Ausarbeitung

im Kurs

Unix Konzepte der Systemnahen Programmierung

Projektbetreuer : Prof. Frank

Eingereicht am : 10.01.2016

Matthias Beyer
Immatrikulationsnummer: CENSORED

CENSORED, CENSORED, CENSORED

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

API Application Programming Interface

FIFO First In First Out

IPC Inter Process Communication

ISO International Organization for Standardization

POSIX Portable Operating System Interface

TCP Transmission Control Protocol

UDP User Datagram Protocol

1. Einführung

Systemnahe Software unter Unixoiden Betriebssystemen bedient sich Schnittstellen des Betriebssystems um effizient und einfach Aufgaben zu erledigen.

Diese Schnittstellen sind in verschiedenen Standards definiert, der bekannteste dieser Standards, POSIX, wurde im Jahr 1985 initiiert und besteht seitdem in verschiedenen Versionen ([Wik15b]).

Ein Programm unter dem Betriebssystem Linux (welches allerdings nicht komplett POSIX-Konform ist) kann sich an den verschiedenen Schnittstellen des Systems bedienen um Funktionalität abzubilden. Diese von Linux angebotenen Schnittstellen sollen in dieser Arbeit analysiert und diskutiert werden. Diese Arbeit kann somit als Einleitung und/oder Sammlung von Beispielen zur systemnahen Programmierung unter Linux benutzt werden. Grundlagen in der Programmierung mit der Programmiersprache "C" werden dabei vorausgesetzt, da diese Arbeit ausschließlich "C" als Programmiersprache benutzt. Die angesprochenen Schnittstellen beschränken sich allerdings nicht auf die Programmiersprache "C" und können auch mit anderen Programmiersprachen benutzt werden.

2. Grundlagen

In den folgenden Kapiteln sollen Systemschnittstellen, welche der POSIX-Standard definiert, analysiert und kommentiert werden. Einige wenige Schnittstellen werden dabei als gegeben vorausgesetzt.

2.1. Programmaufbau

Ein "C"-Programm wird unter Linux mit drei Parametern aufgerufen:

Listing 2.1: Einfache main-Funktion

```
1 int main(int argc, char** argv, char** envp) {
2     return 0;
3 }
```

"argc" beschreibt einen "int"-Wert und gibt die Länge der Liste von Zeigern an, welche sich in "argv" befindet.

"argv" beschreibt einen Zeiger auf eine Liste von Zeigern, welche auf "NULL"-terminierte Zeichenketten zeigen. Die Länge von "argv" ist durch "argc" angegeben.

"envp" beschreibt einen Zeiger auf eine Liste von Umgebungsvariablen. "envp" ist nicht Teil des POSIX-Standards ([Wik15a]).

Diese Programmparameter werden beim Starten des Programms vom Betriebssystem übergeben. Die Parameterliste kann durch ein "void" ersetzt werden.

Der Rückgabewert der "main"-Funktion ist in obigem Beispiel Null. Dieser Wert wird dem Betriebssystem zurückgegeben um Fehler oder Erfolg anzuzeigen.

Unter Linux werden in der Header-Datei "stdlib.h" die beiden Macros "EXIT_SUCCESS" und "EXIT_FAILURE" definiert, welche anstatt einer Zahl verwendet werden sollten.

2.2. Dateien

Unter Unixoiden Betriebssystemen werden alle Systemressourcen als Dateien abgebildet. Dies vereinfacht viele Betriebssystem-Interaktionen, da diese auf einfache Dateioperationen reduziert werden. Dateien können dabei einen bestimmten Typ annehmen, unter Unixoiden Betriebssystemen sind diese Dateitypen bekannt:

- Normale Datei
- Ordner
- Symbolischer Link
- Block-Orientierte Gerätedatei
- Character-Orientierte Gerätedatei
- Röhre, benamt und unbenamt
- Netzwerksocket

[BC05, S. 14]

Für jede Datei sind zudem Metainformationen im POSIX-Standard festgelegt, welche der Betriebssystemkern zur Verfügung stellen können muss, um POSIX-konform zu sein:

- Typ
- Anzahl der harten Links, welche die Datei referenzieren
- Länge der Datei in Bytes
- Identifizierung des Gerätes welche die Datei speichert
- Inode-Nummer
- Nutzer-Identifikation des Besitzers der Datei
- Gruppen-Identifikation der Gruppe des Besitzers der Datei
- Verschiedene Zeitstempel
- Zugriffsberechtigungen

[BC05, S. 15 ff]

2.3. Datei-Handle

"Datei-Handle" (englisch: "File-Descriptor", pl.: "File-Descriptors", oft kurz: "fd"/"fds") bezeichnen Ganzzahlen welche das Betriebssystem nutzen kann um Dateien (und dateiartige Ressourcen) zu identifizieren. Unter Linux werden in jedem Programm die "Handles" 0, 1 und 2 verfügbar gemacht. Jeder dieser "Handles" beschreibt dabei einen Eingabe-Ausgabe-Strom (englisch: "IO-Stream"), welcher genutzt wird um zum Beispiel mit dem Nutzer auf der Kommandozeile zu interagieren:

Listing 2.2: Ausgabe auf stdout, stderr

```
1 int main(void) {
2     write(0, "Hallo\n", 6);
3     write(1, "Error-Ausgabe\n", 14);
4     return 0;
5 }
```

Dabei beschreibt der Deskriptor "0" beschreibt den "standard in" Strom (kurz: "stdin"), welcher zum Beispiel dafür genutzt werden kann um von der Kommandozeile zu lesen (nicht in obigem Beispiel benutzt). "1" den sogenannten "standard out" (kurz: "stdout") Strom, welcher für normale Ausgaben genutzt werden sollte und der Deskriptor "2" den sogenannten "standard error" (kurz: "stderr"), welcher für Fehlerausgaben genutzt werden sollte.

Um auf den Deskriptoren zu schreiben wird, wie in Listing 2.2 zu sehen ist, der Systemaufruf "write" verwendet. "write" werden folgende Argumente übergeben:

- Ein Deskriptor
- Ein Zeiger auf den Puffer welcher ausgegeben werden soll
- Die Anzahl der Bytes, welche aus dem Puffer auf den Deskriptor geschrieben werden sollen

Um ein solches Handle für eine Datei zu erlangen, kann der Programmierer den Systemaufruf "open" verwenden. Dieser Aufruf wird mit einem Pfad aufgerufen und gibt dann eine Ganzzahl zurück, welche die geöffnete Datei identifiziert. Diese Ganzzahl/dieser Deskriptor kann nun verwendet werden um in die Datei zu schreiben, von ihr zu lesen, etc.

Weitere Systemaufrufe welche verwendet werden können um mittels Datei-Handles zu interagieren sind:

- "read" um davon zu lesen
- "lseek" um den "Schreibekopf" zu bewegen
- "close" um den Deskriptor zu schließen

Deskriptoren werden zudem verwendet um geteilten Speicher zu verwalten und Nachrichtenschlangen (engl.: "Message Queues") oder Netzwerkressourcen anzusprechen.

Intern werden Deskriptoren verwendet um auf Einträge in der sogenannten "File-Table" zu zeigen. Diese Tabelle beinhaltet Daten darüber, welcher Deskriptor in welchem Modus geöffnet wurde (schreibend, lesend, etc.) und einen Zeiger auf die zugehörige Inode. Eine Inode beschreibt eine physisch vorhandene Datei auf einem Datenträger.

3. Systemaufrufe

In diesem Kapitel sollen verschiedene Systemaufrufe besprochen werden, welche es einem Programm ermöglichen mit dem Betriebssystemkern zu kommunizieren.

Jeder Systemaufruf wird in einer Tabelle wie folgt beschrieben:

Name	Name des Aufrufs
Header	Headerdatei in welcher er definiert ist
Rückgabetyp	Rückgabetyp
Rückgabe im Erfolgsfall	Erklärung des Rückgabewerts im Erfolgsfall
Rückgabe im Fehlerfall	Erklärung des Rückgabewerts im Fehlerfall
Errno	evlt. gesetzte Werte der "errno"-Variable
Parameter	Parameter welche der Aufruf akzeptiert

Tabelle 1.: Systemaufruf: Name des Aufrufs

3.1. fork

Der Systemaufruf "fork" (3.1) bewirkt ein Aufsplitten des Prozesses in zwei absolut gleiche Prozesse. Dies ermöglicht dem Programmierer, Prozesse zu starten. Da der Systemkern den Prozess absolut gleich kopiert wird im Programm nur über den Rückgabewert des Aufrufs unterschieden, ob das Programm im Kind-Prozess oder Eltern-Prozess weiter läuft. Der Programmierer hat volle Kontrolle über beide Prozesse. Mit Hilfe anderer Systemaufrufe kann der Programmierer Kommunikationswege zwischen den Prozessen herstellen oder auch andere Programme ausführen.

Name	fork
Header	unistd.h
Rückgabetyp	pid_t
Rückgabe im Erfolgsfall	PID des Kind-Prozesses an den Eltern-Prozess, 0 an den Kind-Prozess
Rückgabe im Fehlerfall	-1
Errno	EAGAIN, ENOMEM, ENOSYS
Parameter	void

Tabelle 2.: Systemaufruf: fork

Listing 3.1: Fork Systemaufruf

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int main(void) {
6      int pid = fork();
7
8      if (pid == 0) {
9          write(1, "Child␣Process\n", 14);
10         exit(0);
11     } else {
12         write(1, "Parent␣Proces\n", 14);
13         wait(NULL);
14         write(1, "Parent␣Process␣exiting\n", 23);
15         exit(0);
16     }
17 }
```

3.2. clone

Der Systemaufruf "clone" (3.2) ist verwand mit "fork" (3.1). Mittels "clone" (3.2) werden ebenfalls neue Prozesse erzeugt, allerdings erlaubt dieser Systemaufruf Kind-Prozessen Teile des Eltern-Prozesses zu teilen. Zum Beispiel wird der Arbeitsspeicher-bereich oder die Liste der Deskriptoren geteilt. Eine Nutzungsmöglichkeit von "clone" (3.2) ist es, Threads zu implementieren. Diese "Threads" sind allerdings mehrere Prozesse des gleichen Programms, welche Zugriff auf den gleichen Speicherbereich

haben.

Name	clone
Header	sched.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	Thread-ID des Kind-Prozesses
Rückgabe im Fehlerfall	-1
Errno	EAGAIN, EINVAL, ENOMEM, EPERM, EUSERS
Parameter	int (*fn)(void*); void* child_stack, int flags, void* arg, ..., pid_t *ptid, struct user_desc *tls, pid_t *ctid

Tabelle 3.: Systemaufruf: clone

Dem Systemaufruf "clone" (3.2) kann eine aufzurufende Funktion übergeben werden.

Dieser Systemaufruf ist lediglich auf Linux-Betriebssystemen verfügbar und nicht teil des POSIX-Standards.

3.3. vfork

Der Systemaufruf "vfork" (3.3) ist teil des POSIX-Standards und hat eine ähnliche Funktion wie "fork" (3.1).

Name	fork
Header	unistd.h, sys/types.h
Rückgabetyp	pid_t
Rückgabe im Erfolgsfall	PID des Kind-Prozesses an den Eltern-Prozess, 0 an den Kind-Prozess
Rückgabe im Fehlerfall	-1
Errno	EAGAIN, ENOMEM, ENOSYS
Parameter	void

Tabelle 4.: Systemaufruf: fork

Dabei sind die Übergabeparameter sowie die Return-Werte die gleichen wie bei "fork" (3.1).

Die Linux Manpage zu "vfork" (3.3) beschreibt diesen Systemaufruf als Spezialfall von "clone" (3.2) ([Lin12]). Der Systemaufruf "vfork" (3.3) ist für Performance-Kritische Systeme gedacht, bei welchen der kreierte Prozess sofort den Systemaufruf "execve" benutzt. Der aufrufende Prozess wird so lange Pausiert, bis der Kind-Prozess entweder terminiert (durch einen Aufruf von "_exit") oder "execve" aufruft. Des Weiteren ist wird bei einem Aufruf von "vfork" (3.3) die "Page-Table" des Prozesses nicht kopiert. Allerdings werden wie bei "fork" (3.1) mehrere Attribute des aufrufendes Prozesses an den Kind-Prozess vererbt, wie zum Beispiel Dateideskriptoren oder das aktuelle Verzeichnis.

3.4. exec

Der Systemaufruf "exec" (3.4) besteht aus verschiedenen Aufrufen. Man spricht von einer "Familie" an Systemaufrufen.

Jeder dieser Aufrufe ersetzt das Programm, in dem der Aufruf geschieht, mit einem anderen Programm. Das heisst, dass ein Programm "A" mittels eines Aufrufs eines Befehls der "exec"-Familie sich selbst mit einem anderen Programm "B" ersetzen kann, wobei es auch sich selbst aufrufen könnte. Das Programm, welcher den Aufruf ausführt, wird dabei komplett vernichtet.

Einen Aufruf "exec" selbst gibt es nicht, es wird (unter Linux) zwischen folgenden Aufrufen unterschieden:

- execl
- execlp
- execle
- execv
- execvp
- execvpe

Jeder dieser Aufrufe kehrt nicht ins aufrufende Programm zurück, es sei denn ein Fehler ist aufgetreten.

Dabei hat jeder dieser Aufrufe verschiedene Funkionalitäten und Eigenschaften, welche im Folgenden beschrieben werden sollen.

3.4.1. execl

Der Aufruf "execl" (3.4.1) ruft das Programm, welches als erster Parameter (als Pfad) übergeben wird, mit einer Liste von Argumenten auf. Die Argumente, welche hier übergeben werden, werden dem Programm in die "argv"-Liste übergeben, wie sie in Listing 2.1 zu sehen und in 2.1 beschrieben ist.

Name	execl
Header	unistd.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	Nichts
Rückgabe im Fehlerfall	-1
Errno	E2BIG, EACCES, EAGAIN, EFAULT, EINVAL, EIO, EISDIR, ELIBBAD, ELOOP, EMFILE, ENAMETOOLONG, ENFILE, ENOENT, ENOEXEC, ENOMEM, ENOTDIR, EPERM, ETXTBSY
Parameter	const char* path, const char* arg

Tabelle 5.: Systemaufruf: execl

3.4.2. execlp

Der Aufruf "execlp" (3.4.2) verhält sich wie der Aufruf "execl" (3.4.1), beachtet dabei aber die Umgebungsvariable "PATH" und sucht Programme über diese.

Dieser Aufruf ist besonders praktisch wenn man die Umgebungsvariablen des Systems berücksichtigen muss. Da dieser Aufruf die Umgebung selbstständig berücksichtigt, entfällt diese Arbeit beim der Implementation.

3.4.3. execle

Der Aufruf "execle" (3.4.3) verhält sich ebenfalls wie "execl" (3.4.1) und "execlp" (3.4.2), kann aber zumden noch eine Liste an Umgebungsvariablen an das aufzurufende Programm übergeben.

3.4.4. execv, execvp, execvpe

Die Aufrufe "exec", "execvp" und "execvpe" verhalten sich wie die Aufrufe "execl", "execli" und "execlp" allerdings erwarten diese Aufrufe keine Ellipse als Argument, sondern eine Liste an Zeigern auf Strings. Die Übergebene Liste an Argumenten muss mit einem "NULL"-Pointer beendet werden. Diese Verhaltensweiße wirkt in manchen Situationen vorteilhaft auf die Menge des zu schreibenden Quellcodes aus, zum Beispiel wenn die Argumente oder die Umgebungsvariablen bereits in einer Liste vorliegen.

3.4.5. Beispiel

In Listing 3.2 wird mittels "execvp" (3.4.4) das Programm "ls" aufgerufen, nachdem ein "fork" (3.1) das aufrufende Programm in zwei Programme geteilt hat. Das aufrufende Programm (das "Parent") wartet dabei auf die Ausführung des Kindprogrammes um eventuelle Fehler ausgeben zu können.

Listing 3.2: Beispiel: exec

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <string.h>
5  #include <errno.h>
6  #include <sys/types.h>
7  #include <sys/wait.h>
8
9  int main(void) {
10     if (fork() == 0) {
11         printf("doing in the child: ls\n");
12         int res = execvp("ls", (char* const[]){ "." });
13         printf("Hmm... %i, %s\n", res, strerror(errno))
               ;
14         exit(45);
15     } else {
16         printf("Parent waiting\n");
17         int stat;
18         wait(&stat);
19         printf("Parent closes, child status: %i, sys
               status: %i\n",
20                 stat >> 8,       // child status
```

```
21                  stat & 0xFF);    // system status
22          exit(0);
23      }
24  }
```

Die Ausgabe welches dieses Programm erzeugt, wird in 3.3 dargestellt.

