages):

`sigaction`: In `sa_mask` von `struct sigaction` wird die während der Ausführung des Signal-Handlers zu setzende Signalmaske angegeben.
`sigprocmask`: Liest, setzt, oder ändert die Signalmaske.
`sigpending`: Fragt ab, welche Signale gerade hängen (d. h. gekommen sind, aber blockiert sind und daher warten).
`sigsuspend`: Wie `pause`, aber die während des Wartens zu verwendende Signalmaske kann explizit angegeben werden (d. h. man kann auf bestimmte Signale warten).
`sigsetjmp` und `siglongjmp`: Wie `setjmp` und `longjmp`. Zusätzlich merkt sich `sigsetjmp` die gerade aktuelle Signalmaske, und `siglongjmp` stellt diese wieder her (sonst bliebe das Signal ewig blockiert, wenn man aus einem Signal-Handler herausspringt!).

Beispiel:

```
/* Zaehle endlos,
 * ignoriere Ctrl-C usw. mit einer freundlichen Meldung
 * und beginne bei SIGHUP von vorne
 */

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

static sigjmp_buf jmp;

void huphandler(int sig)
{
   siglongjmp(jmp, 1);
}

void termhandler(int sig)
{
   const char *s;

   if (sig==SIGINT) s="Ctrl-C";
   else if (sig==SIGQUIT) s="Ctrl-\\";
   else if (sig==SIGTERM) s="kill";
   else s="???";
   printf("Da wollte mich jemand mit %s umbringen!\n", s);
}

int main(void)
```

```
{
   int i;
   struct sigaction action;

   action.sa_handler=termhandler;
   sigemptyset(&(action.sa_mask));
   action.sa_flags=0;

   sigaction(SIGINT, &action, NULL);
   sigaction(SIGQUIT, &action, NULL);
   sigaction(SIGTERM, &action, NULL);

   if (sigsetjmp(jmp, 1)==0) puts("Auf geht's!"); /* erster Aufruf */
   else puts("Auf ein Neues!");    /* longjmp von huphandler */
   action.sa_handler=huphandler;
   sigaction(SIGHUP, &action, NULL);

   for (i=1; ; ++i) {
      printf("%u\n", i);
      sleep(1);
   }
}
```

7 Unnamed Pipes: `pipe`

Was ist eine Pipe?

Eine Pipe ist eine Einbahn-Datenverbindung zwischen zwei Prozessen: Einer schreibt darauf wie auf einen File, der andere liest daraus wie aus einem File.

Unter Unix laufen Pipes nicht über die Platte: Die Daten werden direkt im Kernel vom schreibenden zum lesenden Prozeß durchgereicht (relativ effizient!).

Eine Pipe hat im Kernel einen Puffer fixer Größe (typischerweise einige KB), Schreiber und Leser werden daher normalerweise abwechselnd ausgeführt: Wenn der Leser nicht liest, läuft der Puffer voll, und weitere Schreib-Aufrufe hängen dann so lange (und blockieren damit den schreibenden Prozeß), bis wieder Daten gelesen wurden.

Bei ringförmigen Konstruktionen aus mehreren Prozessen und Pipes kann es zu Deadlocks kommen (wenn z. B. alle lesen wollen und keiner schreibt oder umgekehrt).

Wie bei einer Datei können auch mehrere Prozesse in ein und dieselbe Pipe schreiben (die Daten werden hintereinandergestückelt) oder aus ein und derselben Pipe lesen (die Daten werden auf die Prozesse aufgeteilt).

Die Funktion `pipe`: `int pipe(int` *fd*`[2])` (aus `unistd.h`)

Die Funktion `pipe` erzeugt eine neue Pipe und liefert in *fd* zwei neu geöffnete Filedeskriptoren für die beiden Enden: *fd* ist ein Array von 2 Filedeskriptoren (`int`, nicht `FILE *`!), *fd*`[0]` dient zum Lesen aus der Pipe, *fd*`[1]` zum Schreiben in die Pipe (Merkregel: `0` wie `stdin` zum lesen, `1` wie `stdout` zum Schreiben). Als Returnwert liefert `pipe` bei Erfolg `0`, bei Fehler `-1`.

Ein Aufruf von `pipe` allein ist nutzlos, da man Pipes nicht innerhalb eines einzigen Prozesses verwenden kann. Unmittelbar nach dem `pipe` werden daher typischerweise ein oder mehrere `fork` gemacht. Jeder benutzt dann einen der beiden Filedeskriptoren und macht den anderen sofort zu.

Beispiele:

1. Sohn schreibt, Vater liest:

```
#include <unistd.h>
#include <sys/types.h>

pid_t pid;
int fd[2];

if (pipe(fd)<0) ...
if ((pid=fork())<0) ...
else if (pid==0) {  /* Sohn */
  close(fd[0]);
  ... write(fd[1], ...) ...
}
else {              /* Vater */
  close(fd[1]);
  ... read(fd[0], ...) ...
}
```

2. Vater schreibt, Sohn liest:

```
#include <unistd.h>
#include <sys/types.h>

pid_t pid;
int fd[2];

if (pipe(fd)<0) ...
if ((pid=fork())<0) ...
else if (pid==0) {  /* Sohn */
  close(fd[1]);
  ... read(fd[0], ...) ...
}
else {              /* Vater */
  close(fd[0]);
  ... write(fd[1], ...) ...
}
```

3. Verbindung zwischen 2 Söhnen:

```
#include <unistd.h>
#include <sys/types.h>

pid_t pid1, pid2;
int fd[2];

if (pipe(fd)<0) ...
if ((pid1=fork())<0) ...
else if (pid1==0) {  /* schreibender Sohn */
  close(fd[0]);
  ... write(fd[1], ...) ...
}
else {              /* Vater */
  close(fd[1]);
  if ((pid2=fork())<0) ...
  else if (pid2==0) {  /* lesender Sohn */
    ... read(fd[0], ...) ...
  }
  else {            /* Vater */
```

```
    close(fd[0]);
    ...
}
```

Was passiert wenn ...

- ... der (bzw. der letzte) schreibende Prozeß sein Ende der Pipe schließt oder endet? Der lesende Prozeß bekommt `EOF`, nachdem er alle noch in der Pipe befindlichen Daten gelesen hat (solange noch ein Schreiber die Pipe offen hat, bekommt der Leser *kein* `EOF`, auch wenn gerade keine Daten in der Pipe sind: Der Lese-Aufruf wartet, bis wieder Daten geschrieben werden oder die Pipe geschlossen wird!).
- ... der (bzw. der letzte) lesende Prozeß sein Ende der Pipe schließt oder endet, wenn es noch einen Schreiber gibt? Unmittelbar passiert nichts, aber der schreibende Prozeß bekommt beim nächsten Schreib-Versuch das Signal `SIGPIPE`, das den Prozeß per Default beendet[4]. Wird es ignoriert oder abgefangen, kehrt der Schreib-Aufruf mit Fehler und `EPIPE` in `errno` zurück.

Folglich schließt üblicherweise zuerst der Schreibende sein Ende, und der Lesende liest, bis er `EOF` bekommt. Deshalb ist es auch wichtig, daß jeder *sofort* das *nicht* benutzte Ende schließt: Vergißt der Leser, sein Schreibende zu schließen, hängt das Programm ewig, weil der Leser nie `EOF` bekommt: Auch nachdem der Schreiber die Pipe am Ende ordnungsgemäß geschlossen hat, hat noch jemand die Pipe zum Schreiben offen, nämlich er selbst ...

Man braucht eine Pipe nicht explizit wegräumen: Nachdem der letzte die Pipe geschlossen hat, werden alle damit verbundenen Ressourcen im Kernel automatisch freigegeben.

Was macht man mit Filedeskriptoren?

Auf Filedeskriptoren kann man nur mit `read` und `write` arbeiten (beide verarbeiten eine fixe Anzahl von Bytes ohne Formatierung). Wenn das zu unbequem ist, gibt es 3 Auswege:

1. Man formatiert mit `sprintf` in einen String, ermittelt dessen Länge, und gibt ihn dann als Ganzes mit `write` aus (ähnlich für `read` und `sscanf`, aber man kann mit `read` nicht zeilenweise lesen!).
2. Man macht sich mit `fdopen`[5] einen neuen File-Pointer auf, der auf denselben File wie einer der beiden Filedeskriptoren und damit auf das entsprechende Ende der Pipe verweist.

   Anschließend verwendet man die C-Library-I/O-Funktionen auf diesen File-Pointer anstatt `read` und `write` auf den Filedeskriptor.
3. Man legt mit `dup2`[6] die Standardein- oder Ausgabe eines Prozesses auf einen der beiden von `pipe` gelieferten Filedeskriptoren um. Danach kann man den ursprünglichen Fieldeskriptor schließen und normal über `stdin` oder `stdout` (z. B. mit `printf` oder `scanf`) auf der Pipe arbeiten.

---

[4] Passiert das dem Hauptprozeß, meldet die Shell `Broken pipe`.

[5] `fdopen` liefert zu einem gegebenen, bereits geöffneten Filedeskriptor einen neuen File-Pointer, der auf denselben File zeigt.

[6] `dup2` schließt den zweiten angegebenen Deskriptor, falls er offen ist, und öffnet ihn dann auf denselben File, auf den der erste angegebene Filedeskriptor zeigt. Für die Filedeskriptoren `0`, `1` und `2` der Standard-Files sind die Konstanten `STDIN_FILENO`, `STDOUT_FILENO` und `STDERR_FILENO` vordefiniert.

Das ist besonders praktisch, wenn man dann mit `exec` oder `system` ein Programm ausführt, das seinen Input auf `stdin` erwartet oder seinen Output auf `stdout` abliefert, und dieses mit der Pipe verbinden möchte.

Die Shell beispielsweise implementiert *cmd1* `|` *cmd2*, indem sie `pipe` aufruft und zwei Mal `dup2` benutzt, um `stdout` von *cmd1* und `stdin` von *cmd2* auf die beiden von `pipe` gelieferten Filedeskriptoren umzubiegen. Auch `popen` macht intern nichts anderes als `pipe`, `fork`, `dup2` und `exec`.

Achtung:

- `dup2` sollte man nur machen, wenn `stdin` bzw. `stdout` im Programm bisher noch nicht benutzt wurde!
- Wenn das aufrufende Programm ev. `stdin` oder `stdout` gar nicht offen hat, muß man zuerst prüfen, ob `pipe` nicht ohnehin schon `0` oder `1` als Filedeskriptor geliefert hat (dann geht die oben beschriebene Methode nämlich schief: Das `dup2` tut nichts, weil beide Argumente ein und derselbe Filedeskriptor sind, und das `close` schließt diesen einen und einzigen Filedeskriptor auf dem betreffenden Ende der Pipe!).

Beispiele zu `pipe` mit `fdopen` oder `dup2`: Siehe Beispiele zu `exec`!

Achtung:
Beim Zugriff auf Pipes über File-Pointer (via `fdopen` oder via `stdin`/`stdout` nach `dup2`) gilt: Eine Pipe ist kein Terminal. C-Library-I/O arbeitet daher auf Pipes grundsätzlich block-buffered (üblicherweise in 4 KB großen Brocken), nicht line-buffered, auch wenn es sich um `stdout` handelt. Wenn man will, daß Output wirklich sofort nach jeder Zeile in die Pipe geschrieben wird, ist daher jedesmal ein `fflush` notwendig, oder man setzt die Pufferung mit `setvbuf` um (Deteils siehe `man`-Page):

- `int setvbuf(FILE *`*f*`, char *`*buf*`, int` *bufmode*`, size_t` *bufsize*`)`
- Returnwert: `0` bei Erfolg, ungleich `0` bei Fehler.
- Aufruf, um File `f` auf line-buffered umzuschalten:
  `setvbuf(f, NULL, _IOLBF, 0)`
- Aufruf, um File `f` auf unbuffered umzuschalten:
  `setvbuf(f, NULL, _IONBF, 0)`
- `setvbuf` muß gleich nach dem Öffnen des Files und vor dem ersten I/O darauf aufgerufen werden!

8 Starten von Programmen: `exec`

Wirkung:

Das gerade laufende Programm samt allen seinen Daten wird durch das angegebene Programm ersetzt. Die Ausführung des neuen Programms beginnt mit `main`. Ein `exec`-Aufruf kehrt nie mehr zum aufrufenden Programm zurück (außer wenn das neue Programm nicht gestartet werden kann).

Der Prozeß (samt Pid) bleibt bei `exec` der gleiche, es wird *kein* neuer Prozeß gestartet.

`exec` startet das angegebene Programm direkt (ohne Zuhilfenahme einer Shell) und ist daher viel schneller als `system` oder `popen`. Damit stehen bei `exec` aber auch ***keine*** Shell-Features zur Verfügung (keine Wildcards, keine Redirection, keine Pipes, keine Backquotes, ...), und

die Argumente müssen ebenfalls fertig aufgeteilt einzeln übergeben werden. Außerdem muß das zu startende Programm ein Binärprogramm sein, kein Shell-Script[7].

Aufruf: Es gibt 6 `exec`-Funktionen, die sich in ihren Argumenten unterscheiden:

```
int execl(const char *command, const char *arg0, const char *arg1, ..., NULL)
int execv(const char *command, const char *argv[])
int execle(const char *command, const char *arg0, const char *arg1, ..., NULL,
const char *envp[])
int execve(const char *command, const char *argv[], const char *envp[])
int execlp(const char *command, const char *arg0, const char *arg1, ..., NULL)
int execvp(const char *command, const char *argv[])
```

Zum Merken:

`l` **wie Liste:** Die Argumente des neuen Programms werden als Liste von einzelnen Werten (Strings) angegeben, abgeschlossen mit einem Argument `NULL`.

`v` **wie Vektor:** Die Argumente werden als ein einziger Vektor (wie `argv`: Array of Stringpointers, letztes Element `NULL`) angegeben.

`e` **wie Environment:** Das Environment für das neue Programm wird explizit als letztes Argument angegeben (ähnlich `argv`: Array of Stringpointers, letztes Element `NULL`, jedes Element in der Form "*Name*=*Wert*"). Bei den Funktionen ohne `e` hinten erbt das neue Programm das Environment des laufenden Programmes.

`p` **wie** `PATH`**:** Das angegebene *command* braucht nicht mit vollem Pfad angegeben werden: Wenn es keinen `/` enthält, wird entsprechend der Environment-Variable `PATH` gesucht. Bei den Funktionen ohne `p` muß *command* mit dem vollen, absoluten Pfad angegeben werden.

Die `exec`-Funktionen sind in `unistd.h` deklariert.

Bei Erfolg kehrt `exec` nicht zurück, bei Fehler ist der Returnwert `-1`.

Es ist üblich, daß man als *arg0* bzw. *argv*`[0]` den Namen des aufgerufenen Programmes übergibt (am besten komplett mit Pfad).

Vererbung:

Das neue Programm übernimmt vom alten:

- Pid und PPid.
- Das Environment (außer bei `execle` und `execve`), das Current Working Directory und das Current Root Directory.
- Die `umask` und die `ulimit`-Werte (z. B. max. File- und Speichergröße, max. Laufzeit).
- Ignorierte Signale, die Signal-Maske, und blockierte Signale, sowie einen ev. noch ausstehenden `alarm`.
- Das Controlling Terminal.
- Record Locking.

Nicht geerbt werden:

- Programmcode und Daten.

---

[7] Auch wenn es nicht im POSIX-Standard steht, versteht `exec` heute allerdings in praktisch allen Unix-Varianten Scriptfiles, die in der ersten Zeile ausdrücklich mit `#!`*shell* beginnen (also z. B. `#!/usr/bin/bash`), und startet die angegebene *shell* mit dem Scriptfile als Input. Dieser Mechanismus läßt sich auch dazu gebrauchen, z. B. Perl- oder Awk-Skriptfiles direkt mit `exec` zu starten, indem man den jeweiligen Interpreter mit `#!` angibt.

- Signal-Handler (werden auf `SIG_DFL` zurückgesetzt).
- Shared-Memory-Segmente, die das alte Programm attached hatte.

Sonderfälle:

- Offene Files werden normalerweise vererbt: Für jeden offenen File gibt es im Kernel ein *close-on-exec*-Flag, das man mit der Funktion `fcntl` setzen kann. Ist es gesetzt, wird der File bei einem `exec` geschlossen, wenn nicht (der Default-Fall), bleibt er offen.
  Das ist allerdings nur für die Files `stdin`, `stdout` und `stderr` allgemein brauchbar und sinnvoll. Auf andere als diese drei Files sollte sich das aufgerufene Programm nur dann verlassen, wenn es ganz sicher weiß, wer sein Vorgänger war und was der hinterlassen hat (z. B. "Filedescriptor `5` ist die Pipe zum Lesen von meinem Vater" oder "Filedescritor `4` ist der gemeinsame Logfile"). Vererbt wird in diesen Fällen nur der offene Filedescriptor, aber *kein* File-Pointer (sinnvollerweise `fdopen` aufrufen)!
- Ist am Executable File des neuen Programms das Setuid-Bit (bzw. das Setgid-Bit) gesetzt, bekommt das neue Programm die Rechte des File Owners des Executables, sonst bekommt es dieselben Rechte wie das alte Programm[8].

Beispiele:

In den meisten Fällen macht man ein `fork` und dann im neuen Sohn (nach Aufräumarbeiten wie Files schließen, Signale umsetzen usw.) ein `exec`; nur in ganz seltenen Fällen macht das Hauptprogramm selbst ein `exec` (Beispiele: `login`, Environment-Putzer).

```
/* Beispiel zu pipe und fork mit exec am hinteren Ende der Pipe
 * (entspricht popen(..., "w"))
 * Gib 100 Zufallszahlen mit "more" aus
 */

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

#define count 100

int main(void)
{
  pid_t pid;
  int fd[2];
  FILE *outf;
  int i, exitstat;

  if (pipe(fd)<0) {
    perror("pipe"); exit(1);}
  if ((pid=fork())<0) {
    perror("fork"); exit(1);}

  if (pid==0) { /* Sohn */
    close(fd[1]);  /* Schliesse Schreib-Ende */
```

[8] Mit anderen Worten: Die zur Rechteprüfung verwendete "Effective User Id", unter der das Programm läuft, ist nicht die des Ausführenden, sondern die des File Owners.

```
    if (fd[0]!=STDIN_FILENO) {   /* else ohnehin schon richtiger File! */
      if (dup2(fd[0], STDIN_FILENO)<0) {  /* Kopiere Lese-Ende auf stdin */
        perror("dup2"); exit(1);}
      close(fd[0]);  /* Schliesse altes Lese-Ende */
    }
    if (execlp("more", "more", NULL)<0) {
      perror("exec"); exit(1);}
  }

  else { /* Vater */
    close(fd[0]);  /* Schliesse Lese-Ende */
    if ((outf=fdopen(fd[1], "w"))==NULL) {  /* outf auf Schreib-Ende oeffnen */
      perror("fdopen"); exit(1);}
    srand(getpid());
    for (i=0; i<count; ++i) {
      fprintf(outf, "%05u\n", rand()); fflush(outf);}
    fclose(outf);  /* Zumachen, damit Sohn enden kann! */
    if (waitpid(pid, &exitstat, 0)<0) {
      perror("waitpid"); exit(1);}
    /* ev. noch exitstat pruefen */
  }

  exit(0);
}

/* Beispiel zu pipe und fork mit exec am vorderen Ende der Pipe
 * (entspricht popen(..., "r"))
 * Gib alle Subdir's von . aus, die mehr als 100 KB Platz brauchen
 */

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

#define limit 100

int main(void)
{
  pid_t pid;
  int fd[2];
  FILE *inf;
  int exitstat;
  char line[PATH_MAX+30];

  if (pipe(fd)<0) {
    perror("pipe"); exit(1);}
  if ((pid=fork())<0) {
    perror("fork"); exit(1);}

  if (pid==0) { /* Sohn */
    close(fd[0]);  /* Schliesse Lese-Ende */
```

```
    if (fd[1]!=STDOUT_FILENO) {   /* else ohnehin schon richtiger File! */
      if (dup2(fd[1], STDOUT_FILENO)<0) {  /* Kopiere Schreib-Ende auf stdout */
        perror("dup2"); exit(1);}
      close(fd[1]);  /* Schliesse altes Schreib-Ende */
    }
    if (execlp("du", "du", "-k", NULL)<0) {
      perror("exec"); exit(1);}
  }

  else { /* Vater */
    close(fd[1]);  /* Schliesse Schreib-Ende */
    if ((inf=fdopen(fd[0], "r"))==NULL) {  /* inf auf Lese-Ende oeffnen */
      perror("fdopen"); exit(1);}
    while (fgets(line, sizeof(line), inf)) {
      if (atol(line)>=limit) fputs(line, stdout);}
    /* ev. auf ferror(inf) pruefen */
    fclose(inf);  /* Zumachen */
    if (waitpid(pid, &exitstat, 0)<0) {
      perror("waitpid"); exit(1);}
    /* ev. noch exitstat pruefen */
  }

  exit(0);
}
```

Manchmal braucht man doch eine Shell zum Starten des gewünschten Programmes (z. B. für Wildcards oder mehrere Commands in einer Pipe). Sofern man dafür nicht überhaupt `system` verwendet, sieht der `exec`-Aufruf in etwa wie folgt aus:

```
char *cmd;
cmd=...;
execl("/bin/sh", "sh", "-c", cmd, NULL);
```

Wenn man mit `exec` eigene Programme ausführen möchte, die nicht in einem fixen System-Directory stehen, muß man entweder die Environment-Variable `PATH` entsprechend setzen und dann `execlp` oder `execvp` verwenden, oder — falls die Programme im Current Directory stehen — beispielsweise `getcwd` aufrufen und vor den Programmnamen setzen: Es gibt leider keine einfache und sichere Möglichkeit, das Verzeichnis herauszufinden, aus dem das gerade laufende Programm geladen wurde, um es für weitere `exec`-Aufrufe zu verwenden[9].

9 Named Pipes: `mkfifo`

Was sind Named Pipes?

Unnamed Pipes (von `pipe`) haben einige Schwächen:

- Sie sind nur zwischen Prozessen möglich, die einen gemeinsamen Vaterprozeß haben.
- Sie können nicht von Unix-Befehlen und Programmen angesprochen werden, die den Namen des zu bearbeitenden Files auf der Commandline übergeben bekommen.

[9] Unter Linux ist `/proc/self/exe` ein symbolischer Link auf den Executable File des gerade laufenden Programmes. Dieser läßt sich mit der Funktion `readlink` ermitteln; daraus läßt sich dann das Verzeichnis extrahieren.

- Filedeskriptoren sind unkomfortabel.

Eine Named Pipe kombiniert

- einen normalen Filenamen (Eintrag in einem Directory auf der Platte)
- mit dem Mechanismus einer Pipe.

Named Pipes werden in der Fachliteratur auch *FIFO*s genannt.

Eine Named Pipe läßt sich wie ein normaler File mit `open` oder `fopen` zum Lesen oder Schreiben öffnen und kann daher wie ein normaler Filenamen an Programme übergeben werden (nur werden die Daten eben nicht auf Platte, sondern in eine Pipe geschrieben bzw. von dort gelesen).