Listing 3.3: Ausgabe: exec

```
1 # Ausgabe von `ls`
2 Parent waiting
3 Parent closes, child status: 0, sys status: 0
```

In Listing 3.4 wird ein nicht vorhandenes Programm "foo" aufgerufen um zu demonstrieren, wie sich das aufrufende Programm im Fehlerfall verhält.

Listing 3.4: Beispiel: Fehlerfall exec

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <errno.h>
6 #include <sys/types.h>
7 #include <sys/wait.h>
8
9 int main(void) {
10      if (fork() == 0) {
11          printf("doing␣in␣the␣child:␣foo\n");
12          int res = execvp("foo", (char* const[]){ "." })
                ;
13          printf("Hmm...␣%i,␣%s\n", res, strerror(errno))
                ;
14          exit(45);
15      } else {
16          printf("Parent␣waiting\n");
17          int stat;
18          wait(&stat);
19          printf("Parent␣closes,␣child␣status:␣%i,␣sys␣
                status:␣%i\n",
20                  stat >> 8,      // child status, prints
                        45
```

```
21                  stat & 0xFF);    // system status
22          exit(0);
23      }
24  }
```

Die Ausgabe welches das Programm mit einem Aufruf an ein nicht existierendes Programm erzeugt ist in Listing 3.5 dargestellt.

Listing 3.5: Ausgabe: Fehlerfall exec

```
1 doing in the child: foo
2 Hmm... -1, No such file or directory
3 Parent waiting
4 Parent closes, child status: 45, sys status: 0
```

3.5. wait

Dieser Systemaufruf kann benutzt werden, um auf Statusänderungen in Kind-Prozessen zu warten. Zudem können mit diesem Aufruf Informationen über den Kind-Prozess erlangt werden.

Statusänderungen in Kind-Prozessen sind:

- Prozess beendet
- Prozess durch ein Signal gestoppt
- Prozess durch ein Signal wieder gestartet

Name	wait
Header	sys/types.h, sys/wait.h
Rückgabetyp	pid_t
Rückgabe im Erfolgsfall	PID des Kind-Prozess
Rückgabe im Fehlerfall	-1
Errno	ECHILD, EINTR, EINVAL
Parameter	int* status

Tabelle 6.: Systemaufruf: wait

"wait" wartet auf eine Statusänderung eines beliebigen Kind-Prozesses. Soll auf einen bestimmten Prozess gewartet werden, kann der Systemaufruf "waitpid" verwen-

det werden. Mit "waitid" kann zudem auf eine bestimmte Statusänderung gewartet werden.

Listing 3.6: Wait Systemaufruf, Variante "waitid"

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

int main(void) {
    int pid = fork();

    if (pid == 0) {
        printf("Child Process\n");
        exit(0);
    } else {
        siginfo_t status;
        printf("Parent Proces\n");
        waitid(P_PID, pid, &status, WEXITED);
        printf("Child Process\n");
        printf(" - PID:     %i\n", status.si_pid);
        printf(" - UID:     %i\n", status.si_uid);
        printf(" - STATUS: %i\n", status.si_status);
        exit(0);
    }
}
```

Listing 3.6 zeigt einen Systemaufruf "waitid", welcher auf das Beenden des Kind-Prozesses wartet und die Informationen, welche mittels dieses Systemaufrufes erlangt werden können, danach ausgibt.

Listing 3.7: Wait Systemaufruf, Variante "waitpid", warten auf mehrere Kind-Prozesse

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

```

```c
/*
 * Compileraufruf: gcc -std=c11 -D_POSIX_C_SOURCE=200809L <source.c>
 */

#if !(_SVID_SOURCE || _XOPEN_SOURCE >= 500 || \
    _XOPEN_SOURCE && _XOPEN_SOURCE_EXTENDED || \
    _POSIX_C_SOURCE >= 200809L)
#error "waitid not supported"
#endif

#define ARY_SZ(x) (sizeof(x) / sizeof(x[0]))
#define foreach(x,iter) for (int iter = 0; i < ARY_SZ(x); ++iter)

int create_child(void) {
    int pid = fork();

    if (pid != 0) {
        return pid;
    }

    printf("Child Process: %i\n", getpid());
    exit(0);
}

int main(void) {
    int pids[] = { 0, 0, 0, 0, 0 };

    foreach(pids, i) { pids[i] = create_child(); }

    sleep(1); /* Number crunching */

    foreach(pids, i) {
        int status  = 0;
        int deadpid = waitpid(-1, &status, WIFEXITED(status));
        printf("Child Process %i exited: %i\n", deadpid, status);
```

```
43      }
44
45      printf("Parent␣Processes␣exiting\n");
46      return 0;
47  }
```

In Listing 3.7 wird mittels einer "for"-Schleife auf alle Kind-Prozesse gewartet. Der Übergabewert -1 gibt "waitpid" an, dass auf beliebige Kind-Prozesse gewartet werden soll, was einem Aufruf an "wait" gleichkommen würde. In Listing 3.7 wurde "waitpid" nur zu Demonstrationszwecken verwendet. Dieser Aufruf kann durch "wait" ersetzt werden. Die einzige Unterscheidung besteht darin, dass mittels "waitpid" auf das explizite Beenden des Kindprozesses gewartet wird.

Die Ausgabe des Programmes ist in Listing 3.8 dargestellt.

Listing 3.8: Ausgabe: Wait Systemaufruf, Variante "waitpid"

```
1  Child Process: 21795
2  Child Process: 21796
3  Child Process: 21797
4  Child Process: 21799
5  Child Process: 21798
6  Child Process 21795 exited: 0
7  Child Process 21796 exited: 0
8  Child Process 21797 exited: 0
9  Child Process 21798 exited: 0
10 Child Process 21799 exited: 0
11 Parent Processes exiting
```

Nachdem ein Kind-Prozess seinen Status verändert hat, wird dessen Rückgabewert und seine Prozess-ID ausgegeben.

Listing 3.9: Compileraufruf für Beispiel: waitmulti.c

```
1  gcc -std=c11 -D_POSIX_C_SOURCE=200809L waitmulti.c
```

Bei Listing 3.7 ist zu beachten, dass der Compileraufruf sich von den anderen Beispielen dieser Arbeit unterscheidet: Der Compiler muss hier mit speziellen Parametern aufgerufen werden, wie sie in Listing 3.9 zu sehen sind. Wird das Programm so umgeschrieben, dass die Deklarationen der Variablen nicht innerhalb des Kopfes der "for"-Schleifen geschiet, kann der ISO Standard von "C" zum übersetzen des Quelltextes verwendet werden. Das Makro muss definiert werden um den Systemaufruf "waitpid" zu unterstützen.

3.6. flock

Mittels des Systemaufrufes "flock" (3.6) kann eine Sperre (engl.: "lock") auf einen Deskriptor angelegt werden. Eine solche Sperre ist allerdings mit dem Systemaufruf "flock" (3.6) nur eine Empfehlung.

Name	flock
Header	sys/file.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	0
Rückgabe im Fehlerfall	-1
Errno	EBADF, EINTR, EINVAL, ENOLCK, EWOULD-BLOCK
Parameter	int fd, int operation

Tabelle 7.: Systemaufruf: flock

Die Sperren welche mittels "flock" (3.6) angelegt werden, werden mit einem (offenen) Dateideskriptor assoziiert, wobei duplizierte Dateideskriptoren berücksichtigt werden ("dup" (3.8)). Eine Sperre wird losgelassen wenn die entsprechende Funktionalität aufgerufen wird, oder alle Dateideskriptoren zu der entsprechenden Ressource geschlossen sind. Dies bedeutet zum Beispiel, dass ein Prozess eine Datei nicht über seine Lebenszeit hinaus sperren kann.

Die Sperren welche mit "flock" (3.6) erstellt werden können auch geteilt werden zwischen Prozessen. Dazu muss beim Aufruf von "flock" (3.6) der Parameter "LOCK_SH" als Parameter im "operation"-Feld übergeben werden. Das Gegenstück dazu wäre der Parameter "LOCK_EX", welcher eine exclusiven Sperre anlegt. Ein Prozess kann nur eine Sperre auf eine Datei halten, egal ob diese Sperre "shared" oder "exclusive" angelegt wurde. Zudem kann ein solcher Aufruf nicht-blockierend ausgeführt werden ("LOCK_NB" muss übergeben werden).

Der Rückgabewert von "flock" (3.6) gibt an, ob eine Sperre angelegt wurde. Ist der Rückgabewert eine Zahl ungleich von Null, so muss die "errno"-Variable überprüft werden.

Weitere Informationen zu "flock" (3.6) sind in der Manpage des Aufrufes nachzulesen ([Lin14]).

3.7. lockf

Mittels des Systemaufrufes "lockf" (3.7) kann eine POSIX-Konforme Sperre auf einen Deskriptor angelegt, getestet oder entfernt werden.

Name	lockf
Header	unistd.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	0
Rückgabe im Fehlerfall	-1
Errno	EACCES, EAGAIN, EBADF, EDEADLK, EINVAL, ENOLCK
Parameter	int fd, int cmd, off_t len

Tabelle 8.: Systemaufruf: lockf

Aus Linux-Betriebssystemen ist der Aufruf "lockf" (3.7) lediglich ein Interface auf den Systemaufruf "fcntl", welcher diese Funktionalität auch anbietet.

Wie auch schon beim Systemaufruf "flock" (3.6) kann an "lockf" (3.7) übergeben werden, welche Operation ausgeführt werden soll. So existieren Flags für normale Sperren, nicht-blockierende Sperren und um eine Sperre zu testen.

Wie auch "flock" (3.6) gibt der Aufruf "lockf" (3.7) eine Zahl ungleich Null zurück, falls der Aufruf fehl schlug. Danach muss die "errno"-Variable überprüft werden um den Fehler zu identifizieren.

Der Aufruf "lockf" (3.7) unterscheidet sich durch die Möglichkeit, bestimmte Bereiche einer Datei zu sperren, von "flock" (3.6).

Weitere Informationen zu "lockf" (3.7) sind in der Manpage des Aufrufes nachzulesen ([Lin15b]).

3.8. dup

Der Aufruf "dup" kann zur Duplizierung eines Deskriptors genutzt werden. Der duplizierte Deskriptor referenziert die gleiche Systemressource, hat allerdings unterschiedliche Eigenschaften. So ist zum Bespiel die Option "FD_CLOEXEC" auf dem neuen Deskriptor ausgeschalten.

Name	dup
Header	unistd.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	Neuer Deskriptor
Rückgabe im Fehlerfall	-1
Errno	EBADF, EBUSY, EINTR, EINVAL, EMFILE
Parameter	int fd

Tabelle 9.: Systemaufruf: dup

Varianten des Systemaufrufes sind "dup2" und "dup3", welche einen zu benutzenden Deskriptor als zweites Argument, und im Fall von "dup3" zusätzliche Optionen als drittes Argument erwarten. Im Falle von "dup2" wird der zweite Parameter als Dateideskriptor verwendet, anstatt eines neuen Deskriptors. Sollte der Deskriptor bereits existieren, so wird dieser geschlossen.

Der Aufruf "dup3" erweitert dieses Verhalten und ermöglicht zudem Flags auf dem neuen Deskriptor zu setzen.

Alle drei Varianten des "dup" (3.8)-Aufrufes liefern im Fehlerfall -1 zurück und setzen die "errno"-Variable entsprechend.

Weitere Informationen zu "dup" (3.8) sind in der Manpage des Aufrufes nachzulesen ([Lin15a]).

3.9. Kombination von Systemaufrufen

Während in vorhergehenden Kapiteln schon Systemaufrufe kombiniert wurden um bestimmtes Verhalten einzelner Aufrufe aufzuzeigen, sollen in diesem Kapitel Aufrufe kombiniert werden um Systemoperationen mittels mehreren Prozessen auf gleichen Ressource aufzuzeigen. Dazu werden mittels dem Aufruf "fork" (3.1) mehrere Prozesse erzeugt, welche dann auf die gleiche Ressource zugreifen.

Listing 3.10: "write" aus mehreren Prozessen

```
1 #include <stdio.h>
2 #define _GNU_SOURCE
3 #include <stdlib.h>
4 #include <unistd.h>
5
6 int main(void) {
```

```
7       static char t[] = "/tmp/combwrite-XXXXXX";
8       int fd = mkstemp(t);
9
10      if (fd == -1) {
11          perror("Cannot create tempfile\n");
12      }
13
14      if (fork() == 0) {
15          int i;
16          for (i = 0; i < 100; ++i) {
17              write(fd, "Kind-Prozess\n", 13);
18          }
19          exit(0);
20      } else {
21          int i;
22          for (i = 0; i < 100; ++i) {
23              write(fd, "Eltern-Prozess\n", 15);
24          }
25          exit(0);
26      }
27  }
```

In Listing 3.10 wird eine temporäre Datei mittels "mkstemp()" angelegt, welche dann aus zwei Prozessen mehrmals beschrieben wird (im Beispiel 100 mal). Die Datei wird im "/tmp" Verzeichnis angelegt und ihr Dateiname beginnt mit "combwrite-" und wird von zufälligen Zeichen abgeschlossen.

Die Quellcodedatei aus Listing 3.10 muss mit "C99" oder "C11"-Standard übersetzt werden.

Listing 3.11: Auslesen der Inhalte der Temporären Datei

```
1 cat /tmp/combwrite*
```

Die Datei kann nun mit "cat" auf der Kommandozeile ausgelesen werden, wie in Listing 3.11 beschrieben.

Eventuell muss die resultierende Programmdatei mehrmals ausgeführt werden, um das gewünschte Ergebnis zu erzielen, je nach Geschwindigkeit der Maschine auf welcher das Programm ausgeführt wird. Die Datei beinhaltet die Zeilen nach ausführen der Datei in zufälliger Kombination; es kann vorkommen dass die Eltern-Zeilen und

Kind-Zeilen durcheinander in der Datei auftreten, wie in Listing B.1 aufgezeigt. Dies verdeutlicht, dass die Dateioperationen nicht synchronisiert sind.

Listing 3.12: "write" aus mehreren Prozessen

```
1  #define _GNU_SOURCE
2  #include <fcntl.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <string.h>
6  #include <sys/stat.h>
7  #include <sys/types.h>
8  #include <unistd.h>
9
10 void dowrite(const char* _path, const char* buf, size_t
       n) {
11     int i;
12     char err[256] = {0};
13     char* path = strdup(_path);
14     int fd = open(path, O_CREAT | O_APPEND, 0644);
15
16     if (fd == -1) {
17         snprintf(err, sizeof(err), "Cannot open
               tempfile");
18         goto errout;
19     }
20
21     for (i = 0; i < 100; ++i) {
22         ssize_t w = write(fd, buf, n);
23         if (w == -1) {
24             snprintf(err, sizeof(err), "Schreiben
                   fehlgeschlagen: '%s'", buf);
25             goto errout;
26         }
27     }
28
29     printf("Ready to write: '%s'\n", buf);
30     goto out;
31
32 errout:
```

```
33      perror(err);
34  out:
35      close(fd);
36      free(path);
37
38      if (*err) {
39          exit(1);
40      }
41  }
42
43  int main(void) {
44      static char t[] = "/tmp/combwrite-%i";
45      char buf[256];
46      memset(buf, 0, sizeof(buf));
47      snprintf(buf, sizeof(buf), t, getpid());
48
49      printf("Will write to: %s\n", buf);
50
51      if (fork() == 0) {
52          dowrite(buf, "Kind-Prozess\n", 13);
53      } else {
54          dowrite(buf, "Eltern-Prozess\n", 15);
55      }
56
57      return 0;
58  }
```

4. Interprozesskommunikation

Oftmals müssen Prozesse unter Linux miteinander kommunizieren. Diesen Vorgang nennt man Interprozesskommunikation, auf englisch "Interprocess communication" oder auch nur kurz: Inter Process Communication (IPC). Um dies zu realisieren gibt es unter POSIX-Artigen Betriebssystemen verschiedene Techniken, welche in diesem Kapitel beschrieben werden.

4.1. Pipes

"Pipes", also Röhren, sind die älteste Form der Interprozesskommunikation auf unixoiden Betriebssystemen ([Ker10, S. 889]). Pipes werden verwendet, um mehrere Programme miteinander agieren zu lassen. So arbeiten die Aufrufe in Listing 4.1 zusammen um Daten zu extrahieren, zu modifizieren und um ein gewünschtes Ergebnis, in diesem Fall die Temperatur des Prozessors, auszugeben.

Listing 4.1: Bash: Temperatur des Prozessors

```
1 sensors | grep Phys | cut -d + -f 2 | cut -d . -f 1
```

Das Programm "sensors" gibt Informationen über Systemsensoren aus, welche dann mittels dem Programm "grep" nach dem Text "Phys" gefiltert werden. Die Zeile, welche "grep" findet, wird nun mittels "cut" zerstückelt und (im ersten Aufruf) der Teil vor dem "+"-Zeichen, sowie der Teil nach dem "."-Zeichen wird abgeschnitten

In Listing 4.1 ist dargestellt, wie mehrere Programme mittels Pipes zusammenarbeiten. Das erste Programm speist Daten in das zweite Programm ein, welches die Daten verarbeitet und diese wiederum in das dritte Programm einspeist. Dies kann nahezu beliebig oft wiederholt werden. Dabei wissen die einzelnen Programme nichts voneinander und keines der Programme hat spezielle Funktionalitäten einprogrammiert, um dies zu bewerkstelligen. Der aufrufende Prozess (bash) kümmert sich hier um die Verbindung zwischen den Programmen. Nichts desto trotz kann diese Funktionalität innerhalb eines Programmes repliziert werden und beschränkt sich dabei nicht auf den Eingabe-Ausgabe-Strom des Prozesses.

Eine Pipe ermöglicht byteorientierte Kommunikation zwischen Prozessen. Das heisst, dass lediglich Bytes übertragen werden und keinerlei Informationen über den Prozess, dessen Zustand oder ähnliches. Ausserdem ist die Kommunikation zwischen Prozessen mittels Pipes unidirektional, wobei die Möglichkeit besteht zwei Pipes zu öffnen und so Kommunikation in beide Richtungen zwischen Prozessen zu ermöglichen.

Wenn einer der Prozesse beendet wird, schließt der Betriebssystemkern die Pipe automatisch, das Lesen bzw. Schreiben von/in die Pipe schlägt fehl und der entsprechende Prozess kann dementsprechend reagieren.

Listing 4.2: Beispiel Pipes: Client-Server

```c
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(void) {
    int fd[2];
    pipe(fd);

    int pid = fork();

    if (pid == 0) {
        char str[] = "Text\n";

        close(fd[0]);
        write(fd[1], str, (strlen(str) + 1));
        exit(0);
    } else {
        char buf[80];

        close(fd[1]);
        read(fd[0], buf, sizeof(buf));
        printf("Empfangen: %s", buf);
    }

    return 0;
}
```

In Listing 4.2 werden mittels "fork" (3.1) zwei Prozesse erzeugt, welche über eine Pipe einen Text senden. Der empfangende Prozess gibt diesen Text aus. Der schreibende Prozess schließt den Deskriptor, welcher zum Lesen verwendet werden kann. Der lesende Prozess schließt den, der zum Schreiben verwendet werden kann. Das Betriebssystem schließt nach Beenden der Prozesse die übriggebliebenen Deskriptoren.

4.2. FIFO

Ein weiteres Konzept, welches in dieser Arbeit vorgestellt werden soll, ist die sogenannte "FIFO". FIFO ist eine Abkürzung für "First In First Out" und beschreibt eine "Pipe" (deutsch: "Röhre") über welche zwei Prozesse (unidirektional) kommunizieren können.

Einer der Prozesse öffnet dazu eine spezielle Datei welche als FIFO agieren soll und schreibt Daten in diese. Der lesende Prozess kann diese Daten nun abgreifen indem er die selbe Datei lesend öffnet.

In Folgendem wird der sendende Prozess mit "Client", der empfangende Prozess mit "Server" beschrieben.

In Listing 4.3 öffnet der Server eine FIFO mit dem Namen "fifo".

Listing 4.3: Beispiel FIFO: Server

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <sys/stat.h>
5  #include <unistd.h>
6  #include <linux/stat.h>
7
8  int main(void) {
9      FILE* fp;
10     char buf[80];
11     memset(buf, 0, sizeof(buf));
12
13     umask(0);
14     mknod("fifo", S_IFIFO | 0666, 0);
15
16     for (;;) {
```

```
17          fp = fopen("fifo", "r");
18          fgets(buf, 80, fp);
19          printf("Empfangen: %s\n", buf);
20          fclose(fp);
21      }
22
23      return(0);
24  }
```

Der Server ließt aus der FIFO und schreibt den gelesenen Text auf das Terminal. Der Client (Listing 4.4) wird vom Nutzer aufgerufen und schreibt den übergebenen Text in die FIFO hinein.

Listing 4.4: Beispiel FIFO: Client

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char *argv[])
5  {
6      FILE* fp;
7
8      if (argc != 2) {
9          printf("usage: fifo_client [string]\n");
10         exit(1);
11     }
12
13     if ((fp = fopen("fifo", "w")) == NULL) {
14         perror("fopen");
15         exit(1);
16     }
17
18     fputs(argv[1], fp);
19
20     fclose(fp);
21     return 0;
22 }
```

4.3. Message Queue

Eine weitere Form der Interprozesskommunikation sind die sogenannten "Message Queues".

Message Queues sind im Betriebssystemkern angelegte Verkettungen von Nachrichten. Message Queues können von Prozessen angelegt, befüllt und ausgelesen werden. Es gibt verschiedene Implementationen dieses Mechanismus: Die Implementation der System V Familie sowie die POSIX-Implementation.

4.3.1. System V

Der Linux-Kernel stellt dem Programmierer System V-artige Message Queues zur Verfügung. Diese können über eine Reihe von Systemaufrufen benutzt werden. System V Message Queues haben die Eigentschaft, dass die Länge der Nachrichten beliebig sein kann. Der Programmierer muss lediglich eine Struktur, welche mit einem "long" beginnt, definieren:

Listing 4.5: Message Buffer definition für System V

```
1 struct msgbuf {
2     long mtype; // Typ der Nachricht
3     char data[512]; // Daten, beliebig
4 };
```

([GMBW95, S. 33])

Das Feld "mtype" beschreibt den Typ der Nachricht, welche gesendet wird. Nach diesem Eintrag können beliebige Datentypen in der Struktur verwendet werden.

Der Linux-Kernel definiert eine Maximalgröße für Nachrichten in dem Systemheader "linux/msg.h", wie in Listing 4.6

Listing 4.6: Linux Kernel Maximalgröße für Nachrichten

```
1 #define MSGMAX 4056
```

Diese Maximalgröße schließt den Eintrag für den Typ der Nachricht mit ein ([GMBW95, S. 33]).

Zur Handhabung von Message Queues sind folgende Systemaufrufe verfügbar:

Name	msgget
Header	sys/types.h, sys/ipc.h, sys/msg.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	Message Queue Identifikation
Rückgabe im Fehlerfall	-1
Errno	EACCES, EEXIST, ENOENT, ENOMEM, ENOSPC
Parameter	key_t key, int msgflg

Tabelle 10.: Systemaufruf: msgget

Name	msgsnd
Header	sys/types.h, sys/ipc.h, sys/msg.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	0
Rückgabe im Fehlerfall	-1
Errno	EACCESS, EAGAIN, EFAULT, EIDRM, EINTR, EINVAL, ENOMEM
Parameter	int msgid, const void *msgp, size_tz msgsz, int msgflg

Tabelle 11.: Systemaufruf: msgsnd

Name	msgrcv
Header	sys/types.h, sys/ipc.h, sys/msg.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	Anzahl der kopierten Bytes
Rückgabe im Fehlerfall	-1
Errno	E2BIG, EACCES, EFAULT, EIDRM, EINTR, EINVAL, ENOMSG, ENOSYS
Parameter	int msgid, void *msgp, size_t msgsz, long msgtyp, int msgflg

Tabelle 12.: Systemaufruf: msgrcv

Name	msgctl
Header	sys/types.h, sys/ipc.h, sys/msg.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	0 mit IPC_STAT, IPC_SET und IPC_RMID, Index des höchsten verwendeten Eintrags der Kernel-internen Liste mit Message-Queue-Informationen mit IPC_INFO oder MSG_INFO
Rückgabe im Fehlerfall	-1
Errno	EACCES, EFAULT, EIDRM, EINVAL, EPERM
Parameter	int msgid, int cmd, struct msqid_ds *buf

Tabelle 13.: Systemaufruf: msgctl

4.3.2. POSIX

Neben den System V-artigen Message Queues stellt der Linux-Kernel zudem noch POSIX Message Queues zur Verfügung. Diese Unterscheiden sich zu den System V Message Queues in Interface und Funktionalität. So sind zum Beispiel POSIX Message Queues mit einer Funktionalität zur Benachrichtigung des Kommunikationspartners (Notifications) ausgerüstet, während System V diese Funktionalität nicht bietet. Zudem ist die Implementierung der POSIX Message Queues jünger als die der System V Message Queues, was eine größere Verbreitung letzterer nach sich zieht. Bei der Implementierung der POSIX Message Queues wurde aus den Fehlern, welche bei der Implementierung der System V gemacht wurden, gelernt.

Ein weiterer, sehr signifikanter Unterschied ist, dass die POSIX Message Queues als Thread-Safe gelten, während die Message Queues aus System V dies nicht bieten.

POSIX bietet mehrere Systemaufrufe zur Handhabung von Message Queues an, welche in folgendem erläutert werden sollen.

Name	mq_close
Header	mqueue.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	0
Rückgabe im Fehlerfall	-1
Errno	EBADF
Parameter	mqd_t mqdes

Tabelle 14.: Systemaufruf: mq_close

Name	mq_getattr
Header	mqueue.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	0
Rückgabe im Fehlerfall	-1
Errno	EBADF, EINVAL
Parameter	mqd_t mqdes, const struct mq_attr* newattr

Tabelle 15.: Systemaufruf: mq_getattr

Name	mq_notify
Header	mqueue.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	0
Rückgabe im Fehlerfall	-1
Errno	EBADF, EBUDY, EINVAL, ENOMEM
Parameter	mqd_t mqdes, const struct sigevent* sevp

Tabelle 16.: Systemaufruf: mq_notify

Name	mq_open
Header	mqueue.h
Rückgabetyp	mpd_t
Rückgabe im Erfolgsfall	Message Queue Deskriptor
Rückgabe im Fehlerfall	-1
Errno	EACCES, EEXIST, EINVAL, EMFILE, ENAMETOOLONG, ENFILE, ENOENT, ENOSPC
Parameter	const char* name, int oflag

Tabelle 17.: Systemaufruf: mq_open

Name	mq_receive
Header	mqueue.h
Rückgabetyp	ssize_t
Rückgabe im Erfolgsfall	Anzahl der Empfangenen Bytes
Rückgabe im Fehlerfall	-1
Errno	EAGAIN, EBADF, EINTR, EINVAL, EMSGSIZE, ETIMEDOUT
Parameter	mqd_t mqdesc, char* msdptr, size_t msglen, unsigned int * prio

Tabelle 18.: Systemaufruf: mq_receive

Name	mq_send
Header	mqueue.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	0
Rückgabe im Fehlerfall	-1
Errno	EAGAIN, EBADF, EINT, EINVAL, EMSGSIZE, ETIMEDOUT
Parameter	mqd_t mqdes, const char* msgptr, size_t len, unsigned int prio

Tabelle 19.: Systemaufruf: mq_send

Name	mq_setattr
Header	mqueue.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	0
Rückgabe im Fehlerfall	-1
Errno	EBADF, EINVAL
Parameter	mqd_t mqdes, const struct mq_attr* newattr, struct mq_attr* oldattr

Tabelle 20.: Systemaufruf: mq_setattr

Name	mq_timedreceive
Header	mqueue.h
Rückgabetyp	ssize_t
Rückgabe im Erfolgsfall	Anzahl der Empfangenen Bytes
Rückgabe im Fehlerfall	-1
Errno	EAGAIN, EBADF, EINTR, EINVAL, EMSGSIZE, ETIMEDOUT
Parameter	mqd_t mqdesc, char* msdptr, size_t msglen, unsigned int * prio, const struct timespec* abstimeout

Tabelle 21.: Systemaufruf: mq_timedreceive

Name	mq_timedsend
Header	mqueue.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	0
Rückgabe im Fehlerfall	-1
Errno	EAGAIN, EBADF, EINT, EINVAL, EMSGSIZE, ETIMEDOUT
Parameter	mqd_t mqdes, const char* msgptr, size_t len, unsigned int prio, const struct timespec* abstimeout

Tabelle 22.: Systemaufruf: mq_timedsend

Name	mq_unlink
Header	mqueue.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	0
Rückgabe im Fehlerfall	-1
Errno	EACCES, ENAMETOOLONG, ENOENT
Parameter	const char* name

Tabelle 23.: Systemaufruf: mq_unlink

Eine Besonderheit, welche bei POSIX Message Queues hervorzuheben ist, ist die Identifikation einer Message Queue über einen Namen. Dieser Name muss mit einem "/" beginnen und darf von maximal 255 Zeichen gefolgt sein, von welchem keines ein "/" Zeichen sein darf. Übergeben wird der Name als Null-terminierter "char" Puffer. Sobald ein Prozess den entsprechenden Namen kennt, kann er die damit assoziierte Message Queue öffnen und auf ihr schreiben oder lesen. Die Aufrufe, welche mit Namen arbeiten sind "mq_open" (17) und "mq_unlink" ((23).

Listing 4.7: Beispiel: POSIX Message Queue

```
1 #include <mqueue.h>
2 #include <stdlib.h>
3 #include <stdio.h>
4 #include <errno.h>
5 #include <string.h>
6
7 #define MQNAME "/sometestqueue"
8
9 #define MQ_S (8192)
```