Bei den Default-Einstellungen hängt ein `open` oder `fopen` zum Lesen, bis ein anderer Prozeß die Named Pipe zum Schreiben öffnet, und umgekehrt. So ist sichergestellt, daß das `open`/`fopen` erst dann einen offenen File liefert, wenn auch das andere Ende der Pipe von mindestens einem Prozeß geöffnet ist. Zu beachten ist nur, daß der letzte Leser seinen File erst nach dem letzten Schreiber schließen sollte (sonst bekommt der Schreiber wie bei Unnamed Pipes beim nächsten Schreib-Aufruf das Signal `SIGPIPE`).

Es gibt einen Trick, der vor allem bei Servern nützlich ist. Er verhindert, daß das `fopen` zum Lesen (im Server) hängt, bis ein Schreiber (Client) die Named Pipe öffnet, und daß der Leser (der Server) jedesmal `EOF` bekommt und ein neues `fopen` machen muß, wenn der letzte gerade aktive Schreiber (Client) die Named Pipe schließt: Wenn man das `fopen` auf die Named Pipe read/write macht (mit Modus `"r+"`), funktioniert es sofort, auch wenn es noch keinen Schreiber gibt, und der File bekommt nie ein `EOF`.

Bei `ls -l` werden Named Pipes mit einem `p` in der ersten Spalte und ev. einem `|` hinten am Namen gekennzeichnet, sie haben genauso Owner, Group und Rechte wie normale Files.

Anlegen von Named Pipes:

Shell-Kommando: `mkfifo` *filename*
C-Funktion: `int mkfifo(const char *`*filename*`, mode_t` *permissions*`)`

Die Funktion liefert `0` bei Erfolg, `-1` bei Fehler.

Achtung: Wenn es schon einen File oder eine Named Pipe mit dem gleichen Namen gibt, kommt ein Fehler (`errno EEXIST`)!

- Entweder extra auf diesen Fehler prüfen und in diesem Fall nicht abbrechen (problematisch, wenn das, was da existiert, ein File oder Directory anstatt einer Pipe ist),
- oder unmittelbar vor dem `mkfifo` vorbeugend ein `remove` machen.

Man braucht `#include <sys/types.h>` für `mode_t` und `#include <sys/stat.h>` für die Deklaration von `mkfifo` und die Konstanten für *permissions* (das sind die gleichen wie für `open` und `creat`).

Der Directory-Eintrag einer Named Pipe bleibt bestehen, bis man ihn explizit löscht. Die Pipe-Datenstruktur im Kernel wird beim ersten Öffnen einer Pipe automatisch angelegt und beim letzten Schließen automatisch wieder freigegeben.

Beispiele:

```
...
if (mkfifo("myfifo", S_IRUSR | S_IWUSR) < 0) {
  fprintf(stderr, "%s: mkfifo failed: %s\n", argv[0], strerror(errno));
  exit(1);
}
...
/* Schreib-Prozess */
...
FILE *outf;
...
outf=fopen("myfifo", "a");
...
/* Lese-Prozess */
...
/* vergleiche das, was ich auf outf schreibe, mit dem Inhalt von masterfile */
execlp("diff", "diff", "myfifo", "masterfile", NULL);
...
```

Auf der Shell kann man Named Pipes dazu verwenden, Kommandos in "nichtlinearer" Anordnung durch Pipes zu verbinden, ohne temporäre Dateien auf der Platte zu benutzen (sinnvoll, wenn diese Dateien groß wären).

1. Programm mit 2 Pipes als Input: Finde die gleichen Worte aus den Man-Pages von `printf` und `scanf`:

   ```
   mkfifo /tmp/p1 ; man printf | tr -cs "[A-Z][a-z]" "[\012*]" | sort -u >/tmp/p1 &
   mkfifo /tmp/p2 ; man scanf | tr -cs "[A-Z][a-z]" "[\012*]" | sort -u >/tmp/p2 &
   comm -12 /tmp/p1 /tmp/p2
   ```

2. Pipe mit Verzweigung (T-Stück): Speichere alle Filenamen, die auf `.c` oder `.h` enden, in `~/chfiles`, und alle anderen in `~/otherfiles`.

   ```
   mkfifo /tmp/p1 ; grep -v "\.[ch]$" </tmp/p1 >~/otherfiles &
   find / | tee /tmp/p1 | grep "\.[ch]$" >~/chfiles &
   ```

10 /proc, select, Client-Server-Systeme

Das Pseudo-Filesystem `/proc`:

In den Dateien und Directories im Filesystem `/proc` findet man Informationen zum System (Hardware-Konfiguration, Speicherverwaltung, ...) und zu den gerade laufenden Prozessen (ausgeführtes Command, Environment, Current Working Directory, offene Files, ...): Das Verzeichnis `/proc/`*pid*`/` enthält Informationen über den Prozeß *pid*, `/proc/self` bezieht sich immer auf den aufrufenden Prozeß (d. h. den, der gerade auf `/proc` zugreift). So ist es auf einigen Unix-Systemen über `/proc` zum Beispiel möglich, herauszufinden, in welchem Verzeichnis das gerade ausgeführte Executable liegt (wofür es sonst keine einfache, portable Möglichkeit gibt).

Das Filesystem `/proc` ist ein Pseudo-Filesystem, d. h. die Files und Directories unter `/proc` existieren nicht wirklich auf der Platte, sondern werden vom Kernel nur vorgetäuscht: Die Dateistruktur unter `/proc` wird dynamisch erzeugt (je nach laufenden Prozessen, angeschlossenen Devices usw.), beim Zugriff auf eine Datei unter `/proc` werden in Wirklichkeit Kernel-Informationen gelesen oder geschrieben.

Trotzdem kann man auf die Informationen unter `/proc` wie auf normale Files zugreifen, sowohl von Programmen aus (mit `fopen`, `fgets` usw.), also auch händisch von der Shell aus (mit `ls`, `more` usw.).

Struktur und Inhalt von `/proc` sind nicht standardisiert: Nicht jedes Unix bietet ein `/proc`, und wenn doch, so sind Struktur und Inhalt von Unix zu Unix verschieden.

Die Funktion `select`:

Die Funktion `select` erlaubt es, mehrere Files gleichzeitig "im Auge zu behalten", ob Daten gelesen oder geschrieben werden können (wobei es sich bei den Files typischerweise nicht um normale Platten-Files, sondern um Terminals, Netzverbindungen, Pipes und andere Kommunikationsverbindungen handelt, die Daten "stückchenweise" liefern oder entgegennehmen):

```
int select(int maxfd, fd_set *in_fds, fd_set *out_fds, fd_set *xcpt_fds,
           struct timeval *timeout)
```

- *in_fds*, *out_fds* und *xcpt_fds* sind Mengen von Filedeskriptoren: `select` prüft die in *in_fds* enthaltenen Files, ob sie neue Daten zum Lesen enthalten (d. h. ob ein `read` sofort zurückkehren würde, ohne zu hängen), die in *out_fds* enthaltenen Files, ob weitere Daten in sie geschrieben werden könnten (d. h. ob ein `write` sofort zurückkehren würde), und die in *xcpt_fds* enthaltenen Files, ob bestimmte Sonderfälle (Exceptions, im Falle von Terminals und Netzverbindungen) auf diesen Files eingetreten sind.

  Wenn man beispielsweise nur Files auf Input prüfen möchte, kann man für *out_fds* und *xcpt_fds* `NULL` übergeben.

- `select` wartet, wenn es keinen bereiten File findet, und kehrt zurück, sobald zumindest einer der angegebenen Files bereit zur gewünschten Operation ist.

  Achtung: Ein File ist bereit zum Lesen, wenn entweder Daten da sind, oder wenn `EOF` aufgetreten ist (denn auch bei `EOF` kehrt ein `read` sofort zurück!).

- `select` verändert *in_fds*, *out_fds* und *xcpt_fds*: Nach der (erfolgreichen) Rückkehr enthalten die Deskriptormengen genau jene Filedeskriptoren, die für die gewünschte Operation bereit sind; die anderen Deskriptoren werden vom `select` aus den Deskriptormengen gelöscht.

- Implementiert sind die Deskriptormengen `fd_set` als Bitfelder: Ist das n-te Bit `1`, gehört der Filedeskriptor n zur Menge, ist es `0`, gehört er nicht dazu.

  Zugreifen sollte man auf Variablen vom Typ `fd_set` nur über die dafür definierten Makros: `FD_ZERO(fd_set *`*fds*`)` initialisiert *fds* auf die leere Menge, `FD_SET(int` n`, fd_set *`*fds*`)` fügt den Deskriptor n zur Menge *fds* dazu (und `FD_CLR` entfernt ihn analog), und `FD_ISSET(int` n`, fd_set *`*fds*`)` prüft, ob n in *fds* enthalten ist.

- *maxfd* ist die Anzahl der auszuwertenden Bits in den drei Deskriptormengen: Wenn der höchste zu prüfende Filedeskriptor n ist, so muß man *maxfd* auf n`+1` setzen (nachdem die Filedeskriptoren und damit auch der Index der Bits bei `0` beginnen).

- *timeout* ist ein Timeout-Wert, `struct timeval` ist eine Struktur mit Sekunden- und Mikrosekunden-Komponente: Nach Ablauf der Zeit *timeout* kehrt `select` in jedem Fall zurück, auch wenn kein File bereit ist.

  Will man warten, ohne daß jemals ein Timeout anspricht, übergibt man `NULL` als *timeout*, will man gar nicht warten, sondern nur schnell alle Files durchprüfen, übergibt man eine Struktur mit beiden Komponenten gleich `0` als Zeit.

Als "Abfallprodukt" kann man `select(0, NULL, NULL, NULL, &timeout)` als `sleep` mit Mikrosekunden-Auflösung (real meist Hundertstelsekunden) verwenden.

- `select` liefert als Returnwert
  - die Anzahl der für I/O bereiten Files, wenn welche bereit sind,
  - `0`, wenn das Timeout abgelaufen ist, ohne daß ein File bereit wäre,
  - und `-1`, wenn ein Fehler aufgetreten ist (Beispiel: `EINTR`).
- Die notwendigen Headerfiles sind im Beispiel angegeben.

`select` ist effizienter als selbstgebastelte Lösungen mit ständigem, zyklischem Abprüfen aller Files in einer Schleife: `select` macht *kein* "busy waiting", es verbraucht keine Rechenleistung, solange sich der Zustand der angegebenen Files nicht ändert.

Beispiel:

```
/* Beispiel zu select: Lies wahlweise von stdin oder von einer Pipe */

...
#include <sys/types.h>    /* fuer fd_set */
#include <sys/time.h>     /* fuer struct timeval */
#include <unistd.h>       /* fuer select */

#define timeout_sec 5
#define timeout_microsec 0
#define max(a, b) ((a)<(b)?(b):(a))
#define bufsize 4096
...

  int fd[2];
  int maxfd;
  fd_set infds;
  struct timeval timeout;
  int n;
  char buf[bufsize];

  ... pipe(fd) ...

  FD_ZERO(&infds);
  FD_SET(STDIN_FILENO, &infds);
  FD_SET(fd[0], &infds);
  maxfd=max(STDIN_FILENO, fd[0])+1;
  timeout.tv_sec=timeout_sec; timeout.tv_usec=timeout_microsec;
  n=select(maxfd, &infds, NULL, NULL, &timeout);
  if (n<0) {
    fprintf(stderr, "%s: select failed: %s\n", argv[0], strerror(errno));
    exit(1);}
  else if (n==0) {
    ...  /* Timeout im select */ }
  else {
    if (FD_ISSET(fd[0], &infds)) {  /* Pipe ist bereit zum Lesen */
      if ((n=read(fd[0], buf, bufsize))>0) {
        ...  /* n Bytes gelesen */  }
      else if (n==0) {
```

```
      ...  /* EOF beim read */  }
    else {
      ...  /* Error beim read */  }
  }
  if (FD_ISSET(STDIN_FILENO, &infds)) {  /* stdin ist bereit zum Lesen */
    ...  /* analog fuer stdin */  }
}
```

Client-Server-Systeme:

Viele Applikationen sind heute Client-Server-Systeme, d. h. sie bestehen aus zwei Programmen, dem ***Client*** und dem ***Server***:

Der Client dient zum Zugriff auf die vom Server verwalteten Daten oder Ressourcen: Der Client schickt eine Anfrage oder einen Befehl an den Server, der Server schickt das Ergebnis zurück zum Client. Ein Server kann dabei viele Clients gleichzeitig bedienen.

Typischerweise gilt:

Der Client ...	**Der Server** ...
läuft am Arbeitsplatz des Benutzers	läuft auf zentralen Rechnern
wird bei Bedarf gestartet	läuft rund um die Uhr
kommuniziert mit dem Benutzer	läuft im Hintergrund, ohne Terminal
(Tastaturinput, Fenster)	(***"Daemon-Prozeß"***)
speichert keine Daten lokal	verwaltet zentrale Datenbestände
ist ein rein sequentielles Programm	erzeugt mehrere / viele Prozesse

Beispiele für Client-Server-Systeme sind:

- Fast alle Netzwerkdienste (WWW, FTP, Telnet, Mail, Newsgroups, File- und Printsharing, Directory Services, ...).
- Datenbanken und Systeme zur betrieblichen Datenverarbeitung (z. B. Oracle, SAP).
- Kommunikations-, Workflow- und Terminplanungs-Systeme (MS Exchange + Outlook, Lotus Notes, ...).
- Unix X Window System ("verkehrt": Server am Arbeitsplatz!).

Client und Server (sowie die Serverprozesse untereinander) kommunizieren entweder über Netzverbindungen, Named Pipes, Streams, Sockets, Message Queues usw., oder über Shared Memory.

Server verwenden normalerweise mehrere Prozesse, denn auf der Serverseite sollte nie der Hauptprozeß selbst die Requests von Clients bearbeiten: Ein einzelner Client könnte sonst mit einem mutwillig großen Request, einer hängenden Verbindung, oder dem Verursachen eines Fehlers oder Absturzes den ganzen Server lahmlegen!

Typische Arbeitsweisen von Servern sind folgende:

- Alle Requests von den Clients werden vom Server-Vaterprozeß entgegengenommen. Er startet pro Request einen neuen Sohnprozeß, dieser bearbeitet den Request, schickt die Antwort zum Client, und endet wieder.

  Früher arbeiteten beispielsweise fast alle Webserver nach diesem Prinzip. Da diese Arbeitsweise aber sehr ineffizient ist (ein `fork`-Aufruf pro abgefragtem File!) und der Vater dabei zum Engpaß werden kann, wird sie bei Hochleistungs-Webservern heute nicht mehr verwendet.

- Eine Weiterentwicklung besteht darin, daß der Server-Vaterprozeß von vornherein eine fixe Anzahl von Sohnprozessen startet. Einlangende Requests werden nach wie vor vom Vaterprozeß entgegengenommen, dieser reicht sie dann zur Bearbeitung an einen gerade freien Sohnprozeß weiter bzw. speichert sie zwischen, bis ein Sohnprozeß frei wird.

  Die Söhne enden nun nicht mehr, nachdem sie einen Request abgearbeitet haben, sondern laufen wie der Vater in einer Endlosschleife und bearbeiten einen Request nach dem anderen.

  Diese Struktur liegt einigen modernen Webservern zu Grunde, ebenso beispielsweise SAP.

- Eine andere Architektur teilt den Söhnen nicht einzelne Requests zu, sondern ganze Sitzungen: Wenn ein Client gestartet wird, meldet er sich beim Server-Vaterprozeß an, und dieser erzeugt daraufhin einen neuen Sohn eigens für diesen Client.

  Der Client richtet dann alle seine Requests direkt an diesen Sohnprozeß, der Sohn ist ausschließlich für die Bedienung dieses einen Clients zuständig, zwischen den beiden besteht eine ständige Verbindung (ohne Beteiligung des Vaters). Wenn der Client endet, endet auch der ihm zugeordnete Sohnprozeß im Server.

  Diese Struktur ist typisch für Datenbanken, auch der Samba-Fileserver arbeitet so (ein Sohnprozeß pro zu bedienendem PC).

- Da viele Netzwerkdienste nach der soeben beschriebenen Arbeitsweise vorgehen, hat man dafür in Unix einen zentralen Vater-Prozeß geschaffen, um die ständig laufenden Vater-Prozesse pro Dienst einzusparen: Den `inetd` (Internet-Daemon). Er liest aus seinem Konfigurationsfile (`/etc/inetd.conf`), für welche Netzdienste er auf neue Verbindungswünsche von Clients warten soll, und welches Programm für den jeweiligen Dienst zuständig ist.

  Möchte sich ein Client an einem solchen Dienst anmelden, nimmt der `inetd` die Verbindung entgegen, startet mit `fork` und `exec` das dazugehörige Serverprogramm in einem neuen Sohnprozeß, und verbindet den Client damit.

  Viele der klassischen Netzdienste wie FTP und Telnet sind üblicherweise so implementiert.

Achtung:

- Wenn man einen echten Server-Daemon schreiben will, der sich beim Start automatisch in den Hintergrund versetzt und völlig unabhängig von der startenden Shell weiterarbeitet, sind am Anfang des Programmes einige Initialisierungsschritte notwendig, die wir hier nicht durchnehmen:
  * Ein `fork` mit anschließendem Beenden des Vaterprozesses, damit das aufrufende Terminal (bzw. das Startscript oder was immer) sofort weitermachen kann (und nicht auf das Ende des Daemons wartet), und damit der Daemon `1` (und nicht die aufrufende Shell) als Parent Pid bekommt.
  * Ein Aufruf von `setsid` zum Anlegen einer neuen Prozeßgruppe und einer neuen Session, zum Versetzen in den Hintergrund und zum Abhängen vom Controlling Terminal. Macht man das nicht, bekommt der Daemon unter Umständen das Signal SIGHUP, wenn die Terminalsitzung endet, von der aus er gestartet wurde, und wird dadurch ebenfalls beendet (oder ein SIGINT beim nächsten Ctrl/C am Terminal).
- Nachdem ein Daemon kein Controlling Terminal hat, ist es wenig sinnvoll, einfach auf `stderr` zu schreiben (wohin?). Man könnte `stderr` beim Start entsprechend umleiten

(z. B. auf `/dev/console`), am besten ist es aber, den Unix-`syslog`-Mechanismus zur Ausgabe von Fehlermeldungen zu benutzen.

`syslog` ist ein eigener Dienst mit ständig laufendem Serverprozeß `syslogd`, der alle anfallenden Logmessages vom Kernel, Hintergrundprozessen usw. mit Datum, Uhrzeit, Rechner- und Programmname versieht und je nach Konfigurationsfile filtert und in diverse Logfiles (unter Linux meist in `/var/log/messages`) oder auf die Konsole schreibt oder an `syslogd`'s auf anderen Rechnern zur zentralen Verwaltung weiterschickt.

11 SysV IPC, Grundlagen

Name:

IPC steht für *I*nter*P*rocess*C*ommunication, und *SysV* ist die übliche Abkürzung für jene Unix-Version, die diese Mechanismen eingeführt hat (AT&T Unix System V).

Bestandteile:

SysV IPC bietet drei verschiedene Mechanismen zur Kommunikation und Synchronisation zwischen mehreren Prozessen desselben oder verschiedener Programme:

Shared-Memory-Segmente: Hierbei handelt es sich um Speicherbereiche, die von mehreren Prozessen gemeinsam benutzt werden können[10].

Semaphore: Hierbei handelt es sich um einen Mechanismus, der zur Synchronisation und zur Sequentialisierung paralleler Prozesse dient, um einen geordneten Zugriff auf gemeinsame Ressourcen zu erreichen.

Message Queues: Message Queues sind ein unidirektionales Kommunikationsmedium, das es ähnlich wie Pipes erlaubt, Daten von einem Prozeß zu einem anderen zu senden.

Shared-Memory-Segmente und Semaphore werden in fast jeder größeren Server-Applikation benötigt, sie sind vor allem zentraler Bestandteil jeder Datenbank-Implementierung. Die Bedeutung von Message Queues ist in den letzten Jahren stark zurückgegangen, weil sich deren Funktion auch mit zahlreichen anderen Mechanismen (Named Pipes, Stream Pipes, Netzwerkmechanismen, ...) erreichen läßt.

Gemeinsame Eigenschaften:

1. Alle SysV IPC Mechanismen werden vom Kernel im Hauptspeicher (nicht auf Platte!) realisiert. Daher
   - funktionieren sie nur innerhalb eines Rechners, nicht über das Netz,
   - und gehen beim Neustart des Rechners verloren.
2. Einmal angelegte SysV IPC-Objekte bleiben bestehen, bis
   - sie von einem Programm explizit wieder gelöscht werden,
   - sie händisch mit dem Unix-Befehl `ipcrm` (siehe Man-Page) entfernt werden,
   - oder der Rechner neu gestartet wird.

[10] Unter Unix gibt es inzwischen verschiedenste Implementierungen von Shared Memory: Die hier besprochenen "klassischen" SysV Shared-Memory-Segmente, die davon abweichenden Shared-Memory-Mechanismen der aktuellen C-Standards, die Möglichkeit, Shared Memory über die Funktion `mmap` zu realisieren, und die in Linux 2.4 für SAP implementierte Variante über ein Pseudo-Filesystem ähnlich `/proc`.

Sie verschwinden ***nicht*** automatisch, wenn der anlegende Prozeß endet, oder wenn sie von keinem Prozeß mehr benutzt werden.

Nachdem die betreffenden Kernel-Ressourcen endlich sind (und Shared Memory auch Swap-Space belegt, sobald es einmal benutzt wurde), ist das wiederholte Anlegen von IPC-Objekten (vor allem Shared-Memory-Segmenten) ohne entsprechendes Aufräumen eine sehr wirksame Methode, ein System durch Belegung aller Ressourcen komplett zum Stillstand zu bringen.

Es ist daher sehr zu empfehlen,

- einen Exit-Handler (siehe `atexit` oder `on_exit`) für das Aufräumen der Objekte zu schreiben,
- und alle wichtigen Signale abzufangen und das Programm in diesem Fall ebenfalls mit einem `exit` zu beenden,

damit die IPC-Objekte bei jedem möglichen Programmende entfernt werden[11].

3. Ein Shared-Memory-Segment bleibt nach dem Löschen so lange bestehen, bis der letzte verwendende Prozeß es freigibt, erst dann wird es wirklich gelöscht[12].

   Semaphore und Message Queues werden hingegen beim Löschen wirklich sofort gelöscht, auch wenn es noch Prozesse gibt, die sie noch verwenden: Diese Prozesse bekommen beim nächsten Zugriff einen Fehler (`errno EIDRM`).

4. Die Eigenschaften der gerade existierenden IPC-Objekte kann man mit dem Unix-Befehl `ipcs` am Terminal ansehen: Per Default alle, `-m` nur Shared-Memory-Segmente, `-s` nur Semaphore, `-q` nur Message Queues, per Default Kurzform, mit `-a` Langform[13].

   Die Zugriffsrechte auf IPC-Objekte funktionieren ähnlich wie die auf Files: Je 3 Bits geben die Rechte für den Owner des Objektes, Mitglieder der Group des Objektes, und andere Benutzer an. Ein Bit steuert das Read-Recht, eines das Write-Recht (bei Semaphoren heißt es "Alter" statt "Write"), und das dritte ist unbenutzt (Makros für die Berechtigungsbits siehe unten). Damit kann ein IPC-Objekt bei entsprechend gesetzten Rechten auch von Programmen verschiedener Benutzer verwendet werden.