```
10
11 int main() {
12     mqd_t             mqdes;
13     struct mq_attr   attr = { .mq_maxmsg = 300, .
       mq_msgsize = MQ_S, .mq_flags = 0 };
14
15     // Now open a queue with the default attribute
       structure
16     mqdes = mq_open(MQNAME, O_RDWR | O_CREAT, 0664, 0);
17     if (mqdes == -1) {
18         perror("mq_open");
19         exit(1);
20     }
21
22     mq_unlink(MQNAME);
23
24     char wbuf[MQ_S] = "hallowelt";
25     int mqs = mq_send(mqdes, wbuf, MQ_S, 0);
26     if (mqs == -1) {
27         perror("mq_send()");
28     }
29
30     char rbuf[MQ_S];
31     memset(rbuf, 0, sizeof(rbuf));
32     if (mq_receive(mqdes, rbuf, MQ_S, 0) == -1) {
33         perror("mq_receive");
34     }
35
36     printf("Got␣Message:␣%s\n", rbuf);
37
38     mq_close(mqdes);
39     return 0;
40 }
```

Listing Listing 4.7 zeigt eine Beispielimplementation mit einer einfachen POSIX Message Queue, welche in einem Prozess erzeugt wird und Daten an den selben Prozess sendet. Es findet in diesem Beispiel keine Interprozesskommunikation statt.

4.3.3. Bewertung

Während die System V Implementation der Message Queue unter Linux mehr und interessantere Features bietet, fühlt sich das Interface der POSIX Message Queues sehr viel angenehmer an. Eine Funktion, welche eine Funktionalität abbildet, hat einen UNIX-artigen Charakter, was mich persönlich mehr anspricht.

Die Vorgabe, dass für eine System V Message Queue eine Datenstruktur definiert werden muss welche verpflichtend mit einem "long" beginnt, wirkt unangenehm und umständlich. Der Systemaufruf, der dann die Größe dieser Datenstruktur als Übergabewert erwartet, allerdings abzüglich der Größe des "long" wirkt einfach nur wie eine fehlerhafte Spezifikation.

4.4. Synchronisation von Zugriffen auf Systemressourcen

Da gleichzeitige Zugriffe auf Systemressourcen zu Problemen führen können, wie zum Beispiel sogenannte Race-Conditions, müssen diese Zugriffe synchronisiert werden. Diese Probleme bestehen nicht nur bei Zugriffen aus verschiedenen Threads, sondern auch bei Zugriffen aus verschiedenen Prozessen, zum Beispiel auf die gleiche Datei. Zur Vermeidung dieser Fehler stellt das Betriebssystem verschiedene Mechanismen zur Verfügung, welche in System V und POSIX-Standard definiert sind.

4.4.1. Semaphoren

Semaphoren sind eine der beiden Mittel, welche das Betriebssystem zur synchronisation von Zugriffen auf Ressourcen dem Programmierer zur Verfügung stellt. Im Abstrakten Sinne kann man eine Semaphore als Zähler verstehen, welche einen Atomaren Zugriff auf ihren Wert und warten auf bestimmte Werte zulässt.

> A semaphore is a kernel-maintained integer whose value is restricted to being greater than or equal to 0. Various operations (i.e., system calls) can be performed on a semaphore [...]

[Ker10, S. 965]

Zum Beispiel kann mittels einer Semaphore definiert werden, dass eine Systemressource erste benutzt werden darf sobald der Wert der Semaphore Null ist. Wird eine Systemressource von einem Prozess benötigt, wird der Wert der Semaphore um Eins

erhöht, wird die Systemressource nicht mehr benötigt, wird der Wert wieder um Eins heruntergesetzt. Alle anderen Prozesse können den Wert der Semaphore erst erhöhen wenn der Wert wieder Null ist. Dadurch ist garantiert, dass zu einem Zeitpunkt nur ein Prozess eine Ressource benutzt.

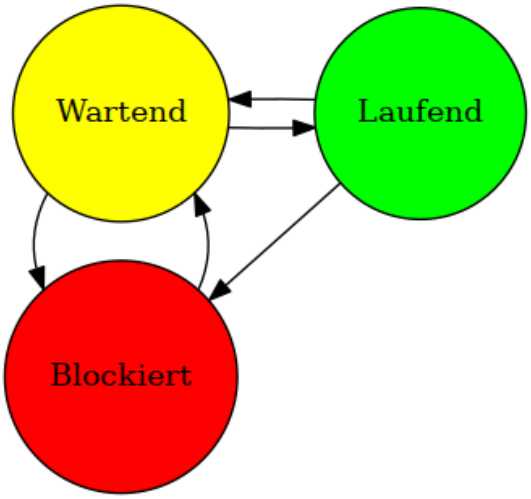

Abbildung 1.: Zustände eines Prozesses

Abbildung 1 beschreibt die Zustände, in welchen ein Prozess sein kann. Semaphoren sind hier an den Übergängen der Prozesszustände angesiedelt. Ein Prozess kann von einer Semaphore blockiert sein, der Übergang zum Wartenden Zustand wird durchgeführt wenn die Semaphore, auf welche der Prozess wartet, frei wird. Bekommt der Prozess die Semaphore, so begibt er sich in den Zustand "Laufend". Muss der Prozess nun auf Systemressourcen warten, wird er wieder auf den Wartenden Zustand zurück gesetzt. In den Blockierenden Zustand kommt er wenn er wieder blockiert wird.

Ein Spezialfall ist der Übergang vom wartenden Zustand zum blockierenden Zustand. Dieser Fall tritt ein, wenn eine Semaphore frei wird und mehrere Prozesse versuchen die Semaphore zu bekommen. In diesem Fall sind alle Prozesse kurzzeitig "Wartend", bis der Betriebssystemkern entscheidet, welcher Prozess die Semaphore bekommt. Dieser Prozess wird in den "Laufenden" Zustand versetzt, alle anderen Prozesse werden wieder in den "Blockiert"-Zustand überführt. Welcher Prozess letztendlich die Semaphore bekommt ist implementierungsabhängig.

4.4.2. System V Semaphoren

Operationen, welche eine System V-Semaphore unterstützt sind

- Setzen auf einen absoluten Wert
- Eine Zahl aufaddieren
- Eine Zahl subtrahieren
- Warten bis die Semaphore den Wert Null erreicht

Letztere zwei Operationen blockieren den aufrufenden Prozess: Wenn der Wert der Semaphore bei einer Subtraktion kleiner als Null werden würde, wird der Prozess der die Operation ausführen möchte, blockiert. Ausserdem wird der Prozess, der darauf wartet dass eine Semaphore Null wird, blockiert bis dieser Zustand eintritt.

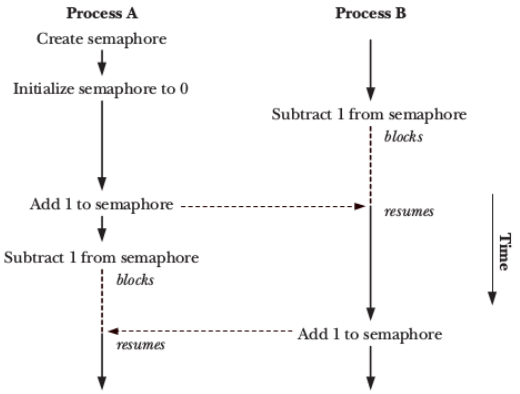

Abbildung 2.: Benutzung einer Semaphore um zwei Prozesse zu synchronisieren [Ker10, S. 966]

In beiden Fällen wird der aufrufende Prozess so lange blockiert bis ein anderer Prozess den Wert der Semaphore verändert, sodass der aufrufende Prozess weiter arbeiten kann. Der Betriebssystemkern weckt den blockierten Prozess dann auf.

In Listing B.6 wird dargestellt, wie eine System V-Semaphore benutzt werden kann.

Listing 4.8: Beispiel System V Semaphore: Operationen

```
1 static struct sembuf sop_lock[2] = {
2     { 0 /* Nr */, 0 /* Op (wait == 0) */, 0  /* Flags (none) */ },
3     { 0 /* Nr */, 1 /* Op (+1) */, SEM_UNDO /* Flags (undo changes on crash) */ },
4 };
5
6 static struct sembuf sop_unlock[2] = {
7     { 0 /* Number */, -1 /* Op (sub one) */, IPC_NOWAIT /* Flags: Do not block */ },
8 };
```

```
9
10 int semid = -1; // id
```

Listing 4.8 zeigt, wie die Operationen, welche die Semaphore durchführen soll, definiert werden. Diese werden im folgenden benutzt um die Semaphore zu sperren und zu entsperren. Zudem wird in diesem Codeabschnitt eine (der Einfachheit wegen globale) Variable "semid" definiert, welche die Identifikation der Semaphore beinhaltet.

Listing 4.9: Beispiel System V Semaphore: Sperren

```
1 void my_lock(void) {
2     if (semid < 0 && ((semid = semget(123456L, 1, ↩
        IPC_CREAT | 0666)) < 0)) {
3         perror("Semget error");
4     }
5
6     if (semop(semid, sop_lock, 2) < 0) {
7         perror("semop lock error");
8     }
9 }
```

In Listing 4.9 wird eine Funktion definiert, welche später benutzt werden kann um die Semaphore zu sperren. Dementsprechend wird in Listing 4.10 eine Funktion definiert, welche benutzt werden kann um die Semaphore zu entsperren.

Listing 4.10: Beispiel System V Semaphore: Entsperren

```
1 void my_unlock(void) {
2     if (semop(semid, sop_unlock, 1) < 0) {
3         perror("semop unlock error");
4     }
5 }
```

Listing 4.11 zeigt die Main-Funktion des Programmes. Hier wird die Semaphore zwei mal gesperrt, ein Text ausgegeben und die Semaphore dann wieder entsperrt. Zwischen diesen Aktionen wird das Programm immer wieder in den Wartezustand versetzt.

Listing 4.11: Beispiel System V Semaphore: Main

```
1 int main(void) {
2     pid_t i = getpid();
3     int j;
```

```
4       my_lock();
5       {
6           sleep(1);
7           printf("Thread running: %i\n", i);
8       }
9       my_unlock();
10
11      sleep(1);
12
13      my_lock();
14      {
15          sleep(1);
16          printf("Thread running: %i\n", i);
17      }
18      my_unlock();
19
20      semctl(semid, 0, IPC_RMID, 0);
21      return EXIT_SUCCESS;
22  }
```

Wie in Listing 4.11 zu sehen ist, werden in dem Programm selbst keine weiteren Prozesse gestartet. Dies ist zur Verdeutlichung dass Semaphoren Systemressourcen sind und über mehrere Prozesse hinweg benutzt werden können. Startet man mehrere Instanzen des Programms (zum Beispiel in Listing 4.12 gezeigt), wird immer nur eine Instanz des Programms einen Text ausgeben, da die Semaphore benutzt wird um den Zugriff auf den Ausgabepuffer zu synchronisieren.

Listing 4.12: Starten mehrerer Instanzen

```
1 ./a.out & ./a.out & ./a.out
```

4.4.3. POSIX Semaphoren

Bei POSIX Semaphoren muss unterschieden werden zwischen den beiden Arten von Semaphoren die es im POSIX-Standard gibt:

- Benamte Semaphoren
- Unbenamte Semaphoren

Dabei existiert die unbenamte Semaphore im Addressbereich des Prozesses, während die benamte Semaphore im System angelegt wird und von mehreren Prozessen benutzt werden kann. Die beiden Arten von Semaphoren müssen auch auf unterschiedliche Weiße angelegt und gelöscht werden, in der Benutzung unterscheiden sich die Funktionen allerdings nicht.

4.4.3.1. Anlegen und Löschen

In folgendem soll beschrieben werden wie POSIX-Semaphoren angelegt und gelöscht werden. Da das Anlegen und Löschen einer benamten Semaphore anders funktioniert als das einer unbenamten, werden jeweils verschiedene Funktionen benötigt.

sem_open

Dieser Aufruf wird benutzt um eine benamte Semaphore anzulegen.

Name	sem_open
Header	semaphore.h, sys/stat.h, fcntl.h
Rückgabetyp	sem_t*
Rückgabe im Erfolgsfall	Semaphore
Rückgabe im Fehlerfall	NULL
Errno	EACCESS, EEXIST, EINVAL, EMFILE, ENAMETOOLONG, ENFILE, ENOENT, ENOMEM
Parameter	const char* name, int oflag

Tabelle 24.: Systemaufruf: sem_open

sem_unlink

Dieser Aufruf wird benutzt um eine benamte Semaphore aus dem System zu entfernen. Die Semaphore existiert weiter und kann weiterhin benutzt, allerdings nicht von anderen Prozessen gefunden werden.

Name	sem_unlink
Header	semaphore.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	0
Rückgabe im Fehlerfall	-1
Errno	ENAMETOOLONG, ENOENT
Parameter	const char* name

Tabelle 25.: Systemaufruf: sem_unlink

sem_close

Dieser Aufruf wird benutzt um eine Semaphore zu löschen.

Name	sem_close
Header	semaphore.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	0
Rückgabe im Fehlerfall	-1
Errno	EINVAL
Parameter	sem_t* sem

Tabelle 26.: Systemaufruf: sem_close

sem_init

Dieser Aufruf wird benutzt um eine unbenamte Semaphore anzulegen. Zu bemerken ist, dass der Aufruf der Semaphore einen Initialwert übergeben kann.

Name	sem_init
Header	semaphore.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	0
Rückgabe im Fehlerfall	-1
Errno	EINVAL, ENOSYS
Parameter	sem_t* sem, int pshared, unsigned int value

Tabelle 27.: Systemaufruf: sem_init

sem_destroy

Dieser Aufruf kann benutzt werden um eine unbenamte Semaphore zu zerstören. Zu bemerken ist, dass die Zerstörung einer Semaphore welche aktuell andere Prozessen blockiert in undefinitiertem Verhalten resultiert. Eine Semaphore welche mit "sem_destroy" (4.4.3.1) zerstört wurde kann mit "sem_init" (4.4.3.1) reinitialisiert werden.

Name	sem_destroy
Header	semaphore.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	0
Rückgabe im Fehlerfall	-1
Errno	EINVAL
Parameter	sem_t* sem

Tabelle 28.: Systemaufruf: sem_destroy

4.4.3.2. Benutzen der Semaphoren

Bei der Benutzung unterscheiden sich unbenamte und benamte Semaphoren nicht, wie durch folgende Auflistung von Systemaufrufen verdeutlicht wird.

sem_wait

Dieser Systemaufruf existiert in verschiedenen Varianten:

- "sem_wait"
- "sem_trywait"
- "sem_timedwait"

Welche ähnliches Verhalten abbilden.

Name	sem_wait
Header	semaphore.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	0
Rückgabe im Fehlerfall	-1
Errno	EINTR, EINVAL, EAGAIN, ETIMEDOUT
Parameter	sem_t*

Tabelle 29.: Systemaufruf: sem_wait

Name	sem_trywait
Header	semaphore.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	0
Rückgabe im Fehlerfall	-1
Errno	EINTR, EINVAL, EAGAIN, ETIMEDOUT
Parameter	sem_t*

Tabelle 30.: Systemaufruf: sem_trywait

Name	sem_timedwait
Header	semaphore.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	0
Rückgabe im Fehlerfall	-1
Errno	EINTR, EINVAL, EAGAIN, ETIMEDOUT
Parameter	sem_t*, const struct timespec*

Tabelle 31.: Systemaufruf: sem_timedwait

Der Aufruf "sem_wait" (29) versucht den Wert der Semaphore um 1 herabzusetzen. Sollte dies nicht möglich sein, weil der Wert der Semaphore bereits Null ist, blockiert die Funktion den aufrufenden Prozess so lange bis der Wert der Semaphore Eins ist.

Der Aufruf "sem_trywait" (30) verhält sich wie "sem_wait" (29), blockiert allerdings nicht falls der Wert der Semaphore nicht sofort verändert werden kann und kehrt stattdessen mit einem Fehler zurück.

Der Aufruf "sem_timedwait" (31) blockiert nur eine bestimmte Zeit lang und kehrt mit einem Fehler zurück sollte die Semaphore nicht verändert sein.

sem_post

Name	sem_post
Header	semaphore.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	0
Rückgabe im Fehlerfall	-1
Errno	EINVAL, EOVERFLOW
Parameter	sem_t*

Tabelle 32.: Systemaufruf: sem_post

Der "sempost" (4.4.3.2) erhöht den Wert der Semaphore um eins und hebt damit die Blockade auf.

4.4.3.3. Szenarien

In Abbildung 4 ist dargestellt, wie Semaphoren dazu benutzt werden können um Prozesse in ihren Schreib- und Lesezugriffen zu synchronisieren. Dabei berechnet der eine Prozess Daten, welche er dann in eine Variable schreibt (in Abbildung 4 nicht abgebildet). Prozess 2 ließt diese Daten dann und verarbeitet sie weiter. Diese Weitergabe von Daten wird in der Programmierung oft verwendet um Prozesse schlank und einfach zu halten und mit einzelnen kleineren Komponenten größere Berechnungen anzustellen, wobei jeder Prozess nur eine Teilberechnung durchführt [1].

Ein weiteres Beispiel zur Benutzung von Semaphoren sind die sogenannten "Dining Philosophers". Das Szenario ist, dass eine Gruppe von Philosophen um einen runden Tisch sitzen und vor sich einen Teller mit Spagetti haben. Neben jedem Philosophen liegen zwei Gabeln auf dem Tisch, insgesamt also gleich viele Gabeln wie Philosophen. Die Philosophen philosophieren und essen abwechselnd. Ein Philosoph benötigt zum essen seiner Spagetti beide Gabeln die neben ihm liegen, wenn ein Philosoph anfängt zu essen nimmt er erst die linke und dann die rechte Gabel, isst dann und legt die Gabeln dann wieder in gleicher Reihenfolge ab. Dies kann nur funktionieren wenn die Gabeln verfügbar sind. Sollte eine Gabel nicht verfügbar sein, wartet der Philosoph bis sie verfügbar ist.

[1] Es muss darauf geachtet werden dass die Berechnungen welche innerhalb der Prozesse durchgeführt werden "teuer" genug sind, dass sich eine Prozesssynchronisation "lohnt". Ist der Austausch der Daten rechenintensiver als die Berechnung selbst, oder rechnet ein Prozess signifikant länger als der andere, sollte auf eine andere Methode zur Berechnung zurückgegriffen werden um die Performanz zu gewährleisten.

Sobald alle Philosophen gleichzeitig die Linke Gabel nehmen ist das System blockiert. Alle Philosophen warten nun darauf die Rechte Gabel zu bekommen, wobei dies nie geschehen kann, da diese vom Sitznachbarn benutzt wird, welcher wiederrum auf seine rechte Gabel wartet (siehe [Zel08, S. 14]).

Dieses Problem kann mit verschiedenen Herangehensweißen mittels Semaphoren (oder auch Mutexen) gelöst werden.

Eine relativ einfache Lösung dieses Problems ist, die Anzahl der gleichzeitig essenden Philosophen auf $n - 1$ zu beschränken, es darf also ein Philosoph nicht essen, wenn alle anderen essen. Dies beschränkt zwar den Spagetti-Durchsatz, verhindert aber einen Deadlock ([Zel08, S. 14 ff.]).

4.4.4. POSIX Mutexes

Bei POSIX Mutexes handelt es sich um ein vereinfachtes Konzept, welches ausschließtlich zur synchronisation von Threads verwendet werden kann. Zur Verwendung von POSIX-Mutexen sind verschiedene Systemaufrufe verfügbar, von denen nicht alle hier beschrieben werden sollen.

Name	pthread_mutex_init
Header	pthead.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	0
Rückgabe im Fehlerfall	ERRNO
Errno	EAGAIN, ENOMEM, EPERM, EBUSY, EINVAL
Parameter	pthread_mutex_t*

Tabelle 33.: Systemaufruf: pthread_mutex_init

Name	pthread_mutex_destroy
Header	pthead.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	0
Rückgabe im Fehlerfall	ERRNO
Errno	EBUSY, EINVAL
Parameter	pthread_mutex_t*

Tabelle 34.: Systemaufruf: pthread_mutex_destroy

POSIX-Mutexe können mit Attributen ausgestattet werden. Diese definieren das Verhalten des Mutex. So kann eine Mutex verschiedene Typen haben:

- Rekursiv
- Deadlock-Detect
- Deadlock
- ...

Es sind Robustheits-Einstellungen oder Priorisierung möglich. Detailierte Erklärungen zu Attributen von Mutexen sind in [Ope15] nachzulesen.

Ein einfaches Beispiel zur Synchronisation von Thread-Abläufen mittels Mutexen wird in Listing B.2 gezeigt.

4.4.5. Bewertung

Während die System V Semaphore sehr viel mächtiger wirkt, fühlt sich die Benutzung der POSIX-artigen Semaphore sehr viel angenehmer an. Die Aufteilung in nur drei wesentliche Funktionen bei System V ist zwar typisch, allerdings fühlt sie sich auch umständlich und unsauber an. Das Interface der POSIX Semaphore wirkt durch die Aufsplittung in mehrere Funktionen schlanker, sauberer.

Generell finde ich das Konzept der Semaphoren und Mutexes sehr interessant, allerdings möchte ich als Entwickler nicht unbedingt etwas mit diese Funktionalitäten zu tun haben. Ich erwarte von jeder bekannten Programmiersprache, dass die Standardbibliothek der Sprache Datenstrukturen (Arrays, HashMaps, Bäume) zur Verfügung stellt, die bereits Mutlitasking-sicher implementiert sind.

4.5. Shared Memory

Beim Shared Memory handelt es sich um eine Technik die es ermöglicht, größere Datenmengen zwischen Prozessen zu teilen. Dazu wird ein spezieller Speicherbereich im Betriebssystemkern reserviert, welcher dann von mehreren Prozessen benutzt werden kann als würde er im Prozesseigenen Speicherbereich existieren. Dabei ist der Lese/Schreibzugriff auf den Speicherbereich selbst nicht synchronisiert.

Wie in Abbildung 3 veranschaulicht, können die Prozesse allerdings nicht auf den

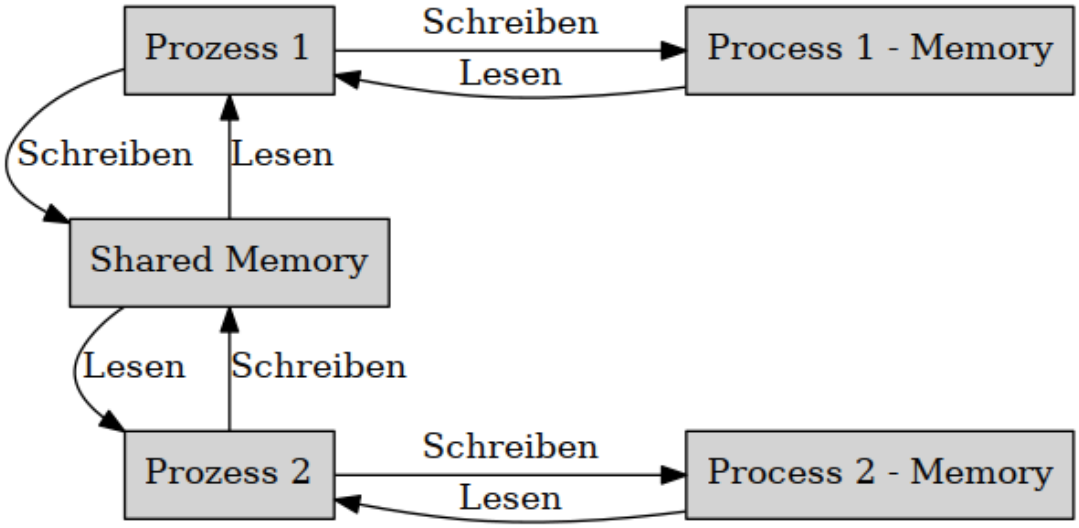

Abbildung 3.: Prozesse und Speicher, Shared Memory

privaten Speicherbereich des jeweils anderen Prozesses zugreifen. Shared Memory ist nicht auf zwei Teilnehmer beschränkt, es können auch mehrere Prozesse den gleichen Shared Memory benutzen.

Selbstverständlich gibt es auch beim Shared Memory zwei verschiedene Implementationen.

4.5.1. System V

Der Linux-Kernel exportiert System V Funktionen zur Handhabung von Shared Memory. Diese sind anhand des Beispiels aus 4.5.1.1 erklärt.

Name	shmget
Header	sys/ipc.h, sys/shm.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	Shared memory ID / Filedeskriptor
Rückgabe im Fehlerfall	-1
Errno	EACCES, EEXIST, EINVAL, ENFILE, ENOENT, ENOMEM, ENOSPC, EPERM
Parameter	key_t key, size_t size, int flags

Tabelle 35.: Systemaufruf: shmget

Name	shmctl
Header	sys/ipc.h, sys/shm.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	>= 0
Rückgabe im Fehlerfall	-1
Errno	EACCES, EFAULT, EIDRM, EINVAL, ENOMEM, EOVERFLOW, EPERM
Parameter	int shmid, int cmd, struct shmid_ds*

Tabelle 36.: Systemaufruf: shmctl

Name	shmat
Header	sys/types.h, sys/shm.h
Rückgabetyp	void*
Rückgabe im Erfolgsfall	Adresse des Shared Memory
Rückgabe im Fehlerfall	(void*) -1
Errno	EACCES, EIDRM, EINVAL, ENOMEM
Parameter	int shmid, const void* shmaddr, int shmflg

Tabelle 37.: Systemaufruf: shmat

Name	shmdt
Header	sys/types.h, sys/shm.h
Rückgabetyp	void*
Rückgabe im Erfolgsfall	Adresse des Shared Memory
Rückgabe im Fehlerfall	(void*) -1
Errno	EINVAL
Parameter	const void* shmaddr

Tabelle 38.: Systemaufruf: shmdt

4.5.1.1. Beispiel: Shared Memory als Textspeicher

In Listing B.7 wird aufgezeigt wie Shared Memory benutzt werden kann. Das Programm legt einen Shared Memory an, in welchen dann ein Text, welcher beim Aufruf des Programmes übergeben wurde, geschrieben wird. Wird das Programm noch einmal aufgerufen, diesmal ohne einen Parameter, wird der Speicher des Shared Memory ausgelesen und der Text auf der Kommandozeile ausgegeben.

Listing 4.13: Beispiel Shared Memory: Helfermacros

```
#define SHM_SIZE (1024)
#define perror_exit(x) do { perror((x)); exit(1); } while (0)
#define ifpexit(cnd, x) do { if ((cnd)) { perror_exit((x)) } } while (0)
```

Die in Listing 4.13 aufgezeigten Programmzeilen definierten Helfer, welche in folgenden Listings benutzt werden um den Quellcode übersichtlicher zu gestalten.

Listing 4.14: Beispiel Shared Memory: Initialisieren

```
    ifpexit(((key = ftok("sysv_shm_simple.c", 'R')) == -1), "ftok");
    ifpexit(((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT)) == -1), "shmget");
```

In Listing 4.14 wird zunächst ein sogenannter "Key" für den Shared Memory angelegt. Dieser wird von der System V-Schnittstelle benötigt um den Speicher zu identifizieren. Sollte dies fehlschlagen, wird das Programm mit einer Fehlermeldung beendet.

In der nächsten Zeile wird nun ein Shared Memory "Segment" vom Betriebssystem angefordert. Dieses wird mittels dem soeben generierten "Key" assoziiert. Es werden 1024 Bytes vom Betriebssystemkern angefordert, die Berechtigungen werden auf 0644 gesetzt. Sollte dies fehlschlagen, wird das Programm mit einer Fehlermeldung beendet.

Listing 4.15: Beispiel Shared Memory: Anbinden

```
    data = shmat(shmid, (void *)0, 0);
    ifpexit((data == (char *)(-1)), "shmat");
```

Listing 4.15 zeigt, wie der soeben angelegte Speicher nun angebunden wird. Sollte diese Operation fehlschalgen, wird das Programm mit einer Fehlermeldung beendet. Der Shared Memory wird nicht gelöscht.

Listing 4.16: Beispiel Shared Memory: Benutzen

```
    if (argc == 2) {
        printf("writing to segment: \"%s\"\n", argv[1]);
        strncpy(data, argv[1], SHM_SIZE);
    } else {
        printf("segment contains: \"%s\"\n", data);
    }
```

Wie Listing 4.15 zeigt, kann der Speicher nun wie gewohnt benutzt werden. In diesen Zeilen wird die Funktionalität des Programmes implementiert.

Listing 4.17: Beispiel Shared Memory: Benutzen

```
1     ifpexit((shmdt(data) == -1), "shmdt");
2
3     return   EXIT_SUCCESS;
```

Zuletzt wird in Listing 4.17 die Anbindung des Speichers nun wieder entfernt. Sollte dies Fehlschlagen wird das Programm ebenfalls mit einer Fehlermeldung beendet.

Zu Bemerken ist, dass der Speicher weiter existiert, obwohl das Programm bereits beendet wurde. Shared Memory muss explizit gelöscht werden, ansonsten existiert er im Betriebssystemkern weiter. Der Shared Memory, welcher von diesem Programm angelegt wurde, kann über die Kommandozeile mittels

Listing 4.18: Löschen von Shared Memory über die Kommandozeile

```
1 ipcs
2 ipcrm shm <ID>
```

gelöscht werden. Dabei dient der erste Aufruf ("ipcs") um die Identifikation des Shared Memory herauszufinden und der zweite Aufruf um diesen zu löschen.

Der volle Programmtext ist in Listing B.7 gelistet.

4.5.1.2. Beispiel: Shared Memory, synchronisiert

Da der Zugriff auf Shared Memory nicht vom Betriebssystem synchronisiert wird, was durchaus in einigen Anwendungsfällen gewünscht sein kann, kann es vorkommen dass zwei Prozesse gleichzeitig in den gleichen Speicherbereich des Shared Memory schreiben. Da dies zu inkonsistenten Daten führt, ist dies nicht erwünscht. Um diese sogenannten "Race-Conditions" zu vermeiden, muss der Shared Memory zwischen den Prozessen synchronisiert werden. In dieser Arbeit wurden bereits Techniken vorgestellt, welche zur Synchronisation von Zugriff auf Ressourcen benutzt werden können (4.4.2, 4.4.3, 4.4.4).

In folgendem Beispiel wird eine System V-Semaphore (4.4.2) verwendet um den Zugriff auf den Shared-Memory des Programmes zu synchonisieren. Das Programm legt zunächst einen Shared Memory an und splittet sich dann auf. Der Eltern-Teil des Programmes schreibt nun die per Kommandozeile übergebenen Parameter einen nach

dem anderen in den Shared Memory. Der Kind-Teil des Programmes ließt diesen Text aus und gibt ihn auf der Kommandozeile aus. Beide Programmteile lassen den Shared Memory vor Beendigung los, löschen ihn aber nicht. Das heisst, dass der Nutzer wie schon in 4.5.1.1 den Shared Memory von Hand löschen muss (Listing 4.18).

Der komplette Quelltext des Programmes ist unter Listing B.8 aufgeführt.

Listing 4.19: Beispiel Synchronisation von Shared Memory: Helfermacros

```
1 #define SHM_SIZE (1024)
2
3 #define print_exit(x) do { \
                                \
4         printf("Errno: %s, "x"\n", strerror(errno)); \
               \
5         exit(1); \
                                                \
6     } while (0)
7
8 #define perror_exit(x) do { perror((x)); exit(1); } \
    while (0)
9
10 #define ifprintexit(cnd, x) do { if ((cnd)) { \
    print_exit(x); } } while (0)
11 #define ifpexit(cnd, x) do { if ((cnd)) { perror_exit(x\
    ); } } while (0)
```

In Listing 4.19 werden zunächst Makros definiert, welche den folgenden Programmtext vereinfachen.

Danach werden, wie auch schon in Listing 4.8 Operationen zum Sperren und Entsperren der Semaphore definiert.

Listing 4.20: Beispiel Synchronisation von Shared Memory: Shared Memory Struct

```
1 struct sshm { // shared memory, synchronized via \
    semaphore
2     int semid;
3     key_t key;
4     int shmid;
5     void* shm;
6 };
```

Mit der in Listing 4.20 definierten Struktur wird eine Semaphore semantisch an einen Shared Memory gebunden und diesem zugeordnet. In den Zeilen 40 bis 79 (siehe Listing B.8) sind Funktionen definiert um das Arbeiten mit dieser Struktur zu vereinfachen.

Die Main-Funktion des Programmes (Zeile 81 bis 128 in Listing B.8) bildet die Funktionalität des Programmes ab. Hier wird der Shared Memory zunächst initialisiert, wonach das Programm sich aufsplittet und der Eltern-Teil des Programmes damit beginnt, die Kommandozeilenargumente in den Shared Memory zu schreiben. Dabei wird immer nur ein Kommandozeilenargument geschrieben und der Text im Shared Memory bei jedem Schreibzyklus verlängert. Der Kind-Teil des Programmes ließt diesen immer wieder aus und gibt den Text dann auf der Kommandozeile aus. Der Eltern-Teil des Programmes beendet sich, sobald alle Argumente in den Shared-Memory geschrieben sind. Der Kind-Teil des Programmes beendet sich sobald der Text im Shared Memory sich in seiner Länge nicht mehr verändert hat.

4.5.2. POSIX

POSIX bietet ebenfalls Funktionen zur Benutzung von Shared Memory an. Die grundlegenden Funktionen welche POSIX bereitstellt sind in folgendem aufgeführt und kurz erklärt.

4.5.2.1. shm_open

Name	shm_open
Header	sys/mman.h, sys/stat.h, fcntl.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	Dateideskriptor des Shared Memory
Rückgabe im Fehlerfall	-1
Errno	EACCES, EECIST, EINVAL, EMFILE, ENAMETOOLONG, ENFILE, ENOENT
Parameter	const char* name, int oflag, mode_t mode

Tabelle 39.: Systemaufruf: shm_open

Mittels dieser Funktion wird ein Shared-Memory angelegt und geöffnet. Die Funktion verhält sich dabei analog zu "open" (2.3) und erwartet als ersten Parameter einen Namen, welcher zur Identifikation des Shared Memory benutzt wird. Mittels

des "oflag"-Parameters können Optionen für den zu erzeugenden Shared Memory angegeben werden, zum Beispiel dass dieser nur Lesbar sein soll.

4.5.2.2. ftruncate

Name	ftruncate
Header	unistd.h, sys/types.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	0
Rückgabe im Fehlerfall	-1
Errno	EACCES, EFAULT, EFBIG, EINTR, EINVAL, EIO, EISDIR; ELOOP, ENAMETOOLONG, ENOENT, ENOTDIR, EPERM, EROFS, ETXTBSY, EBADF, EINVAL
Parameter	int fd, off_t length

Tabelle 40.: Systemaufruf: ftruncate

Dieser Aufruf ist eine Variante des Aufrufes "truncate" welcher sich gleich verhält, allerdings einen Pfad einer Datei als erstes Argument erwartet.

Mittels des "ftruncate" (??) kann die Länge des Shared Memory konfiguriert werden. Wird der Speicher verkleinert, sind die Daten des Bereiches welcher entfernt wird, verloren. Wird der Speicher vergrößert, wird dieser mit 0 befüllt.

4.5.2.3. mmap

Name	mmap
Header	sys/mman.h
Rückgabetyp	void*
Rückgabe im Erfolgsfall	Pointer zum gemappten Speicherbereich
Rückgabe im Fehlerfall	(void*) -1 == MAP_FAILED
Errno	EACCES, EAGAIN, EBADF, EINVAL, ENFILE, ENODEV, ENOMEM, EPERM, ETXTBSY, EOVERFLOW
Parameter	void* addr, size_t length, int prot, int flags, int fd, off_t offset

Tabelle 41.: Systemaufruf: mmap

Dieser Aufruf wird benutzt um den angelegten Speicherbereich in den Speicherbereich des Prozesses einzubinden (zu "mappen").

Siehe auch 4.5.3.1.

4.5.2.4. munmap

Name	mmunmap
Header	sys/mman.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	0
Rückgabe im Fehlerfall	-1
Errno	EACCES, EAGAIN, EBADF, EINVAL, ENFILE, ENODEV, ENOMEM, EPERM, ETXTBSY, EOVERFLOW
Parameter	void* addr, size_t length

Tabelle 42.: Systemaufruf: mmunmap

Dieser Aufruf ist das Gegenstück zu "mmap" (4.5.2.3) und wird benutzt um den Shared Memory wieder aus dem Speicherbereich des Prozesses zu entfernen.

Siehe auch 4.5.3.1.

4.5.2.5. shm_unlink

Name	shm_unlink
Header	sys/mman.h, sys/stat.h, fcntl.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	0
Rückgabe im Fehlerfall	-1
Errno	EACCES, ENAMETOOLONG, ENOENT
Parameter	const char* name

Tabelle 43.: Systemaufruf: shm_unlink

Dieser Aufruf wird benutzt um den Speicher des Shared Memory "loszulassen".

4.5.2.6. Beispiel: Shared Memory als Textspeicher

In Listing B.9 wird das Beispiel aus 4.5.1.1 mittels POSIX-Funktionen repliziert.

Listing 4.21: Beispiel Shared Memory: Helfermacros