   Neben den Rechten, dem aktuellen Owner und der aktuellen Group enthält die Statusinformation zu einem IPC-Objekt noch den Owner und die Group zum Zeitpunkt des Anlegens, sowie je nach Typ des Objektes Pid und Zeitpunkt des Anlegens bzw. Änderns, Pid und Zeitpunkt der letzten Datenoperation auf das Objekt, Größe des Objektes, Anzahl der benutzenden Prozesse, ...

---

[11] Das Bestehenbleiben dieser Objekte über die Lebenszeit der beteiligten Prozesse hinweg hat aber auch sinnvolle Einsatzmöglichkeiten: Man kann damit z. B. Daten auch zwischen den einzelnen Aufrufen eines Programmes im Speicher halten und sich dadurch das neuerliche Laden der Daten von der Platte bei jedem Programmstart sparen.

Ein weiterer beliebter Trick besteht darin, die Pid eines Programmes beim Start in einem kleinen Shared-Memory-Segment abzulegen. Andere Programme können dann einfach feststellen, ob das Programm seit dem letzten Reboot schon einmal lief, und welche Pid es haben müßte, falls es noch aktiv ist. So kann man beispielsweise mehrfaches Starten eines Programmes erkennen.

[12] Ein Shared-Memory-Segment verhält sich beim Löschen ähnlich wie ein File: Wenn man einen File löscht, wird der Directory-Eintrag auch sofort gelöscht, und der File kann damit nicht mehr neu geöffnet werden. Prozesse, die ihn schon offen haben, können aber weiter mit dem File arbeiten, denn der File selbst verschwindet erst, nachdem der letzte Prozeß ihn geschlossen hat.

Man kann ein Shared-Memory-Segment also "vorbeugend" löschen, sobald es von allen Prozessen, für die es gedacht ist, verwendet wird. Es funktioniert weiter, aber es ist sichergestellt, daß es nicht überbleibt, sondern nach dem letzten Prozeß verschwindet, auch wenn das Programm abstürzt.

[13] Das sind die auf den meisten Unix-Systemen üblichen Optionen. Unter Linux hat `ipcs` andere Optionen, vor allem hat `-a` eine andere Bedeutung!

Key und Identifier:

Ein IPC-Objekt wird durch zwei Zahlen identifiziert: Den Key (vom Typ `key_t`, definiert in `sys/types.h`, typischerweise als `int` oder `long`) und den Identifier (vom Typ `int`).

Der Key: Der Key ist ein vom Programmierer gewählter "Name" für jedes seiner IPC-Objekte: Wollen mehrere Programme auf ein und dasselbe IPC-Objekt zugreifen, muß jedes davon den Key kennen, denn zum Anlegen eines IPC-Objektes oder zum Zugriff auf ein bestehendes IPC-Objekt muß man dessen Key angeben.

Der Identifier: Der Identifier ist ein vom System vergebener "Pointer" für jedes im System gerade vorhandene IPC-Objekt: Jedem IPC-Objekt wird beim Anlegen vom Kernel zur Laufzeit ein Identifier zugewiesen. Dieser Identifier wird vom System frei gewählt (es handelt sich üblicherweise um Indices oder Pointer in irgendwelche Kernel-Tabellen), er ist eindeutig, aber nicht direkt aus dem Key berechenbar[14].

Die Zuordnung von Identifiern zu Keys erledigen die `get`-Funktionen: Beim Aufruf der `get`-Funktion gibt man den Key des gewünschten Objektes an und erhält als Ergebnis dessen Identifier.

Alle anderen Daten- und Kontrolloperationen spezifizieren das betroffene IPC-Objekt durch Angabe seines Identifiers, nicht seines Keys.

Will man von mehreren unabhängigen Programmen auf dieselben IPC-Objekte zugreifen, ist der Ablauf daher folgender: Alle diese Programme müssen dieselben Keys verwenden (pro Objekt einen). Jedes für sich ruft einmal ein `get` mit diesem Key auf und erhält vom System als Ergebnis den Identifier des betreffenden Objektes (beim ersten `get`-Aufruf wird das Objekt neu angelegt und ein neuer Identifier dafür vergeben, alle weiteren Aufrufe liefern dann den Identifier des bestehenden Objektes). Mit diesem Identifier kann das Programm dann alle weiteren Operationen auf dem IPC-Objekt ausführen.

Der Grund für diese Zweigleisigkeit Key / Identifier war die Forderung nach höchstmöglicher Effizienz der IPC-Operationen: Der Identifier ist für den schnellstmöglichen Zugriff auf IPC-Objekte im Kernel optimiert, der Key für die Verwaltung der Objekte aus Sicht des Anwendungsprogrammes. Würden alle IPC-Operationen mit dem Key erfolgen, müßte der Kernel bei jeder Operation zuerst das dem Key zugeordnete Objekt suchen.

`ipcs` zeigt sowohl den Key als auch den Identifier jedes Objektes an, bei `ipcrm` kann man wahlweise (mit `-m` / `-s` / `-q`) den Identifier des zu löschenden Objektes angeben, oder (mit `-M` / `-S` / `-Q`) seinen Key[15].

Das Key-Problem:

Das Key-Konzept hat mehrere Tücken:

1. Man kann zwar für das eigene Programmpaket nach Belieben Keys wählen, aber es ist nie ganz auszuschließen, daß ein anderes Programmpaket für seine IPC-Objekte nicht zufällig

---

[14] Während der Lebenszeit eines Objektes liefert dessen Key immer das dasselbe Objekt und damit auch denselben Identifier (und kein anderer Key liefert diesen Identifier), aber nach dem Löschen des Objektes kann demselben Key beim neuerlichen Anlegen des Objektes ein anderer Identifier zugewiesen werden, je nachdem, welchen Platz das neue Objekt im Kernel bekommt. Ebenso kann der alte Identifier nach dem Löschen wieder einem anderen Key zugeordnet werden.

Eine Ausnahme bildet der Key `IPC_PRIVATE`: Auch wenn mehrere Objekte gleichzeitig den gleichen Key `IPC_PRIVATE` haben, haben diese Objekte natürlich trotzdem stets *verschiedene* Identifier.

[15] Auch `ipcrm` hat unter Linux andere Optionen als auf den meisten anderen Unix-Systemen; unter Linux kann man `ipcrm` nur mit Identifiern, nicht aber mit Keys aufrufen!

die gleichen Keys gewählt hat: Dann sprechen beide plötzlich unabsichtlich dieselben IPC-Objekte an!

Im günstigsten Fall fällt das sofort auf, weil einem der beiden die Zugriffsrechte auf die vom anderen angelegten Objekte fehlen oder beispielsweise die Objektgrößen nicht zusammenpassen, im ungünstigsten Fall dürfen tatsächlich beide dieselben Objekte gemeinsam benutzen, was zu den seltsamsten Fehlverhalten führen kann.

Abhilfe:

- Bei Programmen, die jedem im Source verfügbar sind, kann man den Key in irgendeinem Header-File als Konstante definieren: Bei Key-Kollisionen ändert man die Definition und übersetzt das Programmpaket frisch.
- Bei Programmpaketen, die nur in Binärform zum Kunden gelangen, muß der Key kundenseitig angegeben werden können (beispielsweise, indem man ihn aus einem Konfigurationsfile ausliest oder beim Programmstart als Argument oder Environmenr-Variable übergibt).
- Eine Alternative bietet die Funktion `ftok`, die aus einem File eine als Key geeignete Zahl ermittelt[16]: Wenn jedes Programm des Programmpaketes `ftok` auf ein und demselben File aufruft (z. B. auf irgendeinem Konfigurationsfile), kann jedes Programm für sich denselben Key dynamisch ermitteln, ohne daß dieser irgendwo zentral konfiguriert werden muß.

  Sollte der von `ftok` ermittelte Key mit einem Key eines anderen Programmpaketes kollidieren, reicht es, den verwendeten File einmal umzukopieren, und schon liefert `ftok` einen anderen Key.

2. Ein Programmpaket mit fix vergebenem Key (oder z. B. einem `ftok`-Aufruf auf einen fixen File) kann natürlich auf einem Rechner nur einmal seine IPC-Objekte anlegen: Es ist nicht möglich, mehrere Instanzen des Programmpaketes gleichzeitig und unabhängig voneinander (d. h. mit jeweils eigenen IPC-Objekten) zu starten (und es gab in den Anfängen der SysV-IPC-Mechanismen tatsächlich kommerzielle Datenbankprodukte, bei denen man aus genau diesem Grund nur eine Datenbank pro Rechner betreiben konnte).

   In diesem Fall muß man die Keys entweder irgendwie dynamisch ermitteln, oder für jede Instanz eine eigene Nummer in Konfigurationsfiles speichern, oder `ftok` auf verschiedene Files verwenden (bei Programmen, die genau einmal pro User laufen können, bietet sich z. B. das Home-Verzeichnis des jeweiligen Users für ein `ftok` an).

Der Key `IPC_PRIVATE` und seine Verwendung:

Ein Sonderfall ist der Key `IPC_PRIVATE` (üblicherweise `0`): Gibt man beim `get` diesen Key an, wird *immer* ein neues Objekt angelegt, auch wenn schon ein anderes Objekt mit dem Key `IPC_PRIVATE` existiert: `IPC_PRIVATE` ist der einzige Key, zu dem es mehrere Objekte gleichzeitig geben kann.

Diese Objekte werden *private Objekte* genannt, weil es anderen Prozessen nicht möglich ist, durch Angabe desselben Keys beim `get` ebenfalls Zugriff auf dieses Objekt zu erlangen.

Damit ist auch die Verwendung dieses Keys naheliegend: Will man IPC-Objekte nur innerhalb eines einzigen Programmaufrufes (d. h. nur in Prozessen, die alle von einem gemeinsamen

[16] Aufruf: `key_t ftok(char *`*path*`, char` *id*`)`. *path* ist der volle Pfad eines existierenden Files oder Directories, und *id* (ein beliebiger Buchstabe) dient dazu, um aus demselben *path* mehrere Keys generieren zu können: Man kann z. B. die Keys für drei Shared-Memory-Segmente mit `'0'`, `'1'` und `'2'` erzeugen und die der dazugehörigen Semaphore mit `'a'`, `'b'` und `'c'`. `ftok` liefert als Ergebnis eine als Key geeignete Zahl oder `-1` bei Fehler. Das Ergebnis hängt nur vom Directory-Eintrag des Files *path* ab, nicht von dessen Inhalt.

Vaterprozeß abstammen) verwenden, so ruft dieser Vaterprozeß einmal die `get`-Funktion mit dem Key `IPC_PRIVATE` zum Anlegen der Objekte auf (das garantiert auch gleich, daß für jeden Programmlauf tatsächlich *neue* Objekte unabhängig von anderen Aufrufen desselben Programmes angelegt werden), und vererbt die erhaltenen Identifier beim `fork` an seine Söhne weiter. Diese können dann mit den Identifiern direkt (ohne neuerliches `get`) auf diese Objekte zugreifen.

`IPC_PRIVATE` hat noch eine zweite Sonderbedeutung: Der Key noch verwendeter Shared-Memory-Segmente wird beim Löschen auf `IPC_PRIVATE` umgesetzt. Damit wird das Segment sozusagen unsichtbar, kein Prozeß kann mehr mit `get` frisch darauf zugreifen[17].

Funktionen:

Alle drei IPC-Mechanismen haben sehr ähnliche Kontroll- und Verwaltungsfunktionen, wir besprechen sie daher hier gemeinsam. Nur die eigentlichen Datenoperationen auf Shared-Memory-Segmente, Semaphoren und Message Queues unterscheiden sich grundlegend und werden daher in den einzelnen Kapiteln besprochen.

Die `get`-Funktionen:

Prüfen, ob es schon ein IPC-Objekt mit dem Key *key* gibt, legen es eventuell neu an, und returnieren den Identifier des Objektes:

- `int shmget(key_t` *key*`, int` *size*`, int` *flags*`)`
  für ein *size* Bytes großes Shared-Memory-Segment[18].
- `int semget(key_t` *key*`, int` *semcnt*`, int` *flags*`)`
  für eine Semaphoren-Menge mit *semcnt* einzelnen Semaphoren[19].
- `int msgget(key_t` *key*`, int` *flags*`)`
  für eine Message-Queue.

Flags:

Für den Parameter *flags* muß das logische "Oder" der gewünschten Berechtigungen für das Objekt sowie wahlweise der Konstanten `IPC_CREAT` und `IPC_EXCL` angegeben werden.

Fälle für `IPC_CREAT` *und* `IPC_EXCL`:

1. Der *key* ist `IPC_PRIVATE`: Es wird immer ein neues IPC-Objekt erzeugt.
2. Der *key* ist nicht `IPC_PRIVATE`, und das Flag `IPC_CREAT` ist nicht angegeben: Ein IPC-Objekt mit dem Key *key* muß bereits existieren, sonst liefert der `get`-Aufruf einen Fehler (`errno ENOENT`).
3. Der *key* ist nicht `IPC_PRIVATE`, das Flag `IPC_CREAT` ist angegeben, aber `IPC_EXCL` ist nicht angegeben: Existiert bereits ein IPC-Objekt mit dem Key *key*, so liefert der `get`-Aufruf den Identifier des bestehenden Objektes, anderenfalls wird das Objekt neu angelegt.
4. Der *key* ist nicht `IPC_PRIVATE`, und die Flags `IPC_CREAT` und `IPC_EXCL` sind beide angegeben: Existiert bereits ein IPC-Objekt mit dem Key *key*, so liefert der `get`-Aufruf einen Fehler (`errno EEXIST`), andernfalls wird das Objekt neu angelegt.

---

[17] Auf ein bestehendes `IPC_PRIVATE`-Objekt kann man nicht mit `get` zugreifen, und zu seinem ursprünglichen Key gehört das Objekt nicht mehr, dieser Key bewirkt daher beim `get`, daß ein neues Shared-Memory-Segment angelegt wird. Wirklich gelöscht wird das alte Segment aber erst, wenn der letzte Prozeß es freigibt.

[18] Wenn man nur auf ein schon bestehendes Objekt zugreifen will, dessen Größe man u. U. nicht kennt, kann man als *size* `0` angeben.

[19] Wir werden meist den Fall *semcnt*`==1` brauchen, d. h. Semaphoren-Mengen mit einer einzigen Semaphore, und unterscheiden daher sprachlich nicht zwischen einer Semaphore und einer Semaphoren-Menge.

Rechte:

1. Die Lese-Rechte für den Owner heißen `SHM_R`, `SEM_R` und `MSG_R`[20].
2. Die Schreib- bzw. Änderungs-Rechte für den Owner heißen `SHM_W`, `SEM_A` und `MSG_W`.
3. Für die entsprechenden Rechte für die Group müssen die Konstanten 3 Bits nach rechts geschoben werden, für die Other-Rechte 6 Bits.

Beispiel: Flags für ein Shared-Memory-Segment, das noch nicht existieren darf, sowie alle Rechte für den Owner sowie Leserechte für Group und Other bekommen soll:
`IPC_CREAT | IPC_EXCL | SHM_R | SHM_W | (SHM_R>>3) | (SHM_R>>6)`

Returnwert und Fehler:
Ist die Funktion erfolgreich, wird als Ergebnis der Identifier des Objektes zurückgegeben. Bei Fehler ist das Ergebnis `-1`.

Die wichtigsten Fehler sind folgende:

1. Die oben genannten Fälle im Zusammenhang mit `IPC_CREAT` und `IPC_EXCL` (existiert nicht, aber `IPC_CREAT` ist nicht angegeben, oder existiert schon, aber `IPC_EXCL` ist angegeben).
2. Das IPC-Objekt existiert bereits, aber die Rechte erlauben den Zugriff nicht (`errno EACCES`).
3. Das Objekt kann nicht angelegt werden, entweder, weil die angegebenen Parameter (z. B. die Shared-Memory-Größe) die vom System vorgegebenen Limits überschreiten, oder weil nicht mehr genug Ressourcen frei sind.

Die `ctl`-Funktionen:
Dienen zum Ausführen diverser Status- und Kontrollfunktionen auf das IPC-Objekt mit dem Identifier *id*:
`int shmctl(int` *id*`, int` *command*`, struct shmid_ds *`*arg*`)`
`int semctl(int` *id*`, int` *semnum*`, int` *command*`, union semun` *arg*`)`
`int msgctl(int` *id*`, int` *command*`, struct msgid_ds *`*arg*`)`

Operationen:
Die auszuführende Operation wird durch den Parameter *command* bestimmt:

1. `IPC_RMID`: Das IPC-Objekt wird gelöscht[21]. Für *arg* ist `NULL` anzugeben.
2. `IPC_STAT`: Die Kontrollinformation des IPC-Objektes (Owner, Rechte, Zeitstempel usw.) wird ausgelesen und in *arg* gespeichert.
3. `IPC_SET`: Bestimmte Felder der Kontrollinformation des IPC-Objektes werden auf die in *arg* angegebenen Werte gesetzt.
4. Für Semaphore gibt es noch weitere Operationen, u. a. zum Initialisieren der Semaphore. Falls sich die Operation auf eine einzelne Semaphore in der Semaphoren-Menge bezieht, ist deren Index in *semnum* anzugeben[22], sonst übergibt man in *semnum* üblicherweise `0`.

Returnwert:
`0` bei Erfolg, `-1` bei Fehler.

---

[20] Unter Linux sind meist nur die `SHM`-Konstanten definiert. Man kann diese auch für `SEM` und `MSG` verwenden oder die Rechte überhaupt als Oktalzahl angeben.

[21] Semaphoren und Message Queues werden sofort gelöscht. Shared-Memory-Segmente werden auf `IPC_PRIVATE` gesetzt und dann gelöscht, wenn der letzte Prozeß sie freigibt oder endet.

[22] Da wir üblicherweise nur eine Semaphore verwenden, wird *semnum* bei uns meist `0` sein.

Headerfiles:

Man benötigt in jedem Fall den Headerfile `sys/ipc.h` (und `sys/types.h` für `key_t`) sowie je nach verwendetem IPC-Mechanismus `sys/shm.h`, `sys/sem.h` und `sys/msg.h`.

12 Shared-Memory-Segmente, Grundlagen

Definition:

Ein Shared-Memory-Segment ist ein zusammenhängender Speicherbereich von beim Erzeugen wählbarer Größe[23], der von mehreren Prozessen gleichzeitig gemeinsam benutzt werden kann: Jeder dieser Prozesse kann Daten in diesem Speicherbereich lesen und schreiben, und wenn ein Prozeß Daten in diesem Speicherbereich schreibt, sind diese Daten sofort und ohne weitere Maßnahmen in allen anderen Prozessen sichtbar.

Damit sind Shared-Memory-Segmente die einfachste und vor allem effizienteste Möglichkeit zum Datenaustausch zwischen Prozessen: Alle beteiligten Prozesse können auf die Daten zugreifen, ohne daß die Daten kopiert werden müssen, und ohne daß bei den einzelnen Datentransfers der Aufruf einer Betriebssystem- oder Library-Funktion notwendig wäre.

Funktionen:

Zusätzlich zu den allgemeinen IPC-Funktionen zum Anlegen und Löschen gibt es zwei Funktionen speziell für Shared-Memory-Segmente:

`void *shmat(int` *id*`, void *`*adr*`, int` *flags*`)`

`shmat` verbindet den aufrufenden Prozeß mit dem Shared-Memory-Segment mit dem Identifier *id* (von `shmget`): `shmat` blendet das Segment in einen bisher unbenutzten bzw. ungültigen Bereich des Adressraumes des aufrufenden Prozesses ein (wobei es dem Betriebssystem überlassen ist, an welcher Stelle des Adressraumes das Segment zu liegen kommt).

Im Englischen heißt diese Operation "attach the segment to the process", im Deutschen oft mit "Anbinden des Segmentes an den Prozeß" übersetzt.

Bei Erfolg liefert `shmat` einen Pointer auf den Anfang des Shared-Memory-Segments, bei Fehler `-1` (Achtung: Vom Typ `void *`, nicht `int`!). Nach `shmat` "sieht" der aufrufende Prozeß das Shared-Memory-Segment also von der Adresse *ptr* bis zur Adresse *ptr*`+`*shm_size*`-1`, wobei *ptr* der Returnwert von `shmat` und *shm_size* die Größe des Segmentes ist.

Für *adr* übergibt man üblicherweise `NULL`[24], für *flags* `0`[25].

`int shmdt(void *`*shm_adr*`)`

`shmdt` trennt die Verbindung zwischen dem aufrufenden Prozeß und dem an der Adresse *shm_adr* (von `shmat`) eingeblendeten Shared-Memory-Segment wieder: Der Prozeß kann

---

[23] Die Größe des Speicherbereiches wird beim Anlegen fixiert, Shared-Memory-Segmente können nicht dynamisch wachsen.

[24] In *adr* könnte man eine Adresse für das Einblenden des Segmentes vorgeben. Nachdem man aber nicht weiß, welche Adressbereiche durch Code, Daten und Stack des Prozesses sowie durch andere Shared-Memory-Segmente bereits belegt sind und welche Regeln das jeweilige Betriebssystem für die Platzierung von Shared-Memory-Segmenten vorgibt, ist dieses Feature praktisch nicht verwendbar und vor allem nicht portabel.

[25] Oder `SHM_RDONLY`, wenn der Prozeß in dem Shared-Memory-Segment nur lesen, aber nicht schreiben dürfen soll.

danach nicht mehr auf das Segment zugreifen, der betreffende Bereich seines Adressraumes ist wieder undefiniert[26].

`shmdt` löscht das Segment *nicht*: Das Segment bleibt bestehen, auch wenn kein Prozeß mehr damit verbunden ist. Nur wenn das Segment bereits mit `shmctl IPC_RMID` gelöscht worden ist, verschwindet es in dem Moment, wo sich der letzte Prozeß mit `shmdt` davon trennt.

Der Returnwert ist `0` bei Erfolg, `-1` bei Fehler.

Die Funktionen haben zahlreiche Fehlerfälle (siehe `man`-Page): Fehlende Rechte, Überschreitung eines Limits oder fehlende Ressourcen, ungültiger Identifier, ...

Limits:

Nachdem Shared-Memory-Segmente Kernel-Datenstrukturen zur Verwaltung erfordern und außerdem den Hauptspeicher eines Systems sehr nachhaltig füllen können, sind sie zahlreichen Limits unterworfen, die im Kernel konfiguriert werden und systemweit gelten:

`SHMMAX`: Maximale Größe (Bytes) eines Segmentes (früher typischerweise einige Megabytes, heute auch Gigabytes).
`SHMALL`: Maximale Gesamtgröße (Pages) aller Segmente im System (sinnvollerweise deutlich weniger als Hauptspeicher-Größe plus Swapspace-Größe!).
`SHMSEG`: Maximale Anzahl der Segmente, die ein Prozeß gleichzeitig verwenden kann (typischerweise 10).
`SHMMNI`: Maximale Anzahl der Segmente systemweit (typischerweise einige 100).

Wichtig:

Shared-Memory-Segmente werden nicht automatisch initialisiert, weder beim ersten `shmget` noch beim ersten `shmat`. Auf vielen Unix-Systemen enthalten sie zwar zu Beginn stets lauter Nullbytes, auf anderen aber Garbage! Daher:

- Wenn ein bestehendes Segment verwendet werden soll: `shmget(..., 0)`
- Für ein neues: `shmget(..., shm_perm | IPC_CREAT | IPC_EXCL)`
- Wenn beides sein kann: `shmget` für bestehendes, auf Fehler prüfen, zweites `shmget` für neues (oder umgekehrt)!

Beispiel:

Das folgende Programm liest aus dem Shared-Memory-Segment mit dem Key 666666 den darin gespeicherten Text aus und zeigt ihn an. Text, den man eingibt, wird zeilenweise an den Text im Shared-Memory-Segment angehängt (außer Leerzeilen), nach jeder Eingabezeile wird wieder der im Shared-Memory-Segment neu dazugekommene Text angezeigt.

Paßt eine Zeile nicht mehr ins Segment, wird das Segment gelöscht und das Programm beendet. Wenn am Input `EOF` erreicht wird, endet das Programm ebenfalls, läßt aber das Segment bestehen.

---

[26] Der freigegebene Adressbereich kann z. B. für das Anbinden anderer Shared-Memory-Segmente verwendet werden. Nachdem ein Prozeß bei 32-bit-Prozessoren auf 4 GB Adressraum beschränkt ist und daher nie mehr als 4 GB Daten gleichzeitig ansprechen kann, ist das abwechselnde Einblenden verschiedener Shared-Memory-Segmente auf derartigen Prozessoren die einzige Möglichkeit, einem einzelnen Prozeß die Verwaltung von mehr als 4 GB Daten im Hauptspeicher zu erlauben (wenn auch nicht gleichzeitig). Dieser Trick wird z. B. von SAP genutzt.

```
/* Shared-Memory-Segment als gemeinsamer Textspeicher */
/* Tip zum Probieren: Programm mehrmals gleichzeitig aufrufen! */
/* Achtung: Version ohne Semaphore! Kann schiefgehen!!! */

#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <errno.h>

#define linelen 80
#define shm_key 666666
#define shm_size 16000
#define shm_perm (SHM_R | SHM_W | (SHM_R>>3) | (SHM_W>>3) | (SHM_R>>6) | (SHM_W>>6))

int main(void)
{
  int shm_id;
  char *shm_ptr, *next, *end;
  int new_seg=0;
  char line[linelen+2];

  /* zuerst pruefen, ob es das Segment schon gibt! */
  shm_id=shmget(shm_key, shm_size, 0);
  if (shm_id==-1) {  /* Fehler! */
    if (errno==ENOENT) {  /* Shm gibt's noch nicht, neu anlegen! */
      new_seg=1;
      shm_id=shmget(shm_key, shm_size, shm_perm | IPC_CREAT | IPC_EXCL);
      if (shm_id==-1) {  /* wieder Fehler! */
        perror("shmget new segment failed"); exit(1);}}
    else {  /* anderer Fehler */
      perror("shmget existing segment failed"); exit(1);}}
  shm_ptr=(char *)(shmat(shm_id, NULL, 0));
  if (shm_ptr==(char *)(-1)) {
    perror("shmat failed"); exit(1);}

  if (new_seg) {  /* Neu angelegtes Segment: Auf Leerstring initialisieren */
    *shm_ptr='\0';}
  /* next ist die Position, ab der der Text noch nicht ausgegeben wurde */
  next=shm_ptr;

  for (;;) {
    fputs(next, stdout);  /* neuen Text ab next ausgeben */
    next=strchr(next, '\0');  /* ... und dessen Ende merken */
    /* lies eine Input-Zeile, wenn EOF Schleife beenden */
    fputs(">>> ", stdout); fflush(stdout);
    if (fgets(line, sizeof(line), stdin)==NULL) break;
    if (*line=='\n') continue;  /* Leere Zeile: Ignorieren, nicht dazutun */
    end=strchr(next, '\0');  /* suche aktuelles Ende des Textes im Shm */
    if (end+strlen(line)>=shm_ptr+shm_size) {  /* Zeile passt nicht mehr! */
      fputs("Text storage full!\n", stderr);
      if (shmctl(shm_id, IPC_RMID, NULL)==-1) {  /* Segment loeschen */
        perror("shmctl IPC_RMID failed"); exit(1);}
      break;}  /* Schleife verlassen */
    strcpy(end,line);  /* Zeile hinten dran ins Shm kopieren */
  }

  if (shmdt(shm_ptr)==-1) {  /* Segment abhaengen */
```

```
    perror("shmdt failed"); exit(1);}
  exit(0);
}
```

Diesem Beispiel fehlt noch:

- Eine sinnvolle Strukturierung des Shared-Memory-Segmentes.
- Ein Schutz vor "unglücklichen" gleichzeitigen Zugriffen: Bei diesem Programm ist sowohl der unbemerkte Verlust von Daten als auch ein Absturz (`SIGSEGV`) möglich!

13 Shared-Memory-Speicherverwaltung

Pointer:

Pointer in Shared-Memory-Segmenten zu speichern ist nicht sinnvoll!!!

- Pointer auf Daten außerhalb von Shared-Memory-Segmenten (d. h. auf lokale Daten eines bestimmten Prozesses) sind für alle anderen Prozesse nutzlos, weil sie ja per Definition nicht auf den Speicher des Prozesses, der den Pointer erzeugt hat, zugreifen können.

  Im günstigsten Fall bewirkt ein Zugriff auf einen solchen Pointer ein `SIGSEGV` (wenn die Adresse im lokalen Prozeß ungültig ist), im Normalfall wird ein solcher Pointer-Zugriff irgendwelche ganz anderen, nicht vorhersagbaren lokalen Daten (die im zugreifenden Prozeß gerade an dieser Adresse stehen) liefern.

- Pointer auf Daten im selben oder einem anderen Shared-Memory-Segment sind noch heimtückischer: Sie funktionieren tadellos in jenen Prozessen, bei denen das betreffende Shared-Memory-Segment beim `shmat` vom Betriebssystem zufällig an die gleiche Stelle im Adressraum eingeblendet wurde wie in jenem Prozeß, der den Pointer erzeugt hat.

  Liegt das Shared-Memory-Segment jedoch in dem Prozeß, der den Pointer verwendet, an einer anderen Stelle des Adressraumes (und das kann von Rechner zu Rechner, ja sogar von Programmlauf zu Programmlauf auf ein und demselben Rechner variieren), so ist ein `SIGSEGV` oder der Zugriff auf nicht vorhersagbare Daten die Folge (weil sich ja mit der Anfangsadresse des Segmentes auch die Adresse aller Daten darin verschiebt).

Wenn man in Shared-Memory-Segmenten verkettete Listen, Bäume oder dergleichen speichern will, muß man also an Stelle von Pointern *Indices* zur Verkettung verwenden!

- Siehe unten: Man kann irgendwelche Array-Deklarationen auf das Shared-Memory-Segment anwenden (also das Segment beispielsweise als Array von Strukturen oder als Struktur von Arrays betrachten), und die Indices in diesen Arrays als Referenzen für die Listen- oder Baumelemente verwenden.
- Als Alternative kann man das ganze Segment als ein Array von Bytes betrachten und an Stelle eines Pointers auf eine Datenstruktur deren Index in diesem Array (d. h. deren Adresse relativ zum Segment-Anfang) verwenden. Diesen Index erhält man, indem man vom Pointer auf die Datenstruktur den Pointer auf den Segment-Anfang (beide gecastet

auf `char *`) subtrahiert. Wenn man zum Pointer auf den Segment-Anfang (wieder `char *`) den Index addiert, erhält man wieder den Pointer auf die gewünschte Datenstruktur[27].

Speicherverwaltung:

Weder die Sprache C noch die C Libraryfunktionen bieten Hilfsmittel zur Speicherverwaltung in Shared-Memory-Segmenten: Es ist weder möglich, in Shared-Memory-Segmenten Variablen zu deklarieren, noch ist es möglich, Daten in einem Shared-Memory-Segment mittels `malloc` und `free` dynamisch zu verwalten.

`shmat` liefert einen typlosen Pointer (`void *`) auf ein Shared-Memory-Segment, das eine bestimmte Anzahl von Bytes groß ist, und es ist Aufgabe des Programmierers, welche Daten er wo in diesem Bereich speichert und wie er auf diese Daten zugreift. Folgende 4 Varianten kommen häufig vor:

Verwendung von `char *`:

Im einfachsten Fall reicht es aus, den von `shmat` gelieferten Pointer auf `char *` (oder bei Bedarf auch `int *` usw.) zu casten, und das Shared-Memory-Segment einfach wie einen String oder ein Array zu verwenden.

Beispiel siehe voriges Kapitel.

Struktur fixer Größe:

Am elegantesten und sichersten arbeitet man mit Shared-Memory-Segmenten, deren Inhalt sich durch eine Struktur (oder eine Union, oder ein Array) definieren läßt:

- Im `shmget` verwendet man ein `sizeof` auf diesen Struktur-Typ, damit das Shared-Memory-Segment die richtige Größe bekommt.
- Das Ergebnis von `shmat` castet man auf einen Pointer auf diese Struktur.
- Jeder Zugriff auf das Shared-Memory-Segment erfolgt dann über diesen Pointer auf die entsprechenden Struktur- oder Array-Elemente, so, als ob der Pointer auf eine ganz normale Struktur-Variable zeigen würde.

Beispiel siehe Übungs-Musterlösungen.

Struktur variabler Größe:

In vielen Fällen möchte man die Größe des Shared-Memory-Segmentes nicht fix im Sourcecode festlegen, sondern dynamisch konfigurierbar gestalten (z. B. über ein Argument auf der Command-Line oder einen Parameter in einem Konfigurationsfile).

Nachdem C nicht prüft, ob der Index bei einem Zugriff auf ein Array-Element innerhalb der Grenzen des Arrays liegt, und bei der Indizierung grundsätzlich annimmt, daß das Array groß genug ist, funktioniert es problemlos, auf mehr Elemente zuzugreifen, als man in der Deklaration angegeben hat, solange man darauf achtet, daß man tatsächlich so viel Speicher reserviert hat, wie man in Wahrheit verwendet, und nichts anderes in diesem Speicher steht.

Man deklariert in der Struktur also beispielsweise ein Array ohne Elemente (mit `[0]`)[28], sorgt aber zur Laufzeit dafür, daß an dieser Stelle Speicherplatz für die gewünschte Anzahl

---

[27] Der Gnu C Compiler macht zwar das intuitiv Richtige, wenn man mit `void`-Pointern Arithmetik betreibt, aber laut C-Standard ist das undefiniert, also sollte man `char`-Pointer verwenden.

[28] Man könnte statt `[0]` auch einfach `[]` verwenden: C erlaubt es, daß die *letzte* Komponente einer Struktur-Typdefinition ein Array unbekannter Größe ist. Das wäre zwar noch eleganter, aber leider erlaubt C auf solche "unvollständigen" Strukturen kein `sizeof`, womit die Größenberechnung beim `shmget` etwas komplizierter wird.

von Elementen zur Verfügung steht, und kann dann auf diese Elemente zugreifen, als ob die richtige Anzahl von Elementen in der Deklaration gestanden wäre.

Der einzige Unterschied zur Verwendung einer Struktur fixer Größe besteht also darin, daß man die Größe des Segmentes beim `shmget` jetzt selbst richtig festlegen muß (als Summe der Größen des konstanten Teils und des variablen Teiles, letztere ergibt sich aus Elementzahl mal Elementgröße, erstere als `sizeof` der Struktur).

Das variabel große Array muß natürlich immer die ***letzte*** Komponente der Struktur sein, damit es zusätzlichen Speicherplatz hinter der eigentlich deklarierten Struktur (und keinen anderen Speicherplatz!) korrekt anspricht: Wäre es in der Mitte der Struktur, würden die zur Laufzeit "dazugeschwindelten" Arrayelemente und die hinter dem Array liegenden Komponenten der Struktur denselben Speicherplatz belegen!

Beispiel siehe unten.

Definition einer eigenen dynamischen Speicherverwaltung:

Braucht man mehr Flexibilität, muß man selbst Funktionen analog zu `malloc` und `free` für Shared-Memory-Segmente schreiben. Diese könnten beispielsweise folgende Aufrufe haben:

`void shminit(void *`*shm_ptr*`, size_t` *shm_size*`)` initialisiert die dynamische Speicherverwaltung im Shared-Memory-Segment *shm_ptr* der Größe *shm_size* (*shm_ptr* ist dabei das, was `shmat` geliefert hat).

`void *shmalloc(void *`*shm_ptr*`, size_t` *n_bytes*`)` liefert einen Pointer auf einen dynamisch allokierten Speicherbereich der Größe *n_bytes* im Shared-Memory-Segment *shm_ptr*.

`void shmfree(void *`*shm_ptr*`, void *`*ptr*`)` gibt den von `shmalloc` gelieferten Speicherbereich *ptr* im Segment *shm_ptr* wieder frei.

Entsprechende Algorithmen zur dynamischen Speicher-Verwaltung (z. B. mit verketteten Listen freier Bereiche oder nach dem Buddy-Prinzip) sind in der Literatur beschrieben, für eine Behandlung hier sind sie zu umfangreich.

Beispiel:

Das folgende Programm realisiert eine einfache Datenbank im Hauptspeicher: Pro Tabelle wird ein Shared-Memory-Segment angelegt, jeder Programmaufruf führt eine Operation auf die Tabelle durch.

Jede Tabelle speichert Paare bestehend aus einem Schlüssel und den zum Schlüssel gehörenden Daten, wobei sowohl der Schlüssel als auch die Daten beliebige Strings sein können (z. B. Fremdwort / Übersetzung, Name / Telefonnummer und Email-Adresse, Artikelnummer / Artikelstammdaten). Neben dem Einfügen neuer Paare und dem Anzeigen der Daten zu gegebenem Schlüssel ist das Ausgeben einer sortierten Liste aller Paare implementiert. Weiters kann eine Tabelle in einem File gespeichert und von diesem geladen werden, wobei die Daten direkt (d. h. binär, ohne Formatierung) zwischen File und Shared Memory transferiert werden.

Die Einträge sind als Binärbaum organisiert (wir beschränken uns in diesem Beispiel auf einen unbalancierten Binärbaum, auch wenn dieser bei ungünstiger Datenreihenfolge sehr ineffizient ist). Um Schlüssel und Daten beliebiger Länge effizient (d. h. ohne Speicherplatzverlust) speichern zu können, werden sie nicht unmittelbar in den Knoten des Binärbaumes in einzelnen `char`-Arrays fixer Größe abgelegt, sondern "dicht auf dicht" in einem eigenen String-Speicherbereich außerhalb der Baumknoten.

Jedes Shared-Memory-Segment besteht daher aus drei Teilen: Dem Kopf mit den Verwaltungsdaten, einem Array, dessen Elemente die Baumknoten sind, und einem Array für die Schlüssel- und Daten-Strings. Die Größe der beiden Arrays kann zur Laufzeit (beim Anlegen des Segmentes) festgelegt werden.

Zwei wichtige Operationen wurden nicht implementiert, um die Speicherverwaltung überschaubar zu halten:

- Um ein Update (d. h. ein Ändern bestehender Werte) zu realisieren, wäre es notwendig, den String-Bereich dynamisch zu verwalten, um den Speicherplatz von alten Daten wieder nutzen zu können, nachdem diese durch neue ersetzt wurden.
- Für ein Delete (d. h. ein Löschen einzelner Schlüssel-Daten-Paare) wäre neben einer Freispeicherverwaltung des String-Bereiches auch eine Freispeicherverwaltung im Knoten-Bereich notwendig, um die Knoten gelöschter Paare wieder für neue Paare nutzen zu können (außerdem ist das Löschen in Bäumen an sich nicht trivial).

```
/* Shared-Memory-Datenbank (Binaerbaum) */
/* Achtung: Version ohne Semaphore! Kann schiefgehen!!! */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <string.h>
#include <assert.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define helptext \
"db create table entries strsize  create new table\n" \
"(entries = max. nr of entries, strsize = max. size of key & data strings)\n" \
"db drop table                    delete table\n" \
"db save table filename           save table contents to file\n" \
"db load table filename           load table contents from file\n" \
"db insert table key=data ...     add new key/data pairs to table\n" \
"db list table                    list all key/data pairs\n" \
"db list table key ...            list individual key/data pairs\n"

/* Fehlermeldung bei falschem Aufruf */
void usage(const char *errmsg)
{
  fprintf(stderr, "Usage error: %s\n", errmsg);
  fputs(helptext, stderr);
  exit(2);
}

#define shm_perm (SHM_R | SHM_W | \
                  (SHM_R>>3) | (SHM_W>>3) | \
                  (SHM_R>>6) | (SHM_W>>6))

/* Wert fuer "leere" Indices (kann nicht 0 sein, weil 0 ist gueltiger Index) */
#define nil (-1)

/* Check auf gueltigen Baumknoten-Index */
#define i_check(i) assert(((i)>=0) && ((i)<shm_ptr->free_entry))
/* Abkuerzungen fuer Zugriffe auf die Felder eines Knotens */
```

```
#define i_left(i) (shm_ptr->entries[i].left)
#define i_right(i) (shm_ptr->entries[i].right)
#define i_key(i) (shm_ptr->entries[i].key)
#define i_data(i) (shm_ptr->entries[i].data)

/* Typ fuer einen Knoten des Binaerbaumes */
typedef struct {
  int left, right;   /* Index der Soehne im Binaerbaum (oder nil) */
  int key, data;}    /* Index des Key- und Datenstrings */
entry;

/* Typ fuer das ganze Shm-Segment */
typedef struct {
  int max_entry;     /* Tatsaechliche Groesse des Knoten-Arrays */
  int max_string;    /* Tatsaechliche Groesse des String-Arrays */
  int free_entry;    /* Index des ersten freien Knotens */
  int free_string;   /* Index des ersten freien Char's im String-Array */
  int root;          /* Index der Wurzel des Binaerbaumes (oder nil) */
  entry entries[0];} /* die Baum-Knoten (de facto Index 0...(max_entry-1)) */
                     /* und dahinter die Strings: Ab entries[max_entry] */
shm_type;

/* Pointer auf das Shared Memory (global, waere sonst ueberall Argument) */
shm_type *shm_ptr;

/* berechne String-Pointer auf den String mit String-Index i im Shm */
char *getstr(int i)
{
  char *str_beg;  /* Pointer auf Anfang des String-Bereiches im Shared Mem */
  assert((i>=0) && (i<shm_ptr->free_string));
  str_beg=(char *)(&(shm_ptr->entries[shm_ptr->max_entry]));
  return str_beg+i;
}

/* generiere einen Shm-Key aus dem Tabellennamen */
key_t table2key(const char *table)
{
  key_t shm_key=1;
  const char *p;
  /* wir multiplizieren einfach die Ascii-Werte aller Buchstaben des Namens */
  /* nicht perfekt, aber zum Austesten gut genug */
  for (p=table; *p!='\0'; ++p) shm_key*=(*p);
  return shm_key;
}

/* lege ein neues Shm-Segment an und initialisiere es */
void new_shm(key_t shm_key, const char *entrysize, const char *stringsize)
{
  int shm_id, shm_size;
  int entries, strings;

  entries=atoi(entrysize);
  if (entries<=0) usage("invalid number of entries");
  strings=atoi(stringsize);
  if (strings<=0) usage("invalid size for strings");

  shm_size=sizeof(shm_type)+entries*sizeof(entry)+strings*sizeof(char);
  shm_id=shmget(shm_key, shm_size, shm_perm | IPC_CREAT | IPC_EXCL);
  if (shm_id==-1) {perror("shmget new segment failed"); exit(1);}
```

```
  shm_ptr=(shm_type *)(shmat(shm_id, NULL, 0));
  if (shm_ptr==(shm_type *)(-1)) {perror("shmat failed"); exit(1);}

  shm_ptr->max_entry=entries; shm_ptr->max_string=strings;
  shm_ptr->free_entry=shm_ptr->free_string=0;
  shm_ptr->root=nil;
}

/* loesche Shm-Segment */
void drop_shm(key_t shm_key)
{
  int shm_id;
  shm_id=shmget(shm_key, 0, 0);
  if (shm_id==-1) {perror("shmget existing segment failed"); exit(1);}
  if (shmctl(shm_id, IPC_RMID, NULL)==-1) {  /* Segment loeschen */
    perror("shmctl IPC_RMID failed"); exit(1);}
}

/* binde bestehendes Shm-Segment an, setze shm_ptr */
void old_shm(key_t shm_key)
{
  int shm_id;
  shm_id=shmget(shm_key, 0, 0);
  if (shm_id==-1) {perror("shmget existing segment failed"); exit(1);}
  shm_ptr=(shm_type *)(shmat(shm_id, NULL, 0));
  if (shm_ptr==(shm_type *)(-1)) {perror("shmat failed"); exit(1);}
}

/* sichere das Shm-Segment binaer in einen File:
 * Zuerst drei Integers mit der Anzahl der benutzten Knoten,
 * der Anzahl der benutzten String-Char's, und dem Index des Root-Knotens
 * Dann den benutzten Teil von "entries" in einem Stueck
 * Dann den benutzten Teil des String-Bereiches in einem Stueck
 * Die max-Werte und unbenutzte Teile werden nicht mitgespeichert
 * Dadurch kann man die Daten auch in ein groesseres/kleineres Segment laden */
void save_shm(const char *filename)
{
  int fd;
  fd=open(filename, O_WRONLY|O_CREAT|O_TRUNC, S_IRUSR|S_IWUSR);
  if (fd<0) {perror("open for write failed"); exit(1);}
  /* das zeigt nur die Idee, es fehlen die Fehlerpruefungen beim write!!! */
  write(fd, (void *)(&(shm_ptr->free_entry)), sizeof(int));
  write(fd, (void *)(&(shm_ptr->free_string)), sizeof(int));
  write(fd, (void *)(&(shm_ptr->root)), sizeof(int));
  write(fd, (void *)(&(shm_ptr->entries[0])),
        (shm_ptr->free_entry)*sizeof(entry));
  write(fd, (void *)(&(shm_ptr->entries[shm_ptr->max_entry])),
        (shm_ptr->free_string)*sizeof(char));
  close(fd);
}

/* lade einen von save_shm geschriebenen File
 * bestehende Daten in der Tabelle werden ueberschrieben */
void load_shm(const char *filename)
{
  int fd;
  fd=open(filename, O_RDONLY);
  if (fd<0) {perror("open for read failed"); exit(1);}
  /* ... und hier waeren noch mehr Fehlerpruefungen noetig:
```

```
   * Lesefehler, File zu kurz oder zu lang?
   * passen die Daten in die Table, oder ist sie zu klein?
   * sind die Daten inhaltlich konsistent? */
  read(fd, (void *)(&(shm_ptr->free_entry)), sizeof(int));
  read(fd, (void *)(&(shm_ptr->free_string)), sizeof(int));
  read(fd, (void *)(&(shm_ptr->root)), sizeof(int));
  read(fd, (void *)(&(shm_ptr->entries[0])),
       (shm_ptr->free_entry)*sizeof(entry));
  read(fd, (void *)(&(shm_ptr->entries[shm_ptr->max_entry])),
       (shm_ptr->free_string)*sizeof(char));
  close(fd);
}

/* fuege ein neues Schluessel/Daten-Paar in den Baum ein */
void insert(char *pair)
{
  int *p, i, len, cmp;
  char *key, *data;           /* die beiden Strings */
  int key_index, data_index;  /* und deren Indices im String-Bereich */

  /* teile das Argument beim '=' in Schluessel- und Datenstring */
  key=pair;
  pair=strchr(pair, '=');
  if (pair==NULL) {usage("no '=' in 'key=data'"); return;}
  *pair='\0';   /* Schluessel endet beim '=' */
  data=pair+1;  /* Daten beginnen gleich hinter dem '=' */

  /* p soll auf jenen Speicherplatz fuer einen Index zeigen,
   * in den der Index des neuen Knotens hineingeschrieben gehoert
   * Bei leerem Baum ist das "root",
   * sonst dasjenige leere "left"- oder "right"-Feld
   * wo man den einzufuegenden Wert suchen wuerde */
  p=&(shm_ptr->root);
  for (;;) {
    i=*p;
    if (i==nil) break;  /* die richtige leere Stelle im Baum ist gefunden! */
    i_check(i);         /* Stelle nicht leer: Knoten i im Baum ... */
    cmp=strcmp(getstr(i_key(i)), key);  /* ... mit Schluessel vergleichen */
    if (cmp==0) {               /* gleich: den Schluessel gibt's schon! */
      printf("*** %s: duplicate key ***\n", key);
      return;}
    if (cmp>0) p=&(i_left(i));  /* groesser: im linken Teilbaum weitersuchen */
    else p=&(i_right(i));}      /* kleiner: im rechten Teilbaum weitersuchen */

  /* zuerst die beiden Strings in den String-Bereich kopieren */
  len=strlen(key)+1;
  if (shm_ptr->free_string+len>shm_ptr->max_string) {
    fputs("*** no space for string data ***\n", stdout);
    return;}
  key_index=shm_ptr->free_string; shm_ptr->free_string+=len;
  strcpy(getstr(key_index), key);
  len=strlen(data)+1;
  if (shm_ptr->free_string+len>shm_ptr->max_string) {
    fputs("*** no space for string data ***\n", stdout);
    return;}
  data_index=shm_ptr->free_string; shm_ptr->free_string+=len;
  strcpy(getstr(data_index), data);

  /* dann einen neuen Knoten-Eintrag anlegen und fuellen */
```

```
  if (shm_ptr->free_entry+1>shm_ptr->max_entry) {
    fputs("*** no space for new entry ***\n", stdout);
    return;}
  i=shm_ptr->free_entry; ++(shm_ptr->free_entry);
  i_left(i)=i_right(i)=nil; i_key(i)=key_index; i_data(i)=data_index;
  *p=i;  /* ... und im Baum einfuegen */
}

/* gib den Knoten mit gegebenem Key aus */
void list(const char *key)
{
  int i, cmp;
  i=shm_ptr->root;
  for (;;) {
    if (i==nil) {
      /* der Teilbaum, in dem der richtige Knoten sein muesste, ist leer! */
      printf("*** %s: no entry ***\n", key);
      return;}
    i_check(i);  /* Vergleiche den aktuellen Knoten des Baumes ... */
    cmp=strcmp(getstr(i_key(i)), key);  /* ... mit dem gesuchten Schluessel */
    if (cmp==0) {            /* gleich: gefunden! */
      printf("%s: %s\n", key, getstr(i_data(i)));
      return;}
    if (cmp>0) i=i_left(i);  /* groesser: im linken Teilbaum weitersuchen */
    else i=i_right(i);}      /* kleiner: im rechten Teilbaum weitersuchen */
}

/* gib den Teilbaum mit Wurzel i als sortierte Liste aus: */
void listtree(int i)
{
  if (i==nil) return;  /* Teilbaum ist leer, nichts auszugeben */
  i_check(i);
  listtree(i_left(i));  /* zuerst rekursiv den linken Teilbaum, ... */
  printf("%s: %s\n", getstr(i_key(i)), getstr(i_data(i))); /* die Wurzel, ...*/
  listtree(i_right(i));  /* und zuletzt rekursiv den rechten Teilbaum */
}

/* gib den ganzen Baum aus */
void listall(void)
{
  listtree(shm_ptr->root);
}

int main(int argc, char **argv)
{
  key_t shm_key;
  int i;

  if (argc<3) usage("too few arguments");
  shm_key=table2key(argv[2]);   /* verwandle Tabellenname in Shm-Key */
  if (strcmp(argv[1], "create")==0) {
    if (argc<5) usage("entries & strsize missing");
    if (argc>5) usage("too many args");
    new_shm(shm_key, argv[3], argv[4]);}
  else if (strcmp(argv[1], "drop")==0) {
    if (argc>3) usage("too many args");
    drop_shm(shm_key);}
  else {
    old_shm(shm_key);
```

```
    if (strcmp(argv[1], "save")==0) {
      if (argc<4) usage("filename missing");
      if (argc>4) usage("too many args");
      save_shm(argv[3]);}
    else if (strcmp(argv[1], "load")==0) {
      if (argc<4) usage("filename missing");
      if (argc>4) usage("too many args");
      load_shm(argv[3]);}
    else if (strcmp(argv[1], "insert")==0) {
      if (argc<4) usage("key=data missing");
      for (i=3; i<argc; ++i) insert(argv[i]);}
    else if (strcmp(argv[1], "list")==0) {
      if (argc==3) listall();
      else for (i=3; i<argc; ++i) list(argv[i]);}
    else usage("illegal command");}
  exit(0);
}
```