```
1 #define SHM_SIZE (1024)
2 #define NAME ("/SHARED_MEM_EXAMPLE_SIMPLE")
```

Zunächst werden in Listing 4.21 Markos definiert, welche die Größe und den Namen des anzulegenden Shared Memories definieren.

Listing 4.22: Beispiel Shared Memory: Öffnen

```
1     int i      = 0;
2     void* ptr = NULL;
3     int shm    = shm_open(NAME, O_CREAT | O_RDWR, 0666);
4
5     /* configure the size of the shared memory segment
         */
6     ftruncate(shm, SHM_SIZE);
```

Dann wird in Listing 4.22 der Speicher angelegt und geöffnet. Danach wird dessen Größe gesetzt.

Listing 4.23: Beispiel Shared Memory: Einbinden

```
1     ptr = mmap(0, SHM_SIZE, PROT_READ | PROT_WRITE,
         MAP_SHARED, shm, 0);
2     if (ptr == MAP_FAILED) {
3         printf("Map␣failed\n");
4         exit(1);
5     }
```

In Listing 4.23 wird der Speicher in den Speicherbereich des Prozesses eingebunden. Sollte diese Aktion fehlschlagen wird das Programm mit einer Fehlermeldung beendet.

Listing 4.24: Beispiel Shared Memory: Schreiben/Lesen

```
1     if (argc == 1) {
2         printf("Memory:␣%s\n", (char*) ptr);
3     } else {
4         memset(ptr, 0, SHM_SIZE);
5         for (i = 0; i < argc; ++i) {
6             sprintf(ptr, "%s␣", argv[i]);
```

```
7               ptr += strlen(argv[i]) + 1;
8           }
9       }
10
11      munmap(ptr, SHM_SIZE);
12
13      return EXIT_SUCCESS;
```

In Listing 4.24 wird nun die Funktionalität des Programmes implementiert. Wird dem Programm kein Argument übergeben, so wird der Speicher ausgegeben. Werden dem Programm ein oder mehrere Argumente übergeben, werden diese in den Shared Memory geschrieben. Danach wird der Shared Memory aus dem Speicherbereich des Prozesses entfernt.

4.5.3. mmap

Mittels der "mmap" (4.5.3) Funktion kann ebenfalls ein Shared Memory angelegt werden. Dazu kann "mmap" (4.5.3) ein Dateideskriptor übergeben werden. So kann zum Beispiel eine Datei direkt in den Speicher "gemapped" werden. "mmap" (4.5.3) gibt also keinen Dateideskriptor zurück, sondern direkt die Adresse des Shared Memory.

4.5.3.1. Beispiel: Ausgeben der Quellcodedatei mit Hilfe von "mmap" (4.5.3)

In Listing B.10 ist ein Programm gelistet welches "mmap" (4.5.3) benutzt um den eigenen Quelltext auszugeben.

Dazu öffnet das Programm die Quellcodedatei wie in Listing 4.26 gezeigt mittels "open" (2.3) und bestimmt mittels "fstat" (**??**) die Größe der Datei.

Listing 4.25: Beispiel "mmap" (4.5.3): Öffnen der Datei

```
1   int          fd = open("./mmap_readfile.c", O_RDONLY
        );
2
3       ifpexit(fd < 0, "open");
4
5       status = fstat(fd, &s);
6       ifpexit(status < 0, "stat");
```

```
7     size = s.st_size;
```

Danach wird der Dateiinhalt des Dateideskriptors mittels “mmap” (4.5.3) in den Speicherbereich des Programmes “gemapped” und mittels “printf” ausgegeben. Im Anschluss wird der Shared Memory wieder mittels “munmap” aus dem Speicherbereich des Programmes entfernt.

Listing 4.26: Beispiel “mmap” (4.5.3): Ausgeben der Datei

```
1     mapped = mmap(0, size, PROT_READ, MAP_PRIVATE, fd,
          0);
2     ifpexit(mapped == MAP_FAILED, "mmap");
3
4     printf("%s", mapped);
5     munmap(mapped, size);
6     return 0;
```

4.5.3.2. Vorteile von “mmap” (4.5.3) gegenüber anderen Shared Memory-Schnittstellen

Da das Laden des Dateiinhalts in den Speicher bei “mmap” (4.5.3) vom Betriebssystemkern erledigt und nicht mehr Aufgabe des Programmierers ist, wird “mmap” (4.5.3) gerne verwendet. “mmap” (4.5.3) stellt eine starke Vereinfachung gegebenüber der Schnittstellen aus 4.5.1 und 4.5.2 dar. Es wird in vielen Unix-Standardwerkzeugen verwendet, wie zum Beispiel in “ls”, “mkdir” oder “touch” [2].

[2] Dies kann zum Beispiel mit “strace” nachvollzogen werden, indem dieser Befehl dem Kommandozeilenaufruf für das Programm vorangestellt wird

5. Sockets

Die letzte Form der Interprozesskommunikation, welche in dieser Arbeit behandelt werden soll, ist die Socket. Die Socket wird in dieser Arbeit in einem dedizierten Kapitel behandelt, da sie nicht die gleiche Art der Interprozesskommunikation abbildet wie die unter Kapitel 4 aufgeführt. Die Socket ermöglicht die Kommunikation zwischen Prozessen welche auf verschiedenen Hostsystemen (physischen Rechnern) existieren mittels (zum Beispiel) dem Transmission Control Protocol (TCP) oder dem User Datagram Protocol (UDP). In folgendem wird diese Kommunikationsart als Kommunikation über das Internet bezeichnet. Nichts desto trotz kann die Socket auch für die Interprozesskommunikation zwischen zwei Prozessen auf einem einzelnen Hostsystem benutzt werden.

Eine Socket stellt eine vom Betriebssystem zur Verfügung gestellte Ressource dar, welche (unter Unixoiden Betriebssystemen) wie eine normale Datei benutzt werden kann um Daten zu entfernten Prozessen zu senden. Unter Linux wird eine Socket als Dateideskriptor dargestellt. Bei der Kommunikation mit Sockets existiert immer ein "Server", welcher eine Kommunikationsschnittstelle mittels einer Socket anbietet, sowie einen oder mehrere "Clients" die diese Kommunikationsschnittstelle nutzen um Daten mit dem Server auszutauschen. Die "Clients" kennen sich untereinander nicht und treten in keinem Fall miteinander in Verbindung.

5.1. Syscalls zur Verwendung von Sockets

In Folgendem werden die Systemaufrufe beschrieben welche zur Verwendung von Sockets nötig sind.

5.1.1. socket

Mittels diesem Aufruf kann eine neue Socket erstellt werden.

Name	socket
Header	sys/types.h, sys/socket.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	Dateideskriptor der Socket
Rückgabe im Fehlerfall	-1
Errno	EACCES, EAFNOSUPPORT, EINVAL, EMFILE, ENOBUFS, ENOMEM, EPROTONOSUPPORT
Parameter	int domain, int type, int protocol

Tabelle 44.: Systemaufruf: socket

Die Parameter welche der Aufruf entgegennimmt werden in folgendem kurz beschrieben.

- "domain": Die Art der Kommunikation:

Tabelle 45.: Socket Kommunikationsarten

Name	Zweck
AF_UNIX, AF_LOCAL	Lokale Kommunikation
AF_INET	Kommunikation über **IP!**v4
AF_INET6	Kommunikation über **IP!**v6
AF_IPX	Novell-Protokoll
AF_NETLINK	Kernel user interface device
AF_X25	ITU-T X.25 & ISO-8208 Protokoll
AF_AX25	Amateurradio
AF_ATMPVC	Raw ATM PVC
AF_APPLETALK	AppleTalk
AF_PACKET	Paketinterface (Low-Level)
AF_ALG	Kernel Crypto API

- "type": Unterliegendes Protokoll:

Tabelle 46.: Socket: Unterliegendes Protokoll

Symbol	Bedeutung
STREAM	Zweiwege-Verbindungsbasierte Byte Streams (TCP)
DGRAM	Datagram-Kommunikation (UDP)
SEQPACKET	Sequenzierte, Verbindungsbasierte Zwei-Wege Kommunikation
RAW	Roher Netzwerkprotokollzugriff
RDM	Sichere Datagram Abstraktion welche keine Ordnung garaniert
PACKET	Veraltet

- "protocol": Spezifikation eines bestimmten Protokolls welches mit der Socket verwendet wird. Wird normalerweiße auf 0 initialisiert.

5.1.2. bind

Der Aufruf "socket" (5.1.1) kann verwendet werden um eine Socket anzulegen welche dann mit diesem Systemaufruf an eine Adresse gebunden werden kann. "bind" (5.1.2) bindet eine Adresse mit einer Socket, was traditionell als "einen Namen einer Socket zuweißen" genannt wird.

Name	bind
Header	sys/types.h, sys/socket.h
Rückgabetyp	int
Rückgabe im Erfolgsfall	0
Rückgabe im Fehlerfall	-1
Errno	EACCES, EADDRINUSE, EBADF, EINVAL, ENOTSOCK, EADDRNOTAVAIL, EFAULT, ELOOP, ENAMETOOLONG, ENOENT, ENOMEM, ENOTDIR, EROFS
Parameter	int sockfd, const struct sockaddr*, socklen_t

Tabelle 47.: Systemaufruf: bind

5.1.3. recv

Mittels dieses Aufrufes können Daten aus der Socket in einen internen Puffer gelesen werden um sie von dort aus weiter zu verarbeiten. Es ist dabei nicht von Belang ob die Socket eine TCP-Verbindung oder eine UDP-Verbindung abbildet.

Name	recv
Header	sys/types.h, sys/socket.h
Rückgabetyp	ssize_t
Rückgabe im Erfolgsfall	Anzahl gelesener Bytes
Rückgabe im Fehlerfall	-1
Errno	EAGAIN, EWOULDBLOCK, EBADF, ECONNREFUSED, EFAULT, EINTR, EINVAL, ENOMEM, ENOTCONN, ENOTSOCK
Parameter	int sockfd, void* buf, size_t len, int flags

Tabelle 48.: Systemaufruf: recv

Der Aufruf "recv" (5.1.3) existiert zudem in zwei weiteren Varianten, welche die gleiche Funktionalität anbieten, sich jedoch in ihrem Verhalten vom "recv" (5.1.3)

Aufruf unterscheiden.

Alle Varianten des Aufrufes blockieren bis eine Nachricht vorhanden ist, vorausgesetzt die Socket wurde als Nicht-Blockierende Socket angelegt.

Sollte der Puffer, in welchen die Nachricht geschrieben werden nicht ausreichen um die gesamte Nachricht aufzunehmen, so wird, je nach Art der Socket, der Rest der Nachricht eventuell verworfen.

5.1.3.1. recvfrom

Name	recvfrom
Header	sys/types.h, sys/socket.h
Rückgabetyp	ssize_t
Rückgabe im Erfolgsfall	Anzahl gelesener Bytes
Rückgabe im Fehlerfall	-1
Errno	EAGAIN, EWOULDBLOCK, EBADF, ECONNREFUSED, EFAULT, EINTR, EINVAL, ENOMEM, ENOTCONN, ENOTSOCK
Parameter	int sockfd, void* buf, size_t len, int flags, struct sockaddr* src, socklen_t* addrlen

Tabelle 49.: Systemaufruf: recvfrom

Mittels dieses Aufrufes kann beim Empfangen einer Nachricht zudem erkannt werden, wer die Nachricht versendet hat. Die sogenannte "Source-Adress" wird von diesem Aufruf an den Aufrufenden zurückgegeben.

5.1.3.2. recvmsg

Name	recvmsg
Header	sys/types.h, sys/socket.h
Rückgabetyp	ssize_t
Rückgabe im Erfolgsfall	Anzahl gelesener Bytes
Rückgabe im Fehlerfall	-1
Errno	EAGAIN, EWOULDBLOCK, EBADF, ECONNREFUSED, EFAULT, EINTR, EINVAL, ENOMEM, ENOTCONN, ENOTSOCK
Parameter	int sockfd, struct msghdr* msg, int flags

Tabelle 50.: Systemaufruf: recvmsg

5.1.3.3. Flags für "recv" (5.1.3)

Jedem der Varianten des "recv" (5.1.3)-Aufrufes können Flags übergeben werden.

Tabelle 51.: "recv" (5.1.3)-Flags

Symbol	Beschreibung
MSG_CMSG_CLOEXEC	"Close-on-exec" Flag, wie "O_CLOEXEC"
MSG_DONTWAIT	Führt die Operation Nicht-Blockierend aus
MSG_ERRQUEUE	Empfange von der Socket aufgenommene Errors
MSG_OOB	Empfange "out-of-band" Daten
MSG_PEEK	Lese Daten ohne sie aus der Queue zu entfernen
MSG_TRUNC	Gebe Gesamtlänge der Nachricht zurück, auch wenn der Puffer zum Einlesen kürzer ist
MSG_WAITALL	Blockiere bis der Aufruf den kompletten Request zurückliefern kann

5.2. Beispiel: Uppercase-Server

In folgendem Beispiel wird ein einfacher Server beschrieben, welcher Text empfängt, diesen in Großbuchstaben umwandelt und zurück zum Client sendet. (entnommen aus [Ker10, S. 1171 ff]).

Die Programme verwenden UDP.

5.2.1. Gemeinsamer Header

Listing 5.1: Beispiel Uppercase-Server: Gemeinsamer Header

```
1 #include <sys/un.h>
2 #include <sys/socket.h>
3 #include <ctype.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <errno.h>
7
8 #define BUF_SIZE        (10)
9 #define SV_SOCK_PATH    ("/tmp/socket_udp")
10
```

```
11 #define err_exit(x) do { fprintf(stderr, (x)); exit(1);
       } while (0)
```

In Listing 5.1 wurde ein Headerfile definiert, welches "include"-Statements, eine Definition der Größe des Puffers welche beide Programme benutzen sowie den Pfad zur verwendeten UDP-Socket enthält.

Ausserdem ist ein Helfer-Makro dort definiert.

5.2.2. Server

Der Quellcode des Servers (vollständig unter Listing B.3) soll in Folgendem erklärt werden.

Listing 5.2: Beispiel Uppercase-Server: Server: Part 1

```
1     sfd = socket(AF_UNIX, SOCK_DGRAM, 0);
2     if (sfd == -1) {
3         err_exit("socket");
4     }
```

In Listing 5.2 wird eine Socket des Typs "UNIX" geöffnet. Die Socket wird als Datagram-Socket angelegt, falls dies fehl schlägt, wird das Programm beendet.

Listing 5.3: Beispiel Uppercase-Server: Server: Part 2

```
1     if (remove(SV_SOCK_PATH) == -1 && errno != ENOENT)
          {
2         err_exit("remove");
3     }
4
5     memset(&svaddr, 0, sizeof(struct sockaddr_un));
6     svaddr.sun_family = AF_UNIX;
7     strncpy(svaddr.sun_path, SV_SOCK_PATH, sizeof(
          svaddr.sun_path) - 1);
```

Dann wird in Listing 5.3 zuerst der Socket-Pfad, welcher eventuell noch von vorherigen Aufrufen des Programms existiert, entfernt. Sollte dies nicht funktionieren wird das Programm beendet.

Dann wird die Datenstruktur welche die Adressspezifikation enthält mit Nullen überschrieben und danach die Membervariablen auf die entsprechenden Werte zur Verwendung einer Unix-Socket gesetzt und der Pfad der Socket eingetragen.

Listing 5.4: Beispiel Uppercase-Server: Server: Part 3