14 Semaphore, Grundlagen

Das Problem:

Wenn mehrere Prozesse gleichzeitig oder in ungünstiger zeitlicher Reihenfolge auf dieselben Daten in einem Shared-Memory-Segment zugreifen, kann das zu unerwarteten und schwer reproduzierbaren Fehlern führen.

Solche Fehler können entweder dadurch entstehen, daß die Prozeßverwaltung des Betriebssystems im "falschen" Moment Prozeßwechsel zwischen mehreren lauffähigen Prozessen durchführt, oder dadurch, daß auf Mehrprozessorsystemen mehrere Prozesse echt gleichzeitig ausgeführt werden.

Beispiele:

- Wenn zwei Prozesse gleichzeitig zu einem Kontostand, der in einem Shared-Memory-Segment gespeichert ist, etwas dazurechnen, geht je nach zeitlicher Abfolge eine der beiden Buchungen verloren (beide lesen denselben alten Wert, berechnen "ihren" neuen Wert, und schreiben ihn zurück. Der zuletzt geschriebene neue Wert überschreibt den zuerst geschriebenen, ohne dessen Buchung zu berücksichtigen).

- Wenn zwei Prozesse gleichzeitig verkettete Datenstrukturen manipulieren, können Knoten verlorengehen oder Datenstrukturen auseinanderfallen:

  Zwei Prozesse wollen beispielsweise ein Element an das Ende einer einfach verketteten Liste mit Tail-Pointer anhängen. Der erste setzt den Zeiger im letzten Listenelement auf sein neues Element. Bevor er den neuen Tail-Pointer speichern kann, kommt der zweite Prozeß, überschreibt diese Verkettung mit einem Pointer auf sein neues Element, und setzt den Tail-Pointer auf dieses neue Listenende. Danach beendet der erste Prozeß seine Operation, indem er den Tail-Pointer auf sein zuerst angehängtes Element (dessen Verkettung inzwischen überschrieben wurde!) zeigen läßt. Ergebnis ist eine in zwei Teile zerbrochene Liste!

- Auch unser Shared-Memory-Textspeicher-Beispielprogramm hat gleich zwei Probleme: Erstens passiert Unvorhersehbares, wenn zwei Prozesse gleichzeitig eine Zeile hinten

zum Text dazukopieren: Im günstigsten Fall geht eine der beiden Zeilen verloren, im ungünstigsten Fall werden die Buchstaben der Zeilen durcheinandergemischt.

Schwerer zu finden, aber noch schlimmer ist das zweite Problem: Solange ein Prozeß eine Zeile dazukopiert, ist der Text nicht ordnungsgemäß durch ein `\0`-Byte terminiert! Versucht ein zweiter Prozeß gleichzeitig das Textende zu finden, ist mit großer Wahrscheinlichkeit ein Absturz die Folge!

Kritische Regionen:

Eine kritische Region ist ein Stück Code, das "allein" ausgeführt werden muß, um die korrekte Funktion des Programmes zu garantieren: Es muß sichergestellt werden, daß kein anderer Prozeß gleichzeitig dasselbe Stück Code (oder ein anderes Stück Code, das die gleichen gemeinsamen Daten manipuliert) ausführen kann!

Eine kritische Region umfaßt dabei üblicherweise nicht nur eine einzelne Operation, sondern alle jene Operationen, die notwendig sind, ***um von einem alten konsistenten Stand der Daten zu einem neuen konsistenten Stand der Daten zu gelangen*** (z. B. alle Kontobewegungen im Zusammenhang mit einer Buchung oder alle Pointer-Zuweisungen, die für das Einfügen eines Knotens in einer Baumstruktur notwendig sind). Es muß sichergestellt sein, daß diese Operationen "in einem Rutsch" ausgeführt werden, d. h. zusammenhängend und ohne daß zwischen den einzelnen Operationen fremde Zugriffe auf die betreffenden gemeinsamen Daten erfolgen können.

Man spricht vom "Betreten" der kritischen Region, wenn man beginnt, die erste Operation des kritischen Programmstückes auszuführen, und vom "Verlassen", wenn man mit der letzten Operation dieses Programmstückes fertig wird.

Hier haben wir es mit kritischen Regionen im Zusammenhang mit Daten in Shared-Memory-Segmenten zu tun, das Konzept gilt aber allgemein für alle Arten gemeinsamer Ressourcen (z. B. gemeinsamer Zugriff mehrerer Prozesse auf denselben File oder gleichzeitiges Ansprechen desselben I/O-Gerätes).

Semaphore bildlich erklärt:

Semaphore[29] sind eine Möglichkeit, kritische Regionen zu implementieren. Man kann sich eine Semaphore am besten als eine Schachtel mit einer Kugel drin vorstellen[30]. Zu jeder gemeinsamen Ressource, zu der der Zugang geregelt werden soll, gehört so eine Schachtel mit Kugel (d. h. in unserem Fall typischerweise pro Shared-Memory-Segment eine), und es darf nur derjenige auf die gemeinsame Ressource zugreifen, der gerade die Kugel hat.

- Wer eine kritische Region betreten will, muß zuerst aus der dazugehörigen Schachtel die Kugel rausnehmen.
- Wer eine kritische Region verläßt, legt die Kugel, die er beim Betreten an sich genommen hat, wieder in die dazugehörige Schachtel zurück.
- Wer das Pech hat, eine kritische Region betreten zu wollen, in deren Schachtel im Moment keine Kugel drin ist (weil gerade jemand mit der Kugel unterwegs ist), muß so lange beim Eingang warten, bis derjenige, der die Kugel gerade hat, diese wieder zurücklegt.

---

[29] Zur Terminologie: Laut Duden heißt es "das Semaphor", in der Literatur liest man meist "der Semaphor", und an der örtlichen Universität wurde im mündlichen Sprachgebrauch meist "die Semaphore" verwendet, woran ich mich auch hier halten werde.

[30] Im allgemeinen Fall: Ein oder mehrere Schachteln mit jeweils ein oder mehreren Kugeln drin.

Semaphore technisch:

Eine Semaphore besteht aus einer Zähl-Variable im Betriebssystem-Kernel, die beliebige ganzzahlige Werte `>=0`, aber niemals negative Werte, annehmen kann[31] (wir beschränken uns vorläufig auf die Werte `0` und `1`). Auf diese Variable kann man nicht direkt zugreifen, sondern nur über Systemaufrufe:

- Eine Funktion initialisiert die Variable auf einen bestimmten Wert (in unserem Fall `1`).

  Diese Funktion wird nur einmal (unmittelbar nach dem Neuanlegen der Semaphore) aufgerufen. Man muß sich selbst darum kümmern, daß man für jede Semaphore die Initialisierungsfunktion aufruft, bevor das erste Mal ein Prozeß versucht, sie rauf- oder runterzuzählen[32]!

- Eine weitere Funktion zählt die Variable um einen bestimmten Wert (bei uns wieder `1`) herunter. Ist der aktuelle Wert der Variable kleiner als der angegebene (d. h. will man `1` runterzählen, obwohl die Variable schon `0` ist), so wird der aufrufende Prozeß automatisch so lange angehalten, bis andere Prozesse die Variable weit genug raufgezählt haben.

  Mit anderen Worten: Die Runterzähl-Funktion kehrt erst dann zurück, wenn das Runterzählen durchgeführt werden konnte, ohne daß der Wert der Semaphore ins Negative rutscht[33].

  Diese Runterzähl-Funktion ruft man jedesmal beim Betreten einer kritischen Region auf (d. h. der Aufruf steht im Programm unmittelbar vor jenem Code, der auf Daten im Shared-Memory-Segment zugreift).

- Die letzte Funktion erhöht die Variable um den angegebenen Wert (hier `1`).

  Diese Funktion kehrt immer sofort zurück. Falls andere Prozesse die Variable inzwischen herunterzählen wollten und angehalten wurden, werden diese Prozesse als Nebeneffekt des Raufzählens automatisch wieder aufgeweckt.

  Die Raufzähl-Funktion muß jedesmal beim Verlassen einer kritischen Region aufgerufen werden (d. h. der Aufruf steht im Programm unmittelbar nach jenem Codestück, das Daten im Shared-Memory-Segment bearbeitet).

Betreten der kritischen Region =
Rausnehmen der Kugel =
Runterzählen der Semaphore =
Belegen der Semaphore / der Ressource

Verlassen der kritischen Region =
Zurücklegen der Kugel =
Raufzählen der Semaphore =
Freigeben der Semaphore / der Ressource

Achtung:
Damit das Ganze wie gewünscht funktioniert, müssen die Funktionen in jedem beteiligten

[31] Im allgemeinen Fall besteht eine Semaphore aus ein oder mehreren solchen Variablen.

[32] Üblicherweise kann man das erreichen, wenn man die Semaphore anlegt und initialisiert, bevor man das Shared-Memory-Segment anlegt.

[33] Zu diesem Zweck verwaltet das Betriebssystem für jede Semaphore eine Queue wartender Prozesse.

Programm immer ***paarweise*** aufgerufen werden: Zu jedem Runterzählen muß es einen dazugehörigen Raufzähl-Aufruf geben! Es darf also beispielsweise nicht passieren, daß man aus einer kritischen Region an der Raufzähl-Funktion vorbei herausspringt!

Wird nach einem Runterzählen auf das Raufzählen vergessen, hängen meist früher oder später alle Prozesse endlos an der betreffenden Semaphore. Wird hingegen einmal zuviel raufgezählt, verliert die Semaphore ihre Schutzwirkung: Das Programm scheint zwar normal zu funktionieren, aber alle Prozesse können wild durcheinander auf das Shared Memory Segment zugreifen!

Weiters sollte der Code innerhalb der Semaphor-Aufrufe laufzeitmäßig so kurz wie möglich sein: Nachdem andere Prozesse ja möglicherweise darauf warten, daß der Prozeß mit dem Code in der kritischen Region fertig wird und diese wieder verläßt, sollte er dort wirklich nur Operationen auf gemeinsamen Ressourcen ausführen und nichts, was er genausogut ohne den Schutz einer Semaphore machen könnte. Es sollte vor allem vermieden werden, Benutzereingaben, Netzwerkzugriffe, `sleep` und ähnliche Operationen mit längerer oder gar unvorhersagbarer Dauer zu machen, während man eine Semaphore belegt hat.

Das "undo"-Problem:

Wenn ein Prozeß endet oder abstürzt, solange er sich innerhalb einer kritischen Region befindet (d. h. eine Semaphore heruntergezählt hat), hätte das zur Folge, daß diese Semaphore nie wieder raufgezählt und damit diese kritische Region nie wieder betretbar wird.

Als Konsequenz würden früher oder später alle auf diese kritische Region zugreifenden Programme ewig im Runterzählen der Semaphore hängenbleiben.

Um das zu verhindern, führt bei SysV IPC Semaphoren der Kernel auf Wunsch für jeden Prozeß Buch, um wieviel er jede Semaphore runter- und raufgezählt hat. Endet der Prozeß, obwohl er noch Raufzähl-Operationen ausständig hat (d. h. auf irgendeiner Semaphore bisher insgesamt mehr runter- als raufgezählt hat), so holt der Kernel automatisch die noch fehlenden Raufzähl-Operationen nach.

Bietet ein Betriebssystem diesen Mechanismus nicht, muß man selbst dafür sorgen, daß bei jedem Programmende oder -abbruch alle gerade von diesem Programm runtergezählten Semaphore wieder raufgezählt werden!

Warum kann man Semaphore nicht selbst programmieren?

Auf den ersten Blick sieht es so aus, als ob man Semaphore ganz einfach mit einer `int`-Variable in einem Shared-Memory-Segment selbst nachbauen könnte: Prüfen ob `0`, runterzählen, und wieder raufzählen wären kein Problem. Das scheitert aber aus zwei Gründen:

- *Das Prüfen und Runterzählen von Semaphoren muß* ***atomar*** *und* ***serialisiert*** *erfolgen:* Das Prüfen und Runterzählen muß eine einzige, unteilbare Operation sein (d. h. sie darf nicht durch Prozeßwechsel o. ä. unterbrochen oder in zwei Einzeloperationen zerteilt werden), und während dieser Operation darf kein anderer Prozeß (beispielsweise auf einer Mehrprozessor-Maschine) auf die betreffende Semaphore zugreifen können.

  Diese Forderungen lassen sich nicht in einem Anwenderprogramm, sondern nur innerhalb des Betriebssystems und selbst da nur mit Hardware-Unterstützung erfüllen: Jeder multiprozessor-taugliche Prozessor hat spezielle Instruktionen zur Implementierung von Semaphoren und ähnlichen Konstrukten. Diese Instruktionen bewirken, daß während der zusammengehörenden Speicherzugriffe für alle anderen Prozessoren der gleichzeitige Zugriff auf den Hauptspeicher hardwaremäßig blockiert wird.

- *Semaphore verwenden per Definition eine Prozeßverwaltung* ***ohne "busy waiting"****:* Zu jeder Semaphore verwaltet das Betriebssystem intern eine Warteschlange von Prozessen, die die Semaphore runterzählen wollten, obwohl sie schon auf `0` war: Solche Prozesse werden automatisch in einen inaktiven Zustand versetzt und auch automatisch wieder gestartet, wenn die Semaphore wieder raufgezählt wird. Solange sie warten, verbrauchen diese Prozesse ***keine*** CPU (d. h. das Warten auf eine Semaphore darf *nicht* dadurch implementiert werden, daß man ihren Wert in periodischen Abständen prüft!).

15 Semaphore, Funktionsaufrufe

Anlegen, Initialisieren und Löschen: Siehe Kapitel IPC-Grundlagen:

Das Neuanlegen einer Semaphore geschieht beispielsweise mit

```
sem_id=semget(sem_key, 1, sem_perm | IPC_CREAT | IPC_EXCL);
```

wobei `sem_key` der Key der Semaphore ist und `sem_perm` entsprechend definierte Zugriffsrechte (aus den vordefinierten Konstanten `SEM_R` und `SEM_A`[34]). In `sem_id` wird der Identifier der Semaphore (oder `-1` bei Fehler) gespeichert. Die Anzahl der Semaphor-Zähler wird beim Anlegen der Semaphore als zweites Argument übergeben, in diesem Beispiel `1`.

Die Initialisierung erfolgt (wie gesagt am besten unmittelbar danach und *vor* dem Anlegen des Shared-Memory-Segmentes, das von der Semaphore bewacht wird[35]) mit

```
union semun semctl_arg;
semctl_arg.val=1; semctl(sem_id, 0, SETVAL, semctl_arg);
```

Der Returnwert ist `0` oder `-1`. Der Wert, mit dem die Semaphore initialisiert werden soll, wird in `semctl_arg.val` übergeben[36] (in diesem Fall `1`). Das zweite Argument (hier `0`) ist der Index des zu initialisierenden Semaphor-Zählers: Enthält die Semaphore mehrere, so ist `semctl SETVAL` entsprechend oft mit `0`, `1`, ... aufzurufen (es gibt auch einen `semctl`-Aufruf zum Initialisieren aller Zähler, siehe `man`-Page).

Achtung:
Nach der Initialisierung sollte man eine Semaphore *nicht mehr* mit `semctl SETVAL` verändern:

- `semctl SETVAL` weckt auf die Semaphore wartende Prozesse *nicht* auf, auch wenn die Semaphore auf einen Wert größer `0` gesetzt wird.
- `semctl SETVAL` löscht alle Undo-Informationen für die Semaphore!

Auf eine bereits angelegte Semaphore kann ein Programm zum Beispiel wie folgt zugreifen (wie bei Shared-Memory-Segmenten müssen nur beim Neuanlegen Permissions angegeben werden):

---

[34] In Linux sind diese Konstanten nicht vordefiniert. Man kann aber die entsprechenden Konstanten für Shared-Memory-Segmente (`SHM_R` und `SHM_W`) verwenden: Sie haben dieselben numerischen Werte. Zur Not kann man die Permissions auch numerisch als Oktalzahl angeben, beispielsweise `0644`.

[35] Unmittelbar nach dem Initialisieren sollte der Prozeß, der die Semaphore und das Shared-Memory-Segment anlegt, die Semaphore runterzählen und erst dann wieder raufzählen, wenn er mit allen Initialisierungen fertig ist (damit kein anderer Prozeß auf das Shared-Memory-Segment zugreifen kann, bevor es fertig initialisiert ist).

[36] Der Typ `union semun` ist nur auf wenigen Unix-Systemen vordefiniert, in Linux beispielsweise ist seine Definition in `sem.h` zwar vorhanden, aber auskommentiert. Nachdem das vierte Argument von `semctl` im Prototyp meist mit "`...`" deklariert ist, findet für dieses Argument ohnehin keine Typprüfung statt, und man kann Werte beliebigen Typs ohne Union übergeben (d. h. einfach `1` direkt als Argument angeben).

```
sem_id=semget(sem_key, 1, 0);
```

(in diesem Fall initialisiert man die Semaphore natürlich nachher nicht mehr!), das Löschen dieser Semaphore geschieht mit `semctl` (Returnwert wieder `0` oder `-1`):

```
semctl(sem_id, 0, IPC_RMID, NULL);
```

Rauf- und Runterzählen: Für das Rauf- und Runterzählen gibt es eine einzige Operation `semop`. Diese ist viel komplizierter, als wir sie brauchen, weil sie auch für Semaphor-Mengen mit mehreren Semaphor-Zählern gedacht ist und daher in einem einzigen Aufruf mehrere Zähler nach Wunsch rauf- oder runterzählen können muß:

`int semop(int` *sem_id*`, struct sembuf` *semop_array*`[], size_t` *n_ops*`)`

sem_id ist der Identifier der Semaphore, auf die die Operationen ausgeführt werden sollen, *semop_array* ist ein variabel großes Array von Strukturen, wobei jede Struktur, d. h. jedes Element des Arrays, eine Rauf- oder Runterzähloperation definiert, und *n_ops* ist die Anzahl der Operationen, d. h. die Größe des Arrays *semop_array*.

Der Returnwert ist `0` bei Erfolg und `-1` bei Fehler. Es werden immer *alle* oder *keine* der angegebenen Operationen ausgeführt (und zwar atomar, d. h. ohne daß dazwischen ein anderer Prozeß auf die Semaphore zugreifen kann), es kann bei einem Fehler also nicht passieren, daß ein paar der angegebenen Operationen durchgeführt wurden und ein paar nicht.

Achtung:
Wenn `semop` beim Runterzählen einer Semaphore warten muß, und es trifft während des Wartens ein Signal für den Prozeß ein, so kehrt `semop` unverrichteter Dinge mit dem Fehler `EINTR` zurück!

Die Struktur zur Beschreibung einer einzelnen Operation ist wie folgt vordefiniert:

```
struct sembuf {
  ushort sem_num;  /* index of counter */
  short sem_op;    /* increment / decrement value */
  short sem_flg;}  /* flags */
```

`sem_num` ist der Index des Semaphor-Zählers, auf den die Operation anzuwenden ist, innerhalb der Semaphor-Menge (bei nur einem Zähler daher `0`). `sem_op` ist der Wert, um den der Zähler raufgezählt (wenn `sem_op` größer `0` ist) oder runtergezählt (wenn `sem_op` kleiner `0` ist) werden soll, also normalerweise `-1` zum Belegen einer Semaphore und `1` zum Freigeben[37]. `sem_flg` setzt man auf das bitweise Oder der für die Operation zu verwendenden Flags, oder auf `0`. Für uns ist derzeit nur das Flag `SEM_UNDO` wichtig: Ist es angegeben, berücksichtigt der Kernel diese Operation im Undo-Zähler dieses Prozesses für diese Semaphore, sonst nicht.

Im einfachsten Fall macht man Operationen auf eine einzige Semaphore also wie folgt:

```
struct sembuf enter[1]={{0, -1, SEM_UNDO}};
struct sembuf leave[1]={{0, 1, SEM_UNDO}};

  if (semop(sem_id, enter, 1)==-1) { /* Fehler */ }
  /* kritische Region */
  if (semop(sem_id, leave, 1)==-1) { /* Fehler */ }
```

[37] Der Wert `0` für `sem_op` hat eine Sonderfunktion, die wir später erwähnen.

Limits: Semaphore werden im Kernel typischerweise als `short int` gespeichert, daher ist der maximale Wert einer Semaphore (`SEMVMX`) normalerweise `32767`. Daneben gibt es noch Limits für die Anzahl der Semaphor-Mengen systemweit (`SEMMNI`, zumindest `10`), die Anzahl der Semaphor-Zähler systemweit (`SEMMNS`, zumindest `50`), die Anzahl der Semaphor-Zähler pro Semaphor-Menge (`SEMMSL`, zumindest `20`), und die maximale Anzahl der Rauf- und Runterzähl-Operationen, die ein einziger `semop`-Aufruf ausführen kann (`SEMOPN`, zumindest `10`). Weiters gibt es noch eigene Limits für den Undo-Mechanismus.

16 Semaphore, Deadlocks

Das Problem:

Ein Deadlock ist die zyklische Verklemmung von Prozessen (klassisches Beispiel: "Dining Philosophers"): Jeder der beteiligten Prozesse wartet auf ein Ereignis (in unserem Fall darauf, daß ein anderer Prozeß die gewünschte Semaphore freigibt), das nie eintritt, weil der Prozeß, der für das Eintreten des Ereignisses (in unserem Fall das Freigeben der gewünschten Semaphore) zuständig wäre, selbst ewig auf ein anderes Ereignis (in unserem Fall das Freigeben einer anderen gerade belegten Semaphore) wartet.

Der einfachste Fall sieht wie folgt aus:

- Prozeß `A` hat Semaphore `y` belegt und möchte zusätzlich noch Semaphore `x` belegen.
- Prozeß `B` hat Semaphore `x` belegt und möchte zusätzlich noch Semaphore `y` belegen.
- Als Folge wartet `A` auf `B` (wegen `x`) und `B` auf `A` (wegen `y`).

Nach oben hin ist der Anzahl der beteiligten Prozesse und Semaphoren keine Grenze gesetzt, wesentliches Kennzeichen eines Deadlocks ist, daß

- jeder beteiligte Prozeß mindestens eine Semaphore belegt hat und zumindest eine weitere Semaphore belegen möchte,
- und daß eine Folge von Abhängigkeiten besteht (`A` wartet wegen `x` auf `B`, `B` wartet wegen `y` auf `C`, `C` wartet wegen `z` auf `D` usw.), die zyklisch ist, d. h. irgendwann zu einem Prozeß führt, der seinerseits wieder auf den ursprünglichen Prozeß `A` wartet[38].

Die beteiligten Prozesse hängen alle so lange, bis man einen davon manuell abbricht.

Der banalste und peinlichste Fall eines Deadlocks (mit nur einem Prozeß und nur einer Semaphore) tritt ein, wenn ein Prozeß aus irgendeinem Grund das Raufzählen einer Semaphore vergessen hat und eine Semaphore, die er ohnehin gerade selbst belegt, nochmal runterzählen möchte: Der Prozeß hängt endlos an sich selbst!

Die Lösung:

---

[38] Das gleiche Problem kann es bei Datenbanken geben, wenn gleichzeitig mehrere Transaktionen mit jeweils mehreren Datenbankzugriffen auf dieselben Daten gemacht werden. Beispiel:

* Benutzer `A` hat Datensatz `y` geändert (und damit bis zum Ende seiner Transaktion exklusiv für sich gesperrt) und möchte in der gleichen Transaktion auch noch Datensatz `x` ändern.
* Benutzer `B` hat in seiner Transaktion bereits Datensatz `x` geändert und möchte jetzt noch Datensatz `y` ändern.

- Wenn das Programm nur eine einzige Semaphore verwendet, oder wenn es zwar mehrere Semaphoren gibt, aber jeder Prozeß zu jedem Zeitpunkt maximal eine Semaphore belegt, können keine Deadlocks auftreten.

  Mit anderen Worten: Wenn man immer darauf achtet, die gerade belegte Semaphore freizugeben, bevor man eine andere Semaphore belegt, ist man auf der sicheren Seite.

- Muß man von der Programmlogik her an einer Stelle im Programm wirklich mehrere Semaphoren gleichzeitig belegen, hilft die Möglichkeit von SysV Semaphoren, eine Semaphor-Menge mit mehreren Semaphor-Zählern anzulegen:

  * Anstatt mehrerer getrennter Semaphoren legt man eine einzige mit der benötigten Anzahl von Zählern an.
  * Dann zählt man immer alle Semaphoren, die man an einer bestimmten Stelle im Programm braucht, mit einem *einzigen* `semop`-Aufruf gleichzeitig runter (für das Raufzählen kann man ein oder mehrere Aufrufe verwenden)[39].

  Solange man nie zwei getrennte Runterzähl-Aufrufe nacheinander macht (ohne vorher wieder alle Semaphoren freigegeben zu haben), können ebenfalls keine Deadlocks auftreten.

- Möchte oder muß man mehrere getrennte Semaphoren mit jeweils nur einem Zähler verwenden, gibt es folgenden Weg, die Deadlock-Freiheit eines Programmsystems zweifelsfrei festzustellen:

  * Wenn es eine Möglichkeit gibt, die Semaphoren so zu numerieren, daß bei jedem Runterzählen einer Semaphore mit der Nummer i vom betreffenden Prozeß entweder gar keine anderen Semaphoren oder nur Semaphoren mit Nummern kleiner i (aber keine mit einer Nummer größer als i!) belegt sind, so ist das Programmsystem deadlockfrei.
  * Läßt sich keine solche Numerierung der Semaphoren finden, muß man mit Deadlocks rechnen[40].

Wenn man mit Deadlocks rechnen muß, bleibt als letzte Möglichkeit, mit `alarm` selbst Timeouts zu implementieren und einen der Wartenden abzubrechen (weil ein wartendes `semop` ja beim Eintreffen eines Signal mit `EINTR` endet).

17 Semaphore, Sonderfälle

Weitere Operationen:

SysV Semaphore bieten viel mehr Operationen, als für die Bewachung von Shared-Memory-Segmenten eigentlich notwendig wären:

- **Auslesen des aktuellen Wertes:**

[39] Ähnliches gilt für Datenbanken: Dort darf man nie nacheinander in der gleichen Transaktion ein Shared-Lock und später ein Exclusive-Lock auf die gleiche Tabelle anfordern: Wenn man einen Wert liest, den man später ändern will, muß man mit einem `SELECT FOR UPDATE` gleich beim Lesen das Exclusive-Lock anfordern!

[40] Genau die gleiche Idee funktioniert für die Deadlockfreiheit gleichzeitiger Datenbank-Transaktionen: Wenn ich eine Numerierung für meine Tabellen und meine Zeilen innerhalb der Tabellen finden kann, sodaß jeder Datenbank-Zugriff einer Transaktion auf eine Tabelle / Zeile mit höherer Nummer zugreift als alle vorherigen Zugriffe in derselben Transaktion, so sind die Transaktionen untereinander theoretisch deadlockfrei.

Mit `semctl(`*sem_id*`, `*sem_index*`, GETVAL, 0)` kann man den aktuellen Zählerstand des Zählers *sem_index* der Semaphore *sem_id* auslesen. Das sollte man zwar nicht in der Programmlogik verwenden (weil der Wert unmittelbar darauf ja schon wieder ein ganz anderer sein kann), aber zur Fehlersuche ist es hilfreich: Wenn man versehentlich mehr runterzählt als raufzählt, merkt man das üblicherweise recht schnell (weil das Programm meist hängenbleibt). Zählt man hingegen öfter hinauf als herunter, fällt das normalerweise überhaupt nicht auf, außer durch gelegentliche mysteriöse Dateninkonsistenzen. Hier hilft nur, den Semaphoren-Wert auszulesen und zu prüfen, ob er vielleicht zu groß ist: Ist der Wert innerhalb der kritischen Region größer als `0` (oder jemals größer als `1`), so stimmt etwas nicht ... [41]

Mit `semctl GETNCNT` (bzw. `GETZCNT` für `0`-Warter) kann man erfragen, wie viele Prozesse gerade auf eine Semaphore warten.

- **Warten, bis die Semaphore `0` wird:**

  Gibt man bei `semop` statt eines Rauf- oder Runterzählwertes `0` an, so kehrt der `semop`-Aufruf erst dann zurück, wenn der Semaphoren-Zähler `0` ist oder wird. Der Semaphoren-Zähler selbst wird von einem solchen `semop`-Aufruf mit `0` nicht verändert.

  Damit läßt sich beispielsweise ein Prozeß programmieren, der genau dann aktiv wird (d. h. nach dem `semop`-Aufruf weitermacht), wenn irgendein anderer Prozeß eine kritische Region betritt oder (wenn man eine Semaphore wie unten beschrieben als "Füllstandsanzeige" für eine Queue-artige Datenstruktur einsetzt) wenn die Datenstruktur voll oder leer ist.

- **`semop`-Aufrufe ohne Warten:**

  Es gibt zwar keine Möglichkeit, ein Timeout für die maximale Wartedauer einer `semop`-Operation anzugeben[42], aber wenn bei den Flags einer Operation (im Feld `sem_flg` von `struct sembuf`, wo man u. a. auch `SEM_UNDO` angibt) `IPC_NOWAIT` gesetzt ist, so kehrt der Aufruf sofort (mit Returnwert `-1` und `errno` gleich `EAGAIN`) zurück, wenn die gewünschte Runterzähl-Operation nicht sofort möglich ist, sondern blockieren würde.

  Will man beispielsweise zu Debug- oder Performancetuning-Zwecken wissen, ob eine Semaphoren-Operation sofort klappte oder warten mußte (was man ja normalerweise nicht unterscheiden kann), kann man wie folgt vorgehen: Man macht die Operation zuerst einmal mit `IPC_NOWAIT`. Liefert das keinen Fehler, hat die Operation sofort (ohne Warten) funktioniert. Liefert es hingegen `EAGAIN` (alle anderen Fehler sind wie sonst zu behandeln), hätte die Operation warten müssen. Man kann eine entsprechende Meldung ausgeben, mitzählen oder was immer, und ruft dieselbe Operation dann nochmals — diesmal normal, d. h. ohne `IPC_NOWAIT` — auf.

Weitere Anwendungen:

Im Laufe der Zeit wurden zahlreiche mehr oder weniger trickreiche Einsatzmöglichkeiten für Semaphore gefunden. Zwei Beispiele (von vielen):

- **Shared-Memory-Segmente mit "kritischen" und "unkritischen" Operationen:**

  Auf Shared-Memory-Segmenten gibt es neben den "kritischen" Operationen (während denen sonst niemand auf das Segment zugreifen können darf) oft auch "unkritische"

[41] Ein anderer Trick zur Fehlersuche in Programmen mit Semaphoren besteht darin, kritische Regionen mit `sleep` künstlich zu verlängern und dann zu prüfen, ob die anderen Prozesse wie erwartet dementsprechend lange blockiert werden.

[42] Das muß man mit `alarm` händisch ausprogrammieren: `semop` kehrt nach einem Signal mit `EINTR` zurück.

Zugriffe, die gefahrlos von mehreren gleichzeitig ausgeführt werden können, und bei denen "gewöhnliche" Semaphore das Programm nur unnötig verzögern.

Ganz weglassen kann man die Semaphor-Aufrufe um diese "unkritischen" Zugriffe aber auch nicht, denn während einer "kritischen" Operation sind auch keine gleichzeitigen "unkritischen" Operationen erlaubt.

Typischerweise sind Schreiboperationen auf Shared-Memory-Segmenten immer kritisch (es dürfen gleichzeitig weder andere Schreib- noch Lese-Operationen erfolgen), reine Lesezugriffe hingegen unkritisch (solange keiner etwas ändert, dürfen beliebig viele gleichzeitig lesen)[43].

Realisiert wird das wie folgt:

* Die Semaphore wird nicht auf `1`, sondern auf einen sehr hohen Wert initialisiert, beispielsweise `1000`.
* Vor einer "unkritischen" Operation (d. h. einer nur lesenden kritischen Region im Code) wird die Semaphore wie gewohnt um `1` runtergezählt, und nachher wieder um `1` erhöht.
* Vor einer "kritischen" Operation (kritische Region mit Schreibzugriffen) wird sie hingegen um den Initialwert `1000` erniedrigt, und danach wird auch wieder um `1000` raufgezählt.

Das hat genau den gewünschten Effekt:

* Ein Schreibzugriff muß so lange warten, bis alle kritischen und unkritischen Operationen fertig sind (denn das Runterzählen um `1000` blockiert so lange, bis alle anderen wieder raufgezählt haben).
* Während ein Schreibzugriff läuft, müssen alle anderen Operationen, auch die Lesezugriffe, warten (denn wenn man `1000` runtergezählt hat, steht die Semaphore auf `0`, und jeder weitere Versuch, `1` oder `1000` runterzuzählen, wird blockiert).
* Ist hingegen kein Schreibzugriff aktiv, sind viele gleichzeitige Lesezugriffe möglich (maximal `1000`): Solange nicht tausendmal um `1` runtergezählt wurde, ist die Semaphore größer `0`, und das Runterzählen um `1` ist ohne Blockieren möglich.

Achtung: Logisch zusammenhängende Lese- und Schreibzugriffe (beispielsweise Lesen, Neuberechnen und Zurückschreiben eines Wertes) sind als eine einzige, kritische Operation zu betrachten, d. h. auch der Lesezugriff muß schon innerhalb des mittels Semaphoren für den Schreibzugriff exklusiv gesperrten Codebereiches liegen!

Achtung: Dieser Trick ist nur eine Notlösung: Er bevorzugt Leser und benachteiligt Schreiber, bei hohem Leser-Andrang "verhungern" die Schreiber!

- **Semaphore für Queues in einem Shared-Memory-Segment:**

  Shared-Memory-Segmente wären an sich auch ein sehr effizienter Weg, Daten paket- oder elementweise von einem Prozeß zu einem anderen zu schicken (anstelle von Message Queues oder Pipes): Der sendende Prozeß fügt die Datenpakete beispielsweise in ein zyklisches Array oder eine verkettete Liste im Shared-Memory-Segment ein, der empfangende Prozeß entnimmt sie.

  Offen bleibt allerdings die Frage der Synchronisation: Wie erfährt der empfangende Prozeß, daß der Sender Daten für ihn im Shared-Memory-Segment abgelegt hat?

[43] Ähnlich den "Shared Locks" und "Exclusive Locks" auf Datenbanken.

Hier hilft eine Semaphore, deren Zählerstand (auf `0` statt `1` initialisiert) stets der im Shared-Memory-Segment vom Sender abgelegten, aber vom Empfänger noch nicht entnommenen Datenpakete entspricht: Der Sender zählt die Semaphore jedesmal um eins rauf (aber nie runter), *nachdem* er ein Element im Shared-Memory-Segment gespeichert hat, und der Empfänger zählt die Semaphore jedesmal um eins runter (aber nie rauf), *bevor* er ein Element aus dem gemeinsamen Speicher entnimmt. Die Semaphore sorgt automatisch dafür, daß der Empfänger blockiert wird, wenn das Shared-Memory-Segment keine Elemente für ihn enthält, und wieder gestartet wird, nachdem der Sender Daten bereitgestellt hat.

Für die Steuerung des sendenden Prozesses (wie halte ich den Sender an, wenn die Datenstruktur voll ist und keine weiteren Elemente gespeichert werden können, und wie wecke ich ihn wieder auf, wenn der Empfänger Elemente entnommen hat und wieder Platz ist?) verwendet man analog dazu eine zweite Semaphore, deren Zählerstand stets der aktuellen Anzahl der freien Speicherplätze im Shared-Memory-Segment entspricht: Initialisiert wird sie auf die Anzahl der Plätze in der Datenstruktur, der Sender zählt sie jedesmal eins runter, *bevor* er ein neues Element im Shared-Memory-Segment ablegt, und der Empfänger zählt sie eins rauf, *nachdem* er ein Paket gelesen hat und daher wieder ein Platz freigeworden ist.

Achtung: In diesem Fall müssen die Rauf- und Runterzähl-Operationen alle *ohne* automatisches Undo (d. h. die `semop`-Operationen ***ohne*** das Flag `SEM_UNDO`) gemacht werden! Das Betriebssystem würde sonst beim Beenden des Empfängerprozesses so viele Raufzähl-Operationen auf der Semaphore nachholen, wie der Empfänger während seiner gesamten Laufzeit Pakete empfangen hat. Dann würde der Stand der Semaphore nicht mehr der Anzahl der derzeit im Shared Memory abgelegten Datenelemente entsprechen, und die ganze Programmlogik würde nicht mehr wirken (für die andere Semaphore gilt spiegelbildlich dasselbe).

Diese Synchronisationsmethode funktioniert sogar für mehrere Sender und/oder mehrere Empfänger, die sich eine einzige solche "Queue" in einem Shared-Memory-Segment teilen: Sobald irgendein Sender ein Datenpaket abgelegt hat, wird irgendein gerade wartender Empfänger aufgeweckt, um es zu verarbeiten.

Während man bei nur je einem Sender und Empfänger aber in vielen Fällen ohne zusätzliche "normale" Semaphore für die Bewachung der Zugriffe auf das Shared-Memory-Segment auskommt (weil der Sender und der Empfänger üblicherweise ohnehin nie gleichzeitig auf dasselbe Datenelement zugreifen können und daher keine Gefahr von Inkonsistenzen besteht), braucht man bei mehreren Sendern oder Empfängern zusätzlich zu den beiden Semaphoren, die die vollen und leeren Elemente zählen, weitere Semaphoren für die kritischen Regionen mit den Zugriffen auf das Segment, damit nicht mehrere Sender oder Empfänger durch das gleichzeitige Einfügen oder Entnehmen desselben Elementes Dateninkonsistenzen hervorrufen.

18 Message Queues

Konzept:

Message Queues sind (ähnlich Pipes und FIFO's, Streams, und Netzwerkverbindungen) ein Mechanismus im Kernel, um Daten von einem Prozeß zu einem anderen zu schicken.

Message Queues transportieren im Unterschied zu den oben genannten Mechanismen aber keinen Strom von Bytes, sondern einzelne Messages, Datenblöcke bestimmter Größe. Jeder Sende-Aufruf legt eine Message in der Queue ab, und jeder Empfangs-Aufruf entnimmt eine Message aus der Queue. Dabei können beliebig viele Prozesse Messages in eine einzige Queue senden oder von ihr empfangen.

Anlegen und Löschen:

Wie im Übersichtskapitel beschrieben, erfolgt das Anlegen einer Message Queue (bzw. das Ermitteln des Identifiers einer bestehenden Queue mit gegebenem Key) mit
`int msgget(key_t` *key*`, int` *flags*`)`.

Ergebnis ist der Identifier oder `-1` bei Fehler. In *flags* werden die Permissions und `IPC_CREAT` sowie `IPC_EXCL` übergeben. Für die Permissions gibt es normalerweise die Konstanten `MSG_R` und `MSG_W`, unter Linux fehlen diese aber (wie schon bei den Semaphoren).

Das Löschen der Message Queue mit dem Identifier *id* erfolgt mit `msgctl(`*id*`, IPC_RMID, NULL)` (Returnwert wieder `0` oder `-1`).

Senden und Empfangen:

`int msgsnd(int` *id*`, const void *`*msg_ptr*`, size_t` *msg_len*`, int` *flags*`)`

id ist der Identifier der Message Queue, in die gesendet werden soll, *msg_ptr* ist ein Zeiger auf die zu sendende Message, *msg_len* ist deren Länge, und *flags* ist `0` oder `IPC_NOWAIT`: Wenn eine Message wegen eines Ressourcen-Engpasses nicht gesendet werden kann (die maximale Anzahl der in einer Queue zwischengespeicherten Bytes oder die maximale Anzahl der Messages systemweit würden überschritten werden), so wartet `msgsnd` normalerweise, bis wieder Platz ist. Gibt man `IPC_NOWAIT` an, kehrt `msgsnd` in diesem Fall sofort mit dem Fehler `EAGAIN` zurück.

Der Returnwert ist `0` bei Erfolg und `-1` bei Fehler.

`int msgrcv(int` *id*`, void *`*msg_ptr*`, size_t` *max_len*`, long` *msg_type*`, int` *flags*`)`

id ist der Identifier der Message Queue, aus der empfangen werden soll, *msg_ptr* ist ein Zeiger auf den Speicherplatz, in den die empfangene Message gespeichert werden soll, *max_len* ist die maximale Länge, die die zu empfangende Message haben darf (d. h. die Größe des Platzes, auf den *msg_ptr* zeigt), *msg_type* ist der Typ der zu empfangenden Message (siehe unten), und *flags* ist `0` oder `IPC_NOWAIT`: `msgrcv` wartet normalerweise, wenn in der Queue gerade keine Message (bzw. keine Message passenden Typs) vorhanden ist, bis eine gesendet wird. `IPC_NOWAIT` bewirkt in diesem Fall, daß `msgrcv` sofort mit dem Fehler `ENOMSG` (nicht `EAGAIN`!) zurückkehrt.

Der Returnwert ist die tatsächliche Länge der empfangenen Message bei Erfolg und `-1` bei Fehler. Einer der möglichen Fehler ist `E2BIG`: Die zu empfangende Message wäre größer gewesen als in *max_len* angegeben[44].

Achtung:
Wenn ein Aufruf von `msgsnd` oder `msgrcv` warten muß und währenddessen ein Signal kommt, kehrt er unverrichteter Dinge mit `-1` und `errno` gleich `EINTR` zurück!

Aufbau von Messages:

[44] Außer man hat in *flags* `MSG_NOERROR` angegeben: Dann wird eine zu lange Message ohne Fehlermeldung auf *max_len* Bytes gekürzt und trotzdem empfangen. Die restlichen Bytes gehen verloren.