```
1     if (bind(sfd, (struct sockaddr *) &svaddr, sizeof(
          struct sockaddr_un)) == -1) {
2         err_exit("bind");
3     }
```

In Listing 5.4 wird nun die Socket (mittels des Filedeskriptors welcher beim Aufruf von "socket" zurückgegeben wurde, Listing 5.2) an die Adresse (Listing 5.3) gebunden. Sollte dieser Aufruf nicht erfolgreich sein wird das Programm mit einer Fehlermeldung beendet.

Listing 5.5: Beispiel Uppercase-Server: Server: Part 4

```
1     for (;;) {
2         len = sizeof(struct sockaddr_un);
3         nbytes = recvfrom(sfd, buf, BUF_SIZE, 0,
4                 (struct sockaddr *) &claddr, &len);
5         if (nbytes == -1) {
6             err_exit("recvfrom");
7         }
8
9         printf("Server␣received␣%ld␣bytes␣from␣%s\n", (
              long) nbytes, claddr.sun_path);
10
11        for (j = 0; j < nbytes; j++) {
12            buf[j] = toupper((unsigned char) buf[j]);
13        }
14
15        if (sendto(sfd, buf, nbytes, 0, (struct
              sockaddr *) &claddr, len) !=
16                 nbytes) {
17            err_exit("sendto");
18        }
19    }
```

Im letzten Teil des Programmes (Listing 5.5) läuft der sogenannte "Main-Loop", also der Teil des Programmes, welcher die Kommunikation abwickelt. Es handelt sich hierbei um eine Dauerschleife welche den "recvfrom"-Aufruf benutzt um die Dateien empfängt und in einen Buffer schreibt, dessen Größe in Listing 5.1 definiert wurde. Der Server schreibt dann auf seine Konsole dass er Daten empfangen hat, wandelt diese

dann in Großbuchstaben um und sendet sie zurück zum Client. Sollte das Emfpangen oder Senden fehl schlagen, beendet sich der Server mit einer Fehlernachricht.

5.2.3. Client

In folgendem soll der Client beschrieben werden, welcher mit dem in Unterabschnitt 5.2.2 definierten Server kommuniziert. Der Quellcode des Clients ist vollständig unter Listing B.4.

Der Quellcode des Clients (vollständig unter Listing B.4) soll in Folgendem erklärt werden.

Da dass Client-Programm ein Kommandozeilenprogramm sein soll, wird in **??** als erstes überprüft, ob das Programm mit einem Parameter aufgerufen wurde. Sollte dieser Parameter nicht vorhanden, oder aber auf "–help" gesetzt sein, wird ein kurzer Hilfetext ausgegeben und das Programm dann beendet. Sollte der Parameter (welcher der zu sendende Text sein soll) vorhanden sein, wird das Programm weiter ausgeführt.

Listing 5.6: Beispiel Uppercase-Server: Client: Part 1

```
1     if (argc < 2 || strcmp(argv[1], "--help") == 0) {
2         fprintf(stderr, "%s msg...\n", argv[0]);
3         exit(1);
4     }
```

Danach wird (in Listing 5.7) eine Unix-Socket erstellt, welche über UDP kommuniziert. Sollte das Erstellen dieser Socket fehlschlagen, wird das Programm mit einem Fehler beendet. Im Erfolgssfall wird der Speicher der Adress-Struktur erst mit 0 initialisiert und dann mit einem Pfad zu einer Datei welche dem aktuellen Prozess gehört, beschrieben. Diese wird später benutzt um die Antwort des Servers zu erhalten.

Listing 5.7: Beispiel Uppercase-Server: Client: Part 2

```
1     sfd = socket(AF_UNIX, SOCK_DGRAM, 0);
2     if (sfd == -1) {
3         err_exit("socket");
4     }
5
6     memset(&claddr, 0, sizeof(struct sockaddr_un));
7     claddr.sun_family = AF_UNIX;
8     snprintf(claddr.sun_path, sizeof(claddr.sun_path),
          "/tmp/ud_ucase_cl.%ld", (long) getpid());
```

Diese Adresse wird in Listing 5.8 an die Socket gebunden. Im Anschluss wird die Socket vorbereitet welche dazu benutzt wird die Daten zum Server zu senden.

Listing 5.8: Beispiel Uppercase-Server: Client: Part 3

```
1     memset(&claddr, 0, sizeof(struct sockaddr_un));
2     claddr.sun_family = AF_UNIX;
3     snprintf(claddr.sun_path, sizeof(claddr.sun_path),
         "/tmp/ud_ucase_cl.%ld", (long) getpid());
4
5     if (bind(sfd, (struct sockaddr *) &claddr, sizeof(
         struct sockaddr_un)) == -1) {
6         err_exit("bind");
7     }
```

Nun kann mit dem Senden der Daten begonnen werden. Die Daten sind als Parameter an das Programm übergeben worden, weswegen durch die Liste aller Parameter iteriert wird um jeden einzelnen Parameter an den Server zu senden. Sollte das Senden eines Parameters fehlschlagen, wird das Programm mit einem Fehler beendet.

Listing 5.9: Beispiel Uppercase-Server: Client: Part 4

```
1     for (j = 1; j < argc; j++) {
2         msg_len = strlen(argv[j]);        /* May be
             longer than BUF_SIZE */
3         if (sendto(sfd, argv[j], msg_len, 0, (struct
             sockaddr *) &svaddr, sizeof(struct
             sockaddr_un)) != msg_len) {
4             err_exit("sendto");
5         }
6
7         nbytes = recvfrom(sfd, resp, BUF_SIZE, 0, NULL,
              NULL);
8         if (nbytes == -1) {
9             err_exit("recvfrom");
10        }
11        printf("Response␣%d:␣%.*s\n", j, (int) nbytes,
             resp);
12    }
13
14    remove(claddr.sun_path);
15    exit(EXIT_SUCCESS);
```

```
16 }
```

Nachdem das Senden erfolgreich abgeschlossen wurde, werden nun Daten vom Server empfangen und auf der Kommandozeile ausgegeben.

Sobald alle Parameter gesendet und in Großbuchstaben empfangen sowie ausgegeben wurden, wird die Socket welche zum Empfangen genutzt wurde, entfernt und das Programm beendet.

5.3. Beispiel: Dateideskriptoren senden

Listing B.5 zeigt ein Beispielprogram mittels welchem Ein Dateideskriptor über eine Socket versendet wird. Das Programm öffnet zunächst einen Dateideskriptor mittels welchem die Quellcodedatei des Programms eingelesen werden kann. Danach versendet es diesen Dateideskriptor an einen Fork von sich selbst, welcher dann diesen Deskriptor ausließt und auf der Standardausgabe ausgibt.

In dem Programm sind zwei Helferfunktionen definiert, welche das Senden und Empfangen des Deskriptors vereinfachen.

5.3.1. Senden des Dateideskriptors

Listing 5.10: Beispiel Dateideskriptor senden: Senden

```
1  static void send_fd(int socket, int fd) { // send fd by socket
2      char             buf[CMSG_SPACE(sizeof(fd))];
3      struct msghdr    msg  = { 0 };
4      struct iovec     io   = { .iov_base = "ABC", .iov_len = 3 };
5      struct cmsghdr* cmsg = CMSG_FIRSTHDR(&msg);
6
7      memset(buf, '\0', sizeof(buf));
8
9      msg.msg_iov                = &io;
10     msg.msg_iovlen             = 1;
11     msg.msg_control            = buf;
12     msg.msg_controllen         = sizeof(buf);
13
```

```
14      cmsg->cmsg_level            = SOL_SOCKET;
15      cmsg->cmsg_type             = SCM_RIGHTS;
16      cmsg->cmsg_len              = CMSG_LEN(sizeof(fd));
17
18      *((int *) CMSG_DATA(cmsg)) = fd;
19
20      msg.msg_controllen          = cmsg->cmsg_len;
21
22      if (sendmsg(socket, &msg, 0) < 0) {
23          printf("%s", "Failed to send message\n");
24      }
25  }
```

Listing 5.10 zeigt die Funktion welche zum Senden des Dateideskriptors verwendet wird. Diese legt bei ihrem Aufruf eine Kontrollnachricht an. Diese Nachricht wird initialisiert, mit dem Deskriptor befüllt und dann über die übergebene Socket versendet.

5.3.2. Emfpangen des Dateideskriptors

Listing 5.11: Beispiel Dateideskriptor senden: Empfangen

```
1  static int extract_fd(int socket) { // receive fd from
       socket
2      char m_buffer[256];
3      char c_buffer[256];
4      struct cmsghdr * cmsg = CMSG_FIRSTHDR(&msg);
5      unsigned char * data  = CMSG_DATA(cmsg);
6      int fd                = *((int*) data);
7      struct msghdr msg     = {0};
8      struct iovec io       = { .iov_base = m_buffer, .
         iov_len = sizeof(m_buffer) };
9      msg.msg_iov           = &io;
10     msg.msg_iovlen        = 1;
11     msg.msg_control       = c_buffer;
12     msg.msg_controllen    = sizeof(c_buffer);
13
14     if (recvmsg(socket, &msg, 0) < 0) {
15         printf("%s", "Failed to receive message\n");
16     }
17
```

```
18      printf("%s", "About to extract fd\n");
19      printf("%s", "Extracted fd %d\n", fd);
20
21      return fd;
22  }
```

Listing 5.11 zeigt die Funktion welche zum Emfangen des Dateideskriptors verwendet wird. Die Funktion empfängt die Nachricht über die übergebene Socket und extrahiert den Dateideskriptor aus dieser, welcher dann zum Aufrufer zurückgegeben wird.

5.3.3. Main-Funktion

Die Mainfunktion des Programms aus Listing B.5 erstellt zunächst eine Socket mit zwei Dateideskriptoren, einer zum Lesen und einer zum Schreiben in die Socket. Danach teilt sich das Programm mittels "fork" (3.1). Ein Teil (der Elternteil in diesem Beispiel) öffnet einen weiteren Dateideskriptor welcher die Quellcodedatei des Programms beinhaltet. Dieser wird dann an den Kindteil des Programms versendet. Der Kindteil des Programms versucht den Deskriptor zu empfangen und ließt aus diesem dann den Quelltext des Programmes aus. Dieser wird nun auf der Kommandozeile ausgegeben.

Glossary

Ellipse

Eine Ellipse beschreibt eine unbestimmte Anzahl von Argumenten für eine Funktion oder wird verwendet um einen Bereich anzugeben (Zum Beispiel $1..100$) [?].. 12, 71

System V

"System V" ist der Name eines Unix-Betriebssystems von AT&T, welches im Jahr 1983 veröffentlicht wurde. Mit "System V" werden verschiedene Betriebssysteme referenziert, da aus "System V" einige Derivate hervorgingen. Die Konzepte, welche das Betriebssystem umsetze, sind heute noch in vielen Unix-artigen Betriebssystemen und auch in "BSD"-Derivaten zu finden ([?]).. 29, 31, 36, 38, 47, 48, 50, 51, 71, 88

Literaturverzeichnis

[BC05] Bovet, Daniel P. ; Cesati, Marco: *Understanding the Linux kernel*. Ö'Reilly Media, Inc."̈, 2005

[GMBW95] Goldt, Sven ; Meer, Sven van d. ; Burkett, Scott ; Welsh, Matt: The linux programmer's guide. In: *Linux Documentation Project* (1995)

[Ker10] Kerrisk, Michael: *The Linux programming interface*. No Starch Press, 2010. – ISBN 978–1–59327–220–3

[Lin12] Linux: *VFORK(2) Linux Programmer's Manual*. Manpage, 08 2012

[Lin14] Linux: *FLOCK(2) Linux Programmer's Manual*. Manpage, 09 2014

[Lin15a] Linux: *DUP(2) Linux Programmer's Manual*. Manpage, 08 2015

[Lin15b] Linux: *LOCKF(2) Linux Programmer's Manual*. Manpage, 08 2015

[Ope15] Opengroup: *pthread.h - threads*. `http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html#tag_13_35`. Version: 11 2015

[Wik15a] Wikipedia: *Entry Point*. `https://en.wikipedia.org/wiki/Entry_point`. Version: 10 2015

[Wik15b] Wikipedia: *POSIX*. `https://de.wikipedia.org/wiki/Portable_Operating_System_Interface`. Version: 10 2015

[Zel08] Zelenski, Julie: *Thread and Semaphore Examples*. May 2008

Eidesstattliche Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbständig verfasst und hierzu keine anderen als die angegebenen Hilfsmittel verwendet habe. Alle Stellen der Arbeit die wörtlich oder sinngemäß aus fremden Quellen entnommen wurden, sind als solche kenntlich gemacht.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keinem anderen Studiengang als Prüfungsleistung vorgelegt oder an anderer Stelle veröffentlicht.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

Furtwangen, 10.01.2016

A. Abbildungen

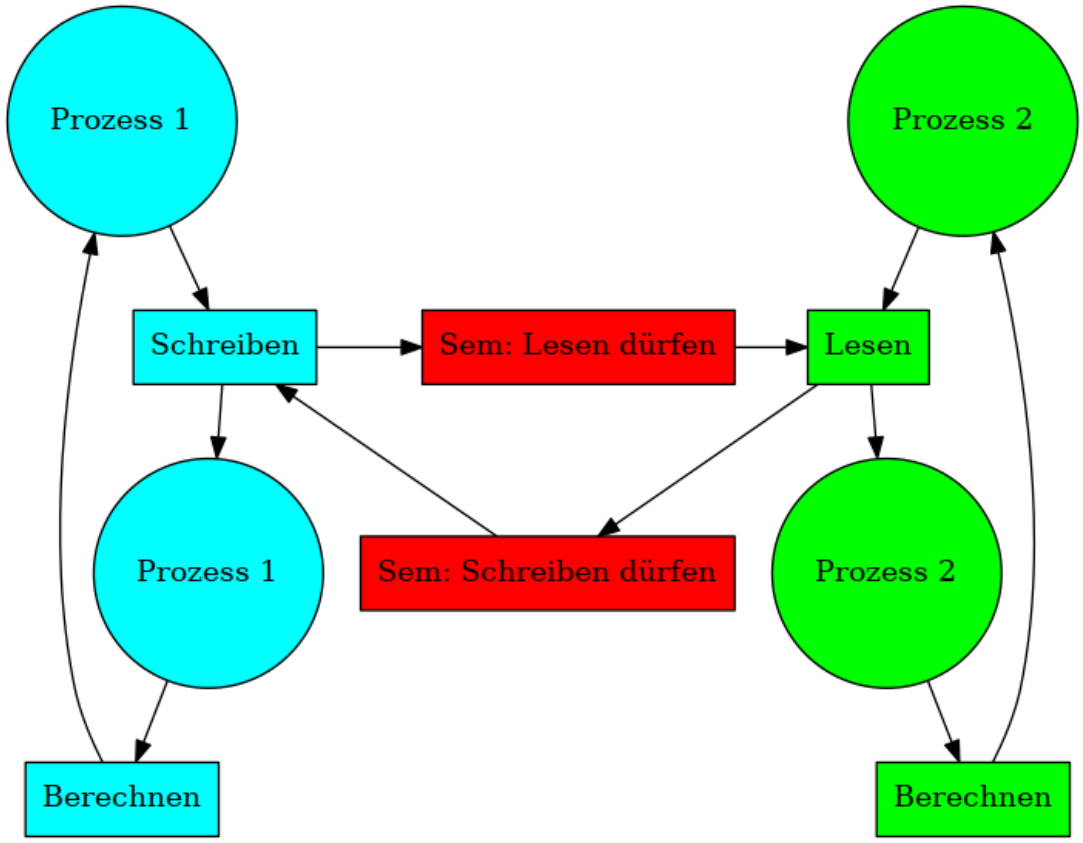

Abbildung 4.: Szenario: Zwei Prozesse

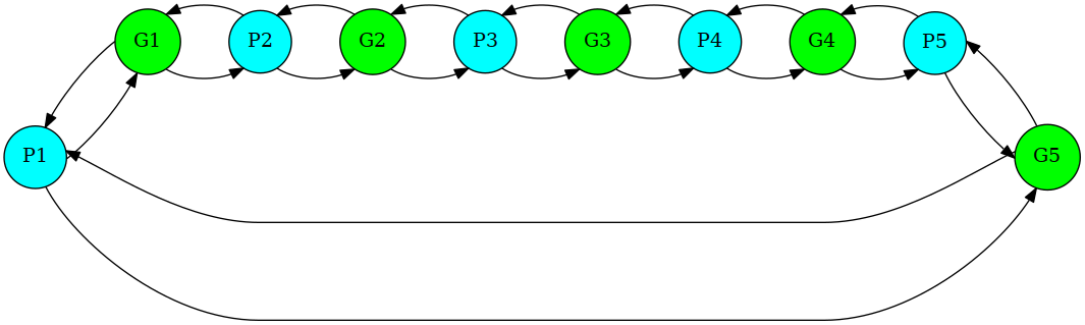

Abbildung 5.: Szenario: Philosophen

B. Listings

Listing B.1: Inhalt einer Datei nach unsynchronisiertem schreiben