Eine Message ist vom Typ her eine Struktur. Das ***erste*** Feld ***muß*** vom Typ `long` sein, es enthält den Message-Typ (siehe unten). Der Rest der Struktur ist beliebig.

- Der Trivialfall ist die leere Message, die nur aus einem `long`-Wert für den Typ und sonst nichts besteht (weiter unten sehen wir, daß das durchaus sinnvoll sein kann).

  ```
  long msg_type;
  ```

  Für *msg_ptr* in `msgsnd` und `msgrcv` übergibt man dann einfach `&msg_type`.

- In vielen Fällen umfaßt eine Message außer dem Typ-Feld nur einen String, beispielsweise mit maximal `256` Zeichen.

  ```
  typedef struct {
    long msg_type;
    char msg_text[256];}
  msg_struct;

  msg_struct msg_buffer;
  ```

  Als *msg_ptr* würde man in diesem Fall `&msg_buffer` verwenden.

- Messages können aber auch beliebig strukturierte binäre Daten enthalten. Für Messages mit Informationen über einen Schultest könnte man z. B. folgenden Typ verwenden:

  ```
  typedef struct {
     time_t datum;
     char klasse[5];
     double schnitt;
     int noten[5];
     int gefehlt;}
  msg_datatype;

  typedef struct {
    long msg_type;
    msg_datatype msg_data;}
  msg_struct;
  ```

  Der *msg_ptr* im `msgsnd` und `msgrcv` würde dann wieder wie oben auf eine Variable vom Typ `msg_struct` zeigen.

  Werden Messages verschiedener Struktur und Länge verwendet, ist dieses Beispiel mit `union`-Typen entsprechend zu erweitern.

Achtung:
Die Länge des Typ-Feldes wird bei Längenangaben (*msg_len*, *max_len* und Returnwert von `msgrcv`) ***nicht*** mitgerechnet: Die Länge einer Message umfaßt nur die Anzahl der Bytes ***hinter*** dem Typ-Feld!

Eine leere Message hat daher Länge `0`, nicht `sizeof(long)`, obwohl sie aus einem `long`-Wert (dem Typ) besteht. Im String-Beispiel oben ist die Länge der Message `256`, im letzten Beispiel `sizeof(msg_datatype)`.

Der Message-Typ:

Jede Message hat einen Message-Typ. Dieser wird beim Senden in einem `long`-Wert am Anfang der Message angegeben und muß größer `0` sein (`msgsnd` liefert den Fehler `EINVAL`, wenn eine Message gesendet werden soll, die mit `0` oder einer negativen Zahl beginnt.).

Beim `msgrcv` gibt man in *msg_type* vor, welchen Message-Typ die zu empfangende Message haben soll:

- Wenn man für *msg_type* `0` angibt, wird die erste Message aus der Queue empfangen, unabhängig von deren Typ.
- Wenn man eine positive Zahl n angibt, wird die erste Message aus der Queue empfangen, deren Message-Typ gleich n ist[45]. In der Queue wartende Messages mit anderem Message-Typ werden ignoriert und verbleiben in der Queue.
- Wenn man eine negative Zahl $-n$ angibt, wird eine Message aus der Queue empfangen, deren Message-Typ kleinergleich n ist. Gibt es mehrere solche, werden diejenigen mit dem kleinsten Message-Typ gewählt, und davon wieder die erste.

Abgesehen von der Auswahl nach Message-Typ arbeiten Message Queues nach dem First-In-First-Out-Prinzip: Die Messages werden in der Reihenfolge des Sendens empfangen.

Der Message-Typ-Mechanismus hat drei Hauptanwendungen:

Messages an verschiedene Empfänger: Es ist möglich, über eine einzige Message-Queue gezielt Daten an einen von mehreren Empfänger zu senden: Jedem einzelnen Empfänger wird ein bestimmter Message-Typ zugeordnet, er macht sein `msgrcv` nur mit diesem Typ, und bekommt daher nur die für ihn bestimmten Messages aus der Queue (wenn der Sender die Messages bei Senden mit dem jeweiligen Typ versehen hat)[46].

Message-Prioritäten: Ein `msgrcv` mit negativem *msg_type* wählt die Message mit dem kleinsten passenden Message-Typ. Eine Message mit niedrigem Message-Typ "überholt" daher alle in der Queue wartenden Messages mit höherem Typ, auch wenn diese früher gesendet wurden.

Kodierung von Befehlen, Fehlern, ...: Man kann Befehle durch verschiedene Message-Typen kodieren, z. B. `1` = "schick mir die nächste Zeile Daten" oder `9999` = "ich melde mich ab". Ebenso kann man bestimmte Message-Typen für Fehlermeldungen reservieren, beispielsweise `10000` = "nichts gefunden" , `10001` = "Keine Berechtigung" usw..

---

[45] Bzw. die erste Message mit Message-Typ ***ungleich*** n, wenn man in *flags* `MSG_EXCEPT` angegeben hat.

[46] Diese Idee hat allerdings einen Haken: Wenn nur ein einziger Empfänger die an ihn gerichteten Messages nicht liest, läuft die Message Queue im Laufe der Zeit voll. Damit ist die Kommunikation mit allen Empfängern blockiert. Eine Methode, die dieses Problem nicht hat und trotzdem dynamisch viele, zu beliebigen Zeitpunkten individuell startende und endende Clients als Empfänger erlaubt, ist folgende:

- Es gibt eine einzige, permanente, vom Server angelegte Message Quere mit bekanntem Key für Messages von den Clients zum Server.
- Jeder Client / Empfänger legt seine eigene Message Queue für Messages vom Server zum Client an (am besten mit `IPC_PRIVATE`) und schickt den Identifier dieser Queue in jeder Message an den Server (über die "zum-Server-Queue") mit.
- An Hand dieses Identifiers weiß der Server, über welche Message Queue er die Antwort an den betreffenden Client zurückschicken muß, ohne daß er sich selbst um diese Antwort-Queue kümmern muß oder deren Key kennt.

Limits:

Neben einigen Anzahl-Beschränkungen (`MSGMNI`, maximale Anzahl der Message Queues systemweit, üblicherweise mindestens `50`, und `MSGTQL`, maximale Anzahl der Messages systemweit, typischerweise auch `50`) unterliegen Message Queues recht knappen Größen-Beschränkungen: `MSGMAX` ist die maximale Anzahl von Bytes, die eine einzelne Message haben darf, und `MSGMNB` ist die maximale Anzahl von Bytes, die eine einzelne Message-Queue zwischenspeichern kann (d. h. die Summe der Größen aller gesendeten, aber noch nicht empfangenen Messages in einer Queue). Auf vielen Unix-Systemen haben diese beiden Konstanten nur die Werte `2048` und `4096`, auf mehr sollte man sich nicht verlassen[47].

19 `mmap`

Idee:

`mmap` realisiert "Memory Mapped I/O", d. h. I/O durch Einblenden der File-Daten in den Hauptspeicher: Nach dem Aufruf von `mmap` sieht ein Programm den Inhalt eines Files in einem bestimmten Bereich seines Hauptspeichers, so, als ob dieser Teil seines Hauptspeichers ein Fenster auf jenes Stück der Platte wäre, das den betreffenden File enthält.

Man bekommt einen Pointer auf den Anfang dieses Hauptspeicher-Bereiches und kann diesen Pointer genauso verwenden wie beispielsweise einen Pointer auf den Anfang eines Shared-Memory-Segmentes, nämlich z. B. als Pointer auf eine große Struktur oder als ein `char`-Array, nur eben mit dem Unterschied, daß die betreffenden Bytes im Speicher direkt den entsprechenden Bytes im File entsprechen.

Anwendungen:

`mmap` hat vielfältige Anwendungsmöglichkeiten, die mit seiner ursprünglichen Funktion nur mehr zum Teil zu tun haben und oft auch in Kombination eingesetzt werden:

File-I/O: `mmap` ermöglicht File-I/O ohne `read` und `write`: Nach `mmap` kann man auf die im betreffenden File enthaltenen Daten durch normale Hauptspeicherzugriffe zugreifen, und auf Wunsch werden die Daten auch wieder in den File zurückgeschrieben, nachdem sie im Speicher verändert worden sind.

Shared Memory: `mmap` wird immer häufiger als Alternative zu SysV Shared-Memory-Segmenten eingesetzt[48]: Wenn mehrere Prozesse ein `mmap` auf denselben File machen, haben sie dieselben Daten in ihren Speicherbereich eingeblendet, also de facto nichts anderes als einen Speicherbereich, auf den mehrere Prozesse gleichzeitig zugreifen.

Diese Daten können, müssen aber nicht mit einem File in Verbindung stehen: Oft wird als File der Pseudo-File `/dev/zero` angegeben[49], wodurch man ein mit `0`-Bytes initialisiertes Shared Memory erhält.

---

[47] Message Queues sind daher nicht zum Austausch großer Datenblöcke geeignet, und sie puffern auch nicht mehr als als Pipes oder FIFO's: Wenn die Messages nicht schnell genug gelesen werden, werden weitere Sende-Aufrufe blockiert.

[48] Oft auch zusammen mit Dateisperren auf den betreffenden File anstatt der Semaphoren zur Verhinderung von Inkonsistenzen durch gleichzeitige Zugriffe.

[49] `/dev/zero` ist ähnlich `/dev/null` kein real existierender File, sondern wird vom Kernel simuliert: Beim Lesen liefert `/dev/zero` beliebig viele `0`-Bytes, und auf `/dev/zero` geschriebene Daten werden "entsorgt".

Der wesentliche Vorteil (außer, daß die Daten eben auch auf der Platte gespeichert werden und daher über Reboots hinweg erhalten bleiben) gegenüber SysV Shared-Memory-Segmenten ist, daß man `mmap`-Bereiche nicht explizit löschen muß: Endet ein Prozeß, werden auch alle seine `mmap`-Bereiche automatisch wieder freigegeben.

Dynamisch allokierter Speicher: `mmap` ist auch eine Möglichkeit, zum Hauptspeicher, der einem Prozeß zur Verfügung steht, dynamisch neue Bereiche hinzuzufügen, und zwar unabhängig von jenem Speicher, der von `malloc` und `free` verwaltet wird[50].

Aufruf:

`void *mmap(void *`*addr*`, size_t` *len*`, int` *prot*`, int` *flags*`, int` *fd*`, off_t` *offset*`)`

- In *addr* könnte man eine Adresse vorgeben, an der die Daten eingeblendet werden sollen. Aus den gleichen Gründen wie bei `shmat` sollte man hier aber immer `0` angeben, damit das System den Speicherbereich beliebig plazieren kann.
- *len* ist die Größe des `mmap`-Bereiches. Es ist leider nicht möglich, einen File automatisch in seiner vollen Länge einzublenden: Man muß die Größe des Files zuerst mit `fstat` ermitteln und dann hier angeben.

  Wenn *len* größer ist als die Länge des Files, so kehrt das `mmap` zwar ohne Fehler zurück, aber der erste Speicherzugriff auf eine Stelle im `mmap`-Bereich, die "hinter" dem Fileende liegt, löst das Signal `SIGBUS` aus (das gleiche passiert, wenn der File nach dem `mmap` geschrumpft ist). Es ist mit `mmap` also nicht möglich, Daten an einen File hinten anzuhängen[51].

  Es ist hingegen kein Problem, wenn *len* kleiner als die Länge des Files ist: Es wird eben nur ein Teil des Files eingeblendet.
- *prot* legt den Speicherschutz für den `mmap`-Bereich fest: `PROT_READ` für lesenden Zugriff und `PROT_WRITE`, wenn Schreiben erlaubt sein soll[52].

  Der beim `open` auf den dazugehörigen File angegebene Modus muß dabei zu den hier angegebenen Zugriffsarten passen, sonst kommt ein Fehler: Wenn man `PROT_READ | PROT_WRITE` angibt, muß auch der File Read/Write geöffnet worden sein.

  Mißachtet man den für einen `mmap`-Bereich festgelegten Speicherschutz (versucht also, in einen Bereich ohne `PROT_WRITE` zu schreiben), bekommt man das Signal `SIGSEGV`.
- Von den Konstanten für *flags* sind drei wichtig; man muß immer entweder `MAP_SHARED` oder `MAP_PRIVATE` angeben:

  `MAP_SHARED`**:** In diesem Fall sind die Daten im `mmap`-Bereich geshart: Jede Änderung an den Daten ist sofort für alle Prozesse, die ein `mmap` auf diese Datei haben, sichtbar. Außerdem ist garantiert, daß alle Änderungen auch in die betreffende Datei zurückgespeichert werden.

---

[50] Auf einigen Unix-Systemen verwendet `malloc` intern selbst `mmap`, um neuen Speicher vom Betriebssystem anzufordern.

[51] Man sorgt bei einem neuen oder zu kurzen File normalerweise für die richtige Länge, indem man ein `lseek` ein Byte vor die Position macht, die das neue Ende werden soll, und dann mit `write` ein Byte dorthin schreibt.

[52] Wenn der `mmap`-Bereich Maschinencode enthält, muß man `PROT_EXEC` angeben, um diesen ausführen zu dürfen. Diese Technik wird für dynamisch ladbare Libraries sowie von Debuggern und Interpretern, die Code "on the fly" erzeugen, nachladen oder modifizieren, verwendet.

`MAP_PRIVATE`: In diesem Fall bekommt der aufrufende Prozeß (und jeder erbende Prozeß) in seinem `mmap`-Bereich eine "private" Kopie der Datei: Ändert er in seinem `mmap`-Bereich Daten, so sind diese Änderungen in den `mmap`-Bereichen dieser Datei in anderen Prozessen *nicht* sichtbar (auch nicht in Sohnprozessen und anderen Verwandten), und sie werden auch ***nicht*** in die Originaldatei zurückgeschrieben.

`MAP_ANONYMOUS` **oder** `MAP_ANON`: In diesem Fall wird ein `mmap`-Bereich ohne jede Verbindung zu einem File auf der Platte eingerichtet: *fd* und *offset* werden in diesem Fall ignoriert, man übergibt üblicherweise `-1` für *fd*. Ein solcher Bereich ist nur für jenen Prozeß (und dessen Söhne) sichtbar, der das `mmap` gemacht hat, kein anderer kann darauf zugreifen (wie bei einem mit `IPC_PRIVATE` angelegten Shared-Memory-Segment). Dieser Modus wird normalerweise zur Anforderung zusätzlicher lokaler Hauptspeicherbereiche verwendet.

- *fd* ist der Filedeskriptor jenes Files, dessen Inhalt eingeblendet werden soll. Man muß also zuerst ein `open` auf den File machen, bevor man ihn mit `mmap` einblenden kann.

  Nach dem `mmap` kann man sofort wieder ein `close` auf den Filedeskriptor machen, der `mmap`-Bereich bleibt davon unberührt und funktioniert auch bei geschlossenem File weiter.

  Wie bereits oben erwähnt, wird in vielen Fällen der Pseudo-File `/dev/zero` verwendet, um ein auf `0` initialisiertes, aber nicht mit einem File auf der Platte verbundenes Shared Memory anzulegen.

- *offset* ist der Offset des eingeblendeten Bereiches im Fileinhalt, d. h. das erste Byte im `mmap`-Bereich entspricht dem Byte *offset* im File. Wenn man den File von Anfang an einblenden möchte, gibt man daher `0` als Offset an.

  offset muß bei den meisten Implementierungen ein Vielfaches der Page-Größe (je nach System üblicherweise `4096` oder `8192`) sein.

Der Returnwert ist ein Pointer auf den Anfang des `mmap`-Bereiches oder `-1` bei Fehler (wie bei `shmat` sollte man den Returnwert auf jenen Typ casten, den man zum Zugriff auf die Daten verwenden will).

Für `mmap` benötigt man die Include-Files `<sys/types.h>` (der `size_t` und `off_t` als `int` oder `long` definiert) und `<sys/mman.h>`.

`mmap`-Bereiche werden bei `fork` vererbt, bei `exec` nicht.

Hinweis:
`mmap` schaufelt nicht sofort den gesamten File in den Speicher: Erst wenn man auf eine Page (üblicherweise 4 KB) des `mmap`-Speicherbereiches tatsächlich zugreift, wird die betreffende Page von der Platte geladen und im Hauptspeicher angelegt. Ebenso werden nur die tatsächlich veränderten Pages zurückgeschrieben, nicht der gesamte Fileinhalt. Auch die private Kopie bei `MAP_PRIVATE` wird nicht im Voraus erstellt, sondern Page für Page erst dann, wenn man wirklich Daten auf einer Page ändert.

Bei Betriebssystemen, die "Sparse Files" unterstützen, funktionieren diese auch mit `mmap` wie erwartet: Jene Bereiche des Files, die vorher leer waren und auch im `mmap`-Bereich nie beschrieben wurden, werden auch beim Zurückschreiben des `mmap`-Bereiches nicht auf der Platte angelegt.

Weitere Funktionen:

`int munmap(void *`*addr*`, size_t` *len*`)` entfernt den `mmap`-Bereich der Länge *len*, auf den der Pointer *addr* zeigt, wieder aus dem Speicher des aufrufenden Prozesses. Bei `MAP_SHARED`

werden die Änderungen automatisch in die Datei zurückgeschrieben, bei `MAP_PRIVATE` nicht. Der Returnwert ist `0` bei Erfolg, `-1` bei Fehler.

Mit dem Aufruf von `msync` (siehe `man`-Page) kann man erzwingen, daß die Änderungen eines `MAP_SHARED`-Bereiches wirklich *sofort* physikalisch auf die Platte geschrieben werden: Normalerweise merkt sich das System nur, welche Pages auf die Platte zurückgeschrieben werden müssen, aber man hat keine Kontrolle darüber, *wann* das geschieht — es kann lang nach dem `munmap`, ja sogar nach dem Ende des Prozesses sein.

Kontrolle:

`mmap`-Speicherbereiche werden von `ipcs` *nicht* angezeigt, zum Anzeigen dieser Bereiche dient das Tool `lsof` (dabei sieht man beispielsweise, daß unter Linux alle Shared Libraries mit `mmap` dynamisch in die Prozesse eingeblendet werden).

A Wiederholung Unix-Befehle

Allg. Unix-Befehle:

`ls`: Dir Listing (`-a -l -tr -R -d`)
`cp mv`: Copy und Move/Rename (`-f -r`)
`rm`: Remove (`-f -r`)
`cd`: Change Dir (ohne Arg, `-`)
`pwd`: Print Working Dir
`mkdir rmdir`: Dir anlegen / löschen
`chmod` *[`-R`] mode files*: Rechte ändern (siehe `man`-Page)
`touch`: File anlegen, Zeit ändern
`more` *[files]*: File anzeigen (`<Space> <Ret> b G g q h / n N :n :p`)
(unter Linux: `less`)
`tail` *[file]*: Ende eines Files (`tail -f`)
`fgrep` *string files*: Sucht Strings (`-i -n -c -l -v`)
`grep` *pattern files*: Sucht Patterns
`diff` *file1 file2*: Vergleicht Files

Shell:

`Ctrl-C` und `Ctrl-\`
Pipe `|` und Redirect `< > 2> >> >& 2>&1`
`Tab` für Filename Completion

`man`**:**

Man-Sections 1 (Commands), 2 (System Calls), 3 (Library Functions), 4 (Devices), 5 (File Formats), 7 (Misc., Tabellen), 8 (Administrative Commands)

Aufbau einer Man-Seite:

Section 1: Aufrufsyntax mit allen Optionen, Beschreibung (Effekt, Argumente, Options-Flags, Fehlermeldungen, Exitcode, Einschränkungen), Files, Beispiele, "see also"
Section 2, 3: Aufrufsyntax mit Parametertypen (und Includefile), Beschreibung (Effekt, Parameter, Ergebniswert), `errno`-Werte, "see also"

`apropos` (sucht in Überschriften, nicht Volltext!)

Editor:
`vi` muß notdürftig beherrscht werden
Tatsächlich verwendeter Editor egal (sinnvollerweise z. B. `kate` oder `nedit`!)
Für ASCII-Terminals (`telnet`, `ssh`): Z. B. `pico`

C-Compiler:
Aufruf: `gcc` *file*`.c` für Gnu C (normales Unix: `cc`):
Preprozessor + Compiler + Assembler + Linker = fertiges Executable

Optionen:
`-o` *file* Output-Filename (bitte nicht `a.out`! Immer `-o` verwenden!)
(*Achtung:* Executables haben unter Unix normalerweise *keine* Extension,
also *nicht* `.exe` oder `.com`!)
Achtung beim Ausführen: "`.`" ist nicht im Pfad:
`./`*exefile* starten oder `PATH` umsetzen!
`-c` ohne Linken (`.o`-File, bei mehreren Source-Files)
`-g` für Debugging-Info
`-E` nur Preprozessor, vom Preprozessor gelieferter C-Sourcecode wird auf `stdout` ausgegeben
Meine Warning-Optionen für `gcc 3.1.1`, bewirken strikte ANSI-Konformität und möglichst viele Warnings für "dubiose" Konstrukte (`bash`-Alias definieren!):
`-ansi -pedantic -Wall -W -Wtraditional -Wfloat-equal -Wundef -Wconversion -Wshadow -Wpointer-arith -Wbad-function-cast -Wcast-qual -Wcast-align -Wwrite-strings -Waggregate-return -Wstrict-prototypes -Wmissing-prototypes -Wmissing-declarations -Wredundant-decls -Wnested-externs -Wunreachable-code`
`-O` (Optimize) meiden (wegen Debugging)!
Festlegen des C-Standards: `-D_XOPEN_SOURCE=600` (siehe `features.h`!)

`make`:
Grundlegende Arbeitsweise:

- Default-Regeln plus `Makefile`
- Variablen (`CC` = Compilername, `CFLAGS` = Flags, ...)
- Abhängigkeiten: "`prog` hängt ab von `prog.c` und `common.h`"
- Befehle: "Was muß ausgeführt werden, um `prog` aus `prog.c` zu erzeugen?"

`ps`, `kill`:
ps-Optionen und -Ausgabe (`ps -ef` oder `ps aux`)
`kill -6` (mit `core`), `kill -9` (sicherer Tod)
`quit`-Character `Ctrl-\`
Linux: `pstree`, `top`, ...

Debugger: 3 Modes:

- Ausführen im Debugger (Breakpoints, Single-Step, ...)
- Debuggen eines normal laufenden Prozesses
- Post-Mortem-Analyse von Cores (Source-Stelle, Stack, Variablen, ...)

Aufruf: `gdb` *exec-file core-file*
Stack Backtrace: `where` (oder `bt`)
Stack Frame wechseln: `up`, `down`

Lokale Variablen: `info locals`
Ausdrücke: `print` *expr*
Weitere Commands: `help`, Manual
Ausstieg: `quit`
Grafische Frontends: `insight`, `kdbg`, `ddd`, ...
Normale Unix-Debugger heißen `sdb` oder `dbx`

`strace`: System Call Tracer:
Schreibt alle Kernel-Aufrufe mit (Argumente & Return-Wert)

`ltrace`: Library Call Tracer:
Schreibt alle C-Library-Aufrufe mit (Argumente & Return-Wert)

`indent`: Formatiert C-Quelltexte

`lsof` und `fuser`:

Zur Fehlersuche: Anzeige offener Files, Netzwerkverbindungen, Working Dir's, ... pro Prozeß
Viele Optionen: Alle, pro Filesystem, pro User, ...

Ursprünglich:
"Warum kann ich das Filesystem nicht unmounten?!"
"Wer hat den riesigen gelöschten File noch offen?!"

`fuser` gibt's standardmäßig auf (fast) jedem Unix
`lsof` kann mehr, ist aber Open Source (auf kommerziellen Unixes normalerweise nicht drauf, selber compilieren!)

B Unix Shell Jobs

Diese sind Sache der Shell (werden von jeder einzelnen Shell separat für sich verwaltet), nicht Sache des Betriebssystems!

command `&`
Führt *command* im Hintergrund aus (parallel zur Shell)
Anzeige: Job-Nummer und Pid
command wird angehalten, sobald es I/O auf das Terminal machen will[53]!
(daher ev.: I/O Redirect)
Wenn fertig: Exit-Meldung auf der Shell

`jobs -l`
Listet derzeitige Jobs: Nummer, Pid, Status, Command

`Ctrl-Z`
Stoppt das gerade im Vordergrund laufende Programm
und macht einen (angehaltenen!) Hintergrund-Job daraus
Es kommt wieder der Shell-Prompt

[53] Unter Linux dürfen Hintergrund-Jobs per Default auf das Terminal schreiben, unter anderen Unix-Systemen werden sie üblicherweise angehalten, wenn sie versuchen, Terminal-I/O zu machen, bis sie wieder in den Vordergrund geholt werden. Eingestellt wird das mit `stty tostop` (Prozeß wird gestoppt) und `stty -tostop` (Prozeß darf schreiben).

Im Hintergrund weiterlaufen lassen: `bg`
Wieder in den Vordergrund holen: `fg`

`bg`, `bg` *%job*
: Läßt den aktuellen / den angegebenen Job im Hintergrund weiterlaufen
(bis fertig / Terminal-I/O / `kill`)

`fg`, `fg` *%job*
: Holt den aktuellen / den angegebenen Job in den Vordergrund
(läuft normal am Terminal weiter)

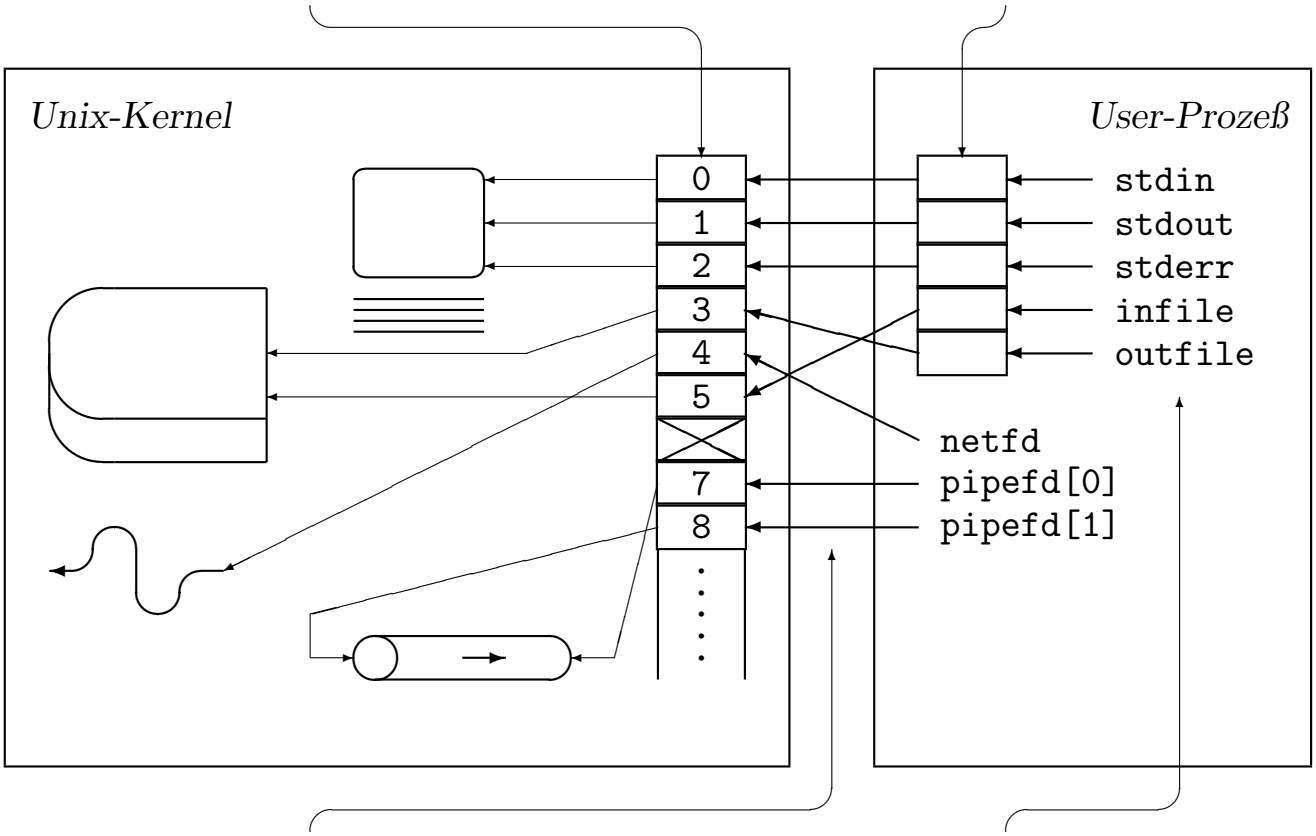
`wait`, `wait` *pid*, `wait` *%job*
: Wartet auf alle Sohnprozesse / den angegebenen Sohnprozess / den angegebenen Job dieser Shell

`kill` *[-signal]* *%job*
: Wie normales `kill`, aber für Jobnummern.

C Filepointer und Filedeskriptoren

Kernel-File-Tabelle
(eine Tabelle pro Prozeß)

C-Stdio-Library
(Formatierung, Pufferung, ...)



Filedeskriptoren
`int`
(= Kernel-File-Tabellen-Index)
`0` = `STDIN_FILENO`
`1` = `STDOUT_FILENO`
`2` = `STDERR_FILENO`
`open`, `creat`, `pipe`, `dup2`
`read`, `write`
`close`

Zu einem neuen Filedeskriptor (von `open`, `pipe`, ...) wird *nicht* automatisch ein File-Pointer geöffnet: Bei Bedarf händisch mit `f=fdopen(fd, ...)` machen!

Bei `fork` wird die File-Tabelle kopiert.

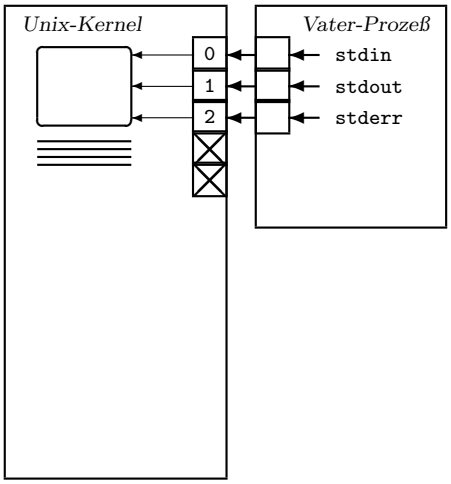
Die File-Tabelle im Kernel bleibt von `exec` unbeeinflußt, offene Filedeskriptoren gelten auch im neuen Programm (außer `close-on-exec`).

File-Pointer
`FILE *`
(= Pointer auf Puffer-Struktur)
`stdin`
`stdout`
`stderr`
`fopen`, `popen`, `fdopen`, `freopen`
`fgets`, `printf`, ...
`fclose`, `pclose`

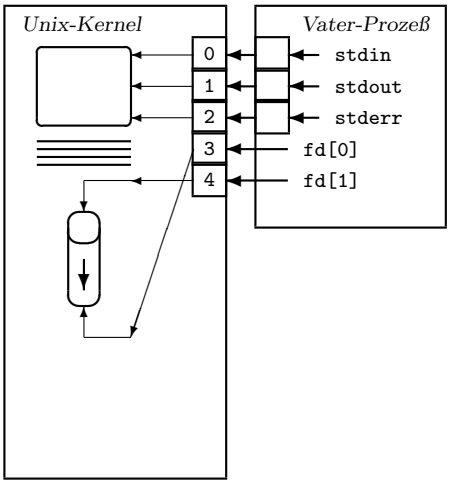
Zu jedem offenen File-Pointer gehört *automatisch* immer genau ein Filedeskriptor (`fopen` macht intern `open`), erfragbar mit `fd=fileno(f)`.

Bei `fork` werden die File-Pointer samt Puffer-Strukturen kopiert.

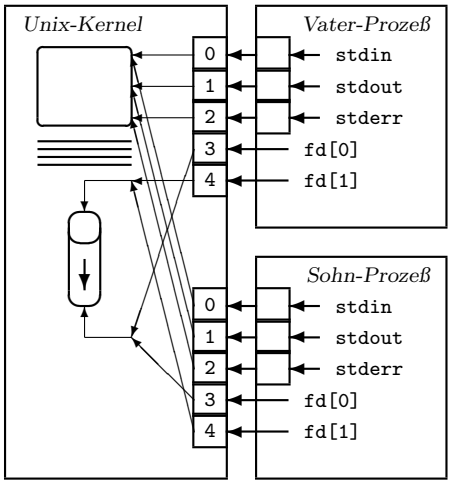
File-Pointer gehören zum Programm und gehen daher bei einem `exec` verloren, daher bei Bedarf mit `fdopen` neue File-Pointer zu den geerbten Filedeskriptoren öffnen.



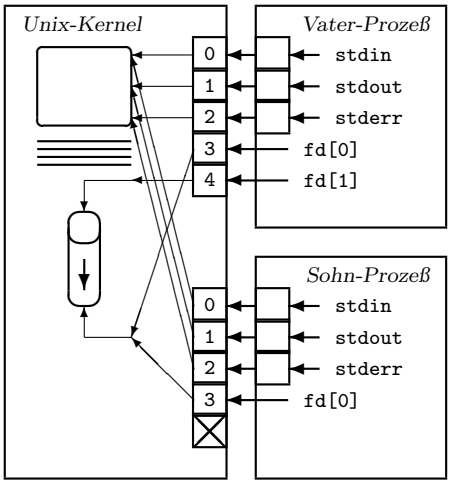
Bei Programmstart:
Nur `stdin`, `stdout`, `stderr` auf das Terminal offen, keine offenen Files



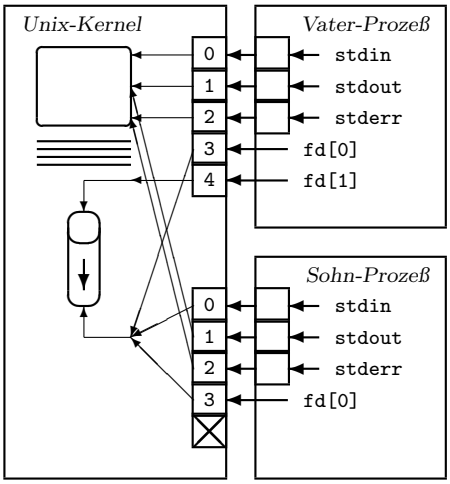
Nach `pipe(fd)`



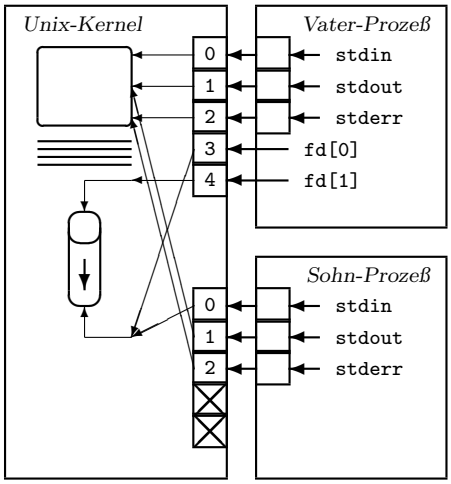
Nach `fork()`



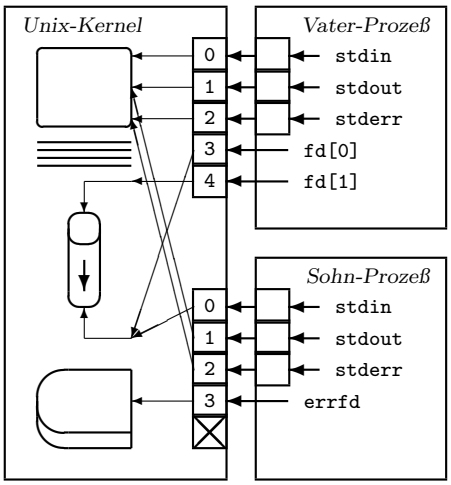
Nach `close(fd[1])` **im Sohn:**
Sohn braucht nur Lese-Ende der Pipe und schließt daher sein Schreib-Ende



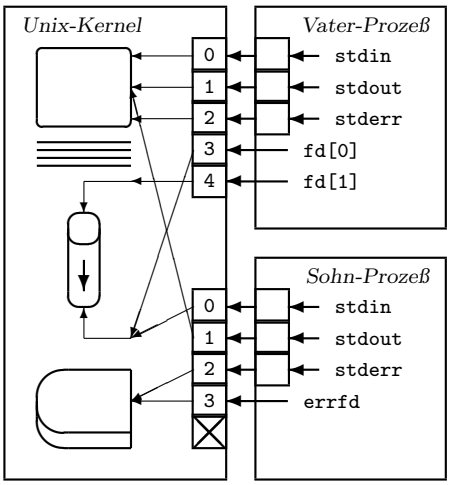
Nach `dup2(fd[0], STDIN_FILENO)` **im Sohn:**
Sohn verbindet Lese-Ende der Pipe mit seinem `stdin` ...



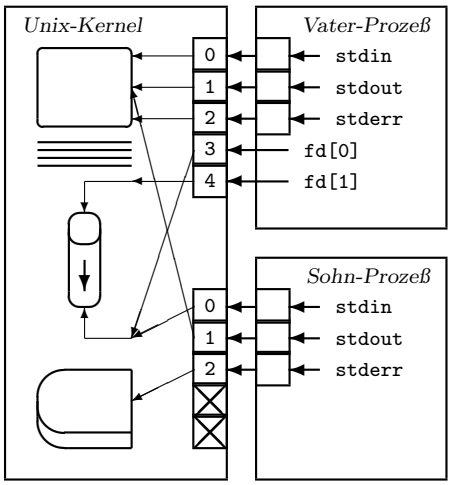
Nach `close(fd[0])` **im Sohn:**
... und schließt danach sein ursprüngliches Lese-Ende



Nach `errfd=open(errname, ...)` **im Sohn:**
Sohn öffnet einen File zum Schreiben der Fehler, ...

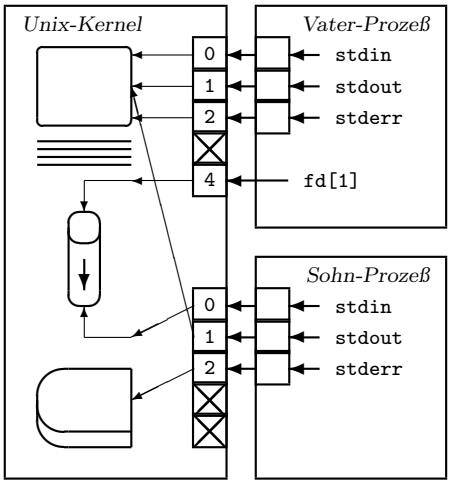


Nach `dup2(errfd, STDERR_FILENO)` **im Sohn:**
... verbindet sein `stderr` mit diesem offenen File ...

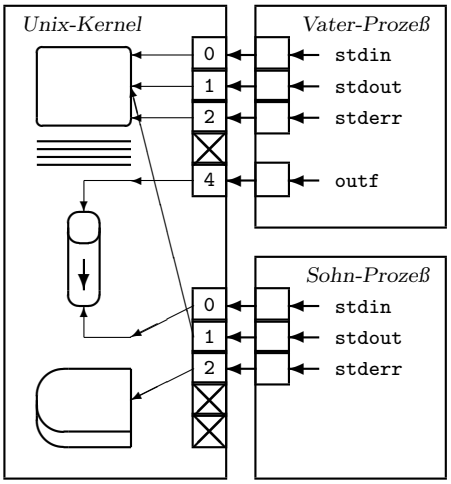


Nach `close(errfd)` **im Sohn:**
... und schließt den ursprünglichen File:

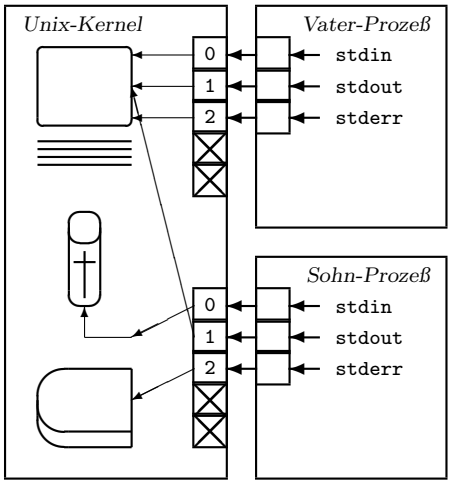
Alles bereit für ein `exec`*!*



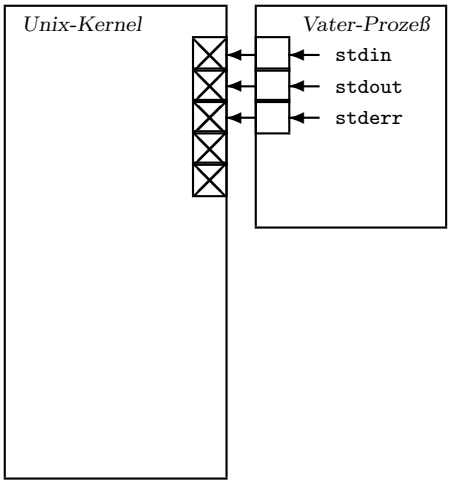
Nach `close(fd[0])` im Vater:
Vater braucht nur Schreib-Ende der Pipe
und schließt daher sein Lese-Ende



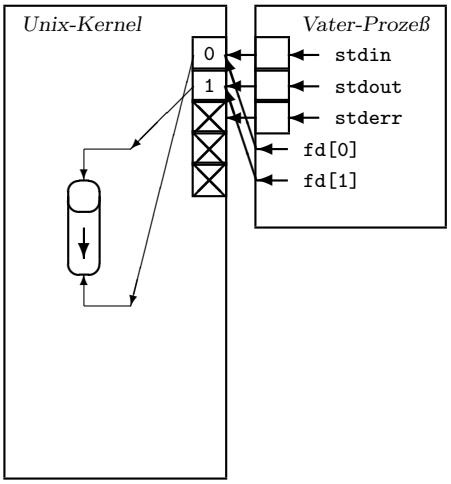
**Nach `outf=fdopen(fd[1], "w")`
im Vater:**
Vater kann nun über `outf` komfortabler
als über `fd[1]` in die Pipe schreiben



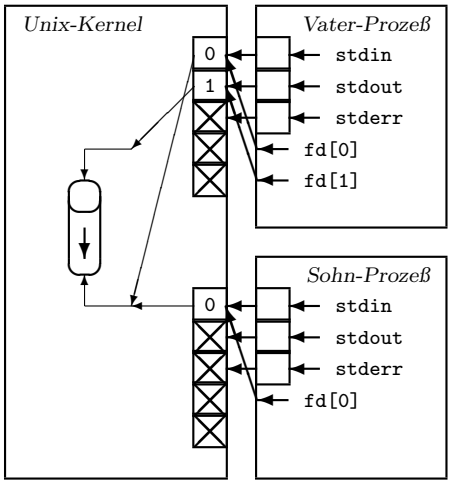
**Am Ende des Vater-Programms,
nach `fclose(outf)` im Vater:**
Pipe hat keinen Schreiber mehr,
Sohn bekommt beim Lesen `EOF`



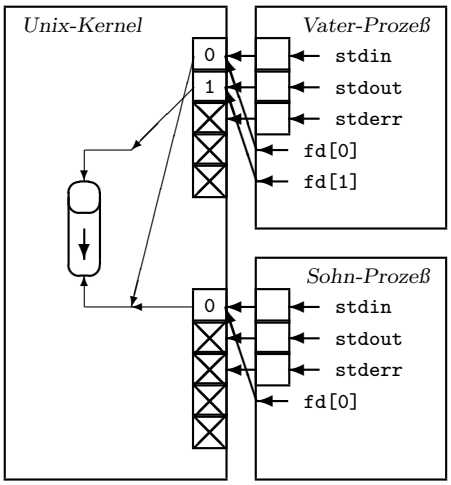
2. Beispiel: Warum ein `if` vor `dup2`?
Programm wird mit geschlossenem
`stdin`, `stdout` und `stderr` gestartet,
keine offenen Files



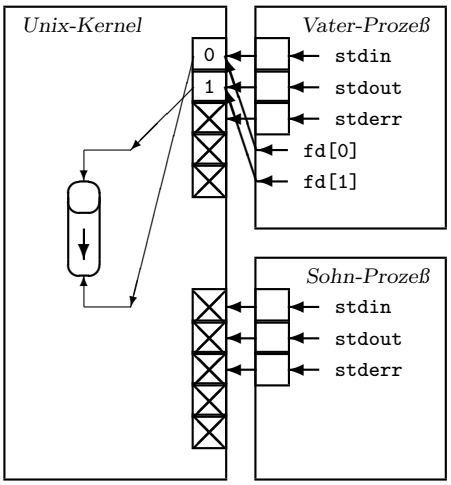
Nach `pipe(fd)`: `fd[0]` = 0, `fd[1]` = 1
`pipe` nimmt die ersten freien Nummern:
`0` und `1`! Damit hängen `stdin` und `stdout`
schon "unabsichtlich" an der Pipe!



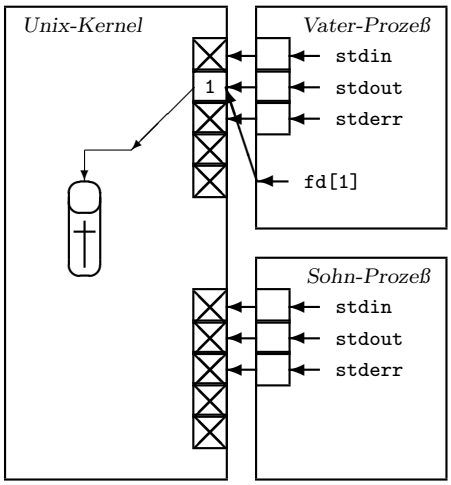
**Nach `fork()` und `close(fd[1])`,
vor `dup2`:**
Jetzt würde das `if` ansprechen,
nehmen wir an, es fehlt!



**Nach `dup2(fd[0], STDIN_FILENO)`
im Sohn:**
Das ist `dup2(0, 0)`
und macht gar nichts!



Nach `close(fd[0])` im Sohn:
Das schließt File `0` (d. h. `stdin`) im Sohn
und damit seinen letzten Verweis
auf das Lese-Ende der Pipe!



Nach `close(fd[0])` im Vater:
Wenn der Vater wie üblich sein Lese-Ende
schließt, hängt die Pipe in der Luft,
das erste Schreiben bewirkt ein `SIGPIPE`