```
1  Eltern-Prozess
2  Eltern-Prozess
3  Eltern-Prozess
4  Eltern-Prozess
5  Eltern-Prozess
6  Eltern-Prozess
7  Eltern-Prozess
8  Eltern-Prozess
9  Eltern-Prozess
10 Eltern-Prozess
11 Eltern-Prozess
12 Eltern-Prozess
13 Eltern-Prozess
14 Eltern-Prozess
15 Eltern-Prozess
16 Eltern-Prozess
17 Eltern-Prozess
18 Eltern-Prozess
19 Eltern-Prozess
20 Eltern-Prozess
21 Eltern-Prozess
22 Eltern-Prozess
23 Eltern-Prozess
24 Eltern-Prozess
25 Eltern-Prozess
26 Eltern-Prozess
27 Eltern-Prozess
28 Eltern-Prozess
29 Eltern-Prozess
30 Eltern-Prozess
31 Eltern-Prozess
32 Eltern-Prozess
33 Eltern-Prozess
34 Eltern-Prozess
35 Eltern-Prozess
36 Eltern-Prozess
37 Eltern-Prozess
38 Eltern-Prozess
39 Eltern-Prozess
40 Eltern-Prozess
```

```
41 Eltern-Prozess
42 Eltern-Prozess
43 Eltern-Prozess
44 Eltern-Prozess
45 Eltern-Prozess
46 Eltern-Prozess
47 Eltern-Prozess
48 Eltern-Prozess
49 Eltern-Prozess
50 Eltern-Prozess
51 Eltern-Prozess
52 Eltern-Prozess
53 Eltern-Prozess
54 Eltern-Prozess
55 Eltern-Prozess
56 Eltern-Prozess
57 Eltern-Prozess
58 Eltern-Prozess
59 Eltern-Prozess
60 Eltern-Prozess
61 Eltern-Prozess
62 Eltern-Prozess
63 Eltern-Prozess
64 Eltern-Prozess
65 Eltern-Prozess
66 Eltern-Prozess
67 Kind-Prozess
68 Eltern-Prozess
69 Kind-Prozess
70 Kind-Prozess
71 Eltern-Prozess
72 Kind-Prozess
73 Eltern-Prozess
74 Kind-Prozess
75 Eltern-Prozess
76 Kind-Prozess
77 Eltern-Prozess
78 Kind-Prozess
79 Eltern-Prozess
80 Kind-Prozess
81 Eltern-Prozess
82 Kind-Prozess
83 Eltern-Prozess
84 Kind-Prozess
85 Eltern-Prozess
86 Kind-Prozess
87 Eltern-Prozess
88 Kind-Prozess
```

```
89 Eltern-Prozess
90 Kind-Prozess
91 Eltern-Prozess
92 Kind-Prozess
93 Eltern-Prozess
94 Kind-Prozess
95 Eltern-Prozess
96 Kind-Prozess
97 Eltern-Prozess
98 Kind-Prozess
99 Eltern-Prozess
100 Kind-Prozess
101 Eltern-Prozess
102 Kind-Prozess
103 Eltern-Prozess
104 Kind-Prozess
105 Eltern-Prozess
106 Kind-Prozess
107 Eltern-Prozess
108 Kind-Prozess
109 Eltern-Prozess
110 Kind-Prozess
111 Eltern-Prozess
112 Kind-Prozess
113 Eltern-Prozess
114 Kind-Prozess
115 Eltern-Prozess
116 Kind-Prozess
117 Eltern-Prozess
118 Kind-Prozess
119 Eltern-Prozess
120 Kind-Prozess
121 Eltern-Prozess
122 Kind-Prozess
123 Eltern-Prozess
124 Kind-Prozess
125 Eltern-Prozess
126 Kind-Prozess
127 Eltern-Prozess
128 Kind-Prozess
129 Eltern-Prozess
130 Kind-Prozess
131 Eltern-Prozess
132 Kind-Prozess
133 Eltern-Prozess
134 Kind-Prozess
135 Eltern-Prozess
136 Kind-Prozess
```

```
137 Kind-Prozess
138 Kind-Prozess
139 Kind-Prozess
140 Kind-Prozess
141 Kind-Prozess
142 Kind-Prozess
143 Kind-Prozess
144 Kind-Prozess
145 Kind-Prozess
146 Kind-Prozess
147 Kind-Prozess
148 Kind-Prozess
149 Kind-Prozess
150 Kind-Prozess
151 Kind-Prozess
152 Kind-Prozess
153 Kind-Prozess
154 Kind-Prozess
155 Kind-Prozess
156 Kind-Prozess
157 Kind-Prozess
158 Kind-Prozess
159 Kind-Prozess
160 Kind-Prozess
161 Kind-Prozess
162 Kind-Prozess
163 Kind-Prozess
164 Kind-Prozess
165 Kind-Prozess
166 Kind-Prozess
167 Kind-Prozess
168 Kind-Prozess
169 Kind-Prozess
170 Kind-Prozess
171 Kind-Prozess
172 Kind-Prozess
173 Kind-Prozess
174 Kind-Prozess
175 Kind-Prozess
176 Kind-Prozess
177 Kind-Prozess
178 Kind-Prozess
179 Kind-Prozess
180 Kind-Prozess
181 Kind-Prozess
182 Kind-Prozess
183 Kind-Prozess
184 Kind-Prozess
```

```
Kind-Prozess
Kind-Prozess
Kind-Prozess
Kind-Prozess
Kind-Prozess
Kind-Prozess
Kind-Prozess
Kind-Prozess
Kind-Prozess
Kind-Prozess
Kind-Prozess
Kind-Prozess
Kind-Prozess
Kind-Prozess
Kind-Prozess
```

Listing B.2: Beispiel: Einfache Threadsynchronisation

```
#include<stdio.h>
#include<string.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>

pthread_t       tid[2];
int             counter;
pthread_mutex_t lock;

void* doSomeThing(void *arg) {
    pthread_mutex_lock(&lock);

    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d started\n", counter);

    for(i = 0; i < (0xFFFFFFF); i++); // do nothing

    printf("\n Job %d finished\n", counter);
    pthread_mutex_unlock(&lock);

    return NULL;
}

int main(void)
{
    int i;
    int err;

```

```
31      if (pthread_mutex_init(&lock, NULL) != 0) {
32          printf("\n mutex init failed\n");
33          return EXIT_FAILURE;
34      }
35
36      for (i = 0; i < 2; ++i) {
37          err = pthread_create(&(tid[i]), NULL, &
                doSomeThing, NULL);
38          if (err != 0) {
39              printf("\ncan't create thread :[%s]", 
                    strerror(err));
40          }
41      }
42
43      pthread_join(tid[0], NULL);
44      pthread_join(tid[1], NULL);
45      pthread_mutex_destroy(&lock);
46
47      return EXIT_SUCCESS;
48  }
```

Listing B.3: Beispiel Uppercase-Server: Server

```
1  #include "simple_sockets.h"
2
3  int
4  main(int argc, char *argv[])
5  {
6      struct sockaddr_un svaddr;
7      struct sockaddr_un claddr;
8      int sfd;
9      int j;
10     ssize_t nbytes;
11     socklen_t len;
12     char buf[BUF_SIZE];
13
14     sfd = socket(AF_UNIX, SOCK_DGRAM, 0);
15     if (sfd == -1) {
16         err_exit("socket");
17     }
18
19     if (remove(SV_SOCK_PATH) == -1 && errno != ENOENT) 
           {
20         err_exit("remove");
21     }
22
23     memset(&svaddr, 0, sizeof(struct sockaddr_un));
24     svaddr.sun_family = AF_UNIX;
25     strncpy(svaddr.sun_path, SV_SOCK_PATH, sizeof(
```

```
        svaddr.sun_path) - 1);
26
27      if (bind(sfd, (struct sockaddr *) &svaddr, sizeof(
            struct sockaddr_un)) == -1) {
28          err_exit("bind");
29      }
30
31      for (;;) {
32          len = sizeof(struct sockaddr_un);
33          nbytes = recvfrom(sfd, buf, BUF_SIZE, 0,
34                  (struct sockaddr *) &claddr, &len);
35          if (nbytes == -1) {
36              err_exit("recvfrom");
37          }
38
39          printf("Server␣received␣%ld␣bytes␣from␣%s\n", (
                long) nbytes, claddr.sun_path);
40
41          for (j = 0; j < nbytes; j++) {
42              buf[j] = toupper((unsigned char) buf[j]);
43          }
44
45          if (sendto(sfd, buf, nbytes, 0, (struct
                sockaddr *) &claddr, len) !=
46                  nbytes) {
47              err_exit("sendto");
48          }
49      }
50  }
```

Listing B.4: Beispiel Uppercase-Server: Client

```
1  #include "simple_sockets.h"
2
3  #include <sys/types.h>
4  #include <unistd.h>
5
6  int
7  main(int argc, char *argv[])
8  {
9      struct sockaddr_un svaddr;
10     struct sockaddr_un claddr;
11     int sfd;
12     int j;
13     size_t msg_len;
14     ssize_t nbytes;
15     char resp[BUF_SIZE];
16
17     if (argc < 2 || strcmp(argv[1], "--help") == 0) {
```

```
18          fprintf(stderr, "%s msg...\n", argv[0]);
19          exit(1);
20      }
21
22      sfd = socket(AF_UNIX, SOCK_DGRAM, 0);
23      if (sfd == -1) {
24          err_exit("socket");
25      }
26
27      memset(&claddr, 0, sizeof(struct sockaddr_un));
28      claddr.sun_family = AF_UNIX;
29      snprintf(claddr.sun_path, sizeof(claddr.sun_path),
            "/tmp/ud_ucase_cl.%ld", (long) getpid());
30
31      if (bind(sfd, (struct sockaddr *) &claddr, sizeof(
            struct sockaddr_un)) == -1) {
32          err_exit("bind");
33      }
34
35      memset(&svaddr, 0, sizeof(struct sockaddr_un));
36      svaddr.sun_family = AF_UNIX;
37      strncpy(svaddr.sun_path, SV_SOCK_PATH, sizeof(
            svaddr.sun_path) - 1);
38
39      for (j = 1; j < argc; j++) {
40          msg_len = strlen(argv[j]);          /* May be
                longer than BUF_SIZE */
41          if (sendto(sfd, argv[j], msg_len, 0, (struct
                sockaddr *) &svaddr, sizeof(struct
                sockaddr_un)) != msg_len) {
42              err_exit("sendto");
43          }
44
45          nbytes = recvfrom(sfd, resp, BUF_SIZE, 0, NULL,
                NULL);
46          if (nbytes == -1) {
47              err_exit("recvfrom");
48          }
49          printf("Response %d: %.*s\n", j, (int) nbytes,
                resp);
50      }
51
52      remove(claddr.sun_path);
53      exit(EXIT_SUCCESS);
54  }
```

Listing B.5: Beispiel Filedeskriptor senden

```
1 #include <fcntl.h>
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <time.h>
#include <unistd.h>

static void send_fd(int socket, int fd) { // send fd by socket
    char             buf[CMSG_SPACE(sizeof(fd))];
    struct msghdr    msg  = { 0 };
    struct iovec     io   = { .iov_base = "ABC", .iov_len = 3 };
    struct cmsghdr* cmsg = CMSG_FIRSTHDR(&msg);

    memset(buf, '\0', sizeof(buf));

    msg.msg_iov                 = &io;
    msg.msg_iovlen              = 1;
    msg.msg_control             = buf;
    msg.msg_controllen          = sizeof(buf);

    cmsg->cmsg_level            = SOL_SOCKET;
    cmsg->cmsg_type             = SCM_RIGHTS;
    cmsg->cmsg_len              = CMSG_LEN(sizeof(fd));

    *((int *) CMSG_DATA(cmsg)) = fd;

    msg.msg_controllen          = cmsg->cmsg_len;

    if (sendmsg(socket, &msg, 0) < 0) {
        printf("%s", "Failed to send message\n");
    }
}

static int extract_fd(int socket) { // receive fd from socket
    char m_buffer[256];
    char c_buffer[256];
    struct cmsghdr * cmsg = CMSG_FIRSTHDR(&msg);
    unsigned char * data  = CMSG_DATA(cmsg);
    int fd                = *((int*) data);
    struct msghdr msg     = {0};
    struct iovec io       = { .iov_base = m_buffer, .iov_len = sizeof(m_buffer) };
    msg.msg_iov           = &io;
    msg.msg_iovlen        = 1;
```

```
46      msg.msg_control    = c_buffer;
47      msg.msg_controllen = sizeof(c_buffer);
48
49      if (recvmsg(socket, &msg, 0) < 0) {
50          printf("%s", "Failed to receive message\n");
51      }
52
53      printf("%s", "About to extract fd\n");
54      printf("%s", "Extracted fd %d\n", fd);
55
56      return fd;
57  }
58
59  int main(int argc, char **argv) {
60      const char *filename = "./send_fd_sock.c";
61
62      printf("%s", argv[0]);
63      if (argc > 1) {
64          filename = argv[1];
65      }
66
67      int sv[2];
68      if (socketpair(AF_UNIX, SOCK_DGRAM, 0, sv) != 0) {
69          printf("%s", "Failed to create Unix-domain ↩
               socket pair\n");
70      }
71
72      int pid = fork();
73      if (pid > 0) { // in parent
74          int sock = sv[0];
75          int fd;
76
77          printf("%s", "Parent at work\n");
78          close(sv[1]);
79
80          fd = open(filename, O_RDONLY);
81          if (fd < 0) {
82              printf("Failed to open file %s for reading\ ↩
                   n", filename);
83          }
84
85          send_fd(sock, fd);
86
87          close(fd);
88          nanosleep(&(struct timespec){ .tv_sec = 1, . ↩
               tv_nsec = 500000000}, 0);
89          printf("%s", "Parent exits\n");
90      } else { // in child
```

```
91          char    buffer[256];
92          ssize_t nbytes;
93          int     sock = sv[1];
94          int     fd;
95
96          printf("%s", "Child at play\n");
97          close(sv[0]);
98
99          nanosleep(&(struct timespec){ .tv_sec = 0, .
               tv_nsec = 500000000}, 0);
100
101         fd = extract_fd(sock);
102
103         printf("Read %d!\n", fd);
104
105         while ((nbytes = read(fd, buffer, sizeof(buffer
               ))) > 0) {
106             write(1, buffer, nbytes);
107         }
108
109         printf("Done!\n");
110         close(fd);
111     }
112
113     return 0;
114 }
```

Listing B.6: Beispiel System V Semaphore

```
1  #include <sys/types.h>
2  #include <sys/ipc.h>
3  #include <sys/sem.h>
4  #include <stdlib.h>
5  #include <stdio.h>
6  #include <unistd.h>
7
8  static struct sembuf sop_lock[2] = {
9      { 0 /* Nr */, 0 /* Op (wait == 0) */, 0  /* Flags (
          none) */ },
10     { 0 /* Nr */, 1 /* Op (+1) */, SEM_UNDO /* Flags (
          undo changes on crash) */ },
11 };
12
13 static struct sembuf sop_unlock[2] = {
14     { 0 /* Number */, -1 /* Op (sub one) */, IPC_NOWAIT
           /* Flags: Do not block */ },
15 };
16
17 int semid = -1; // id
```

```
18
19 void my_lock(void) {
20     if (semid < 0 && ((semid = semget(123456L, 1, ↩
       IPC_CREAT | 0666)) < 0)) {
21         perror("Semget error");
22     }
23
24     if (semop(semid, sop_lock, 2) < 0) {
25         perror("semop lock error");
26     }
27 }
28
29 void my_unlock(void) {
30     if (semop(semid, sop_unlock, 1) < 0) {
31         perror("semop unlock error");
32     }
33 }
34
35 int main(void) {
36     pid_t i = getpid();
37     int j;
38     my_lock();
39     {
40         sleep(1);
41         printf("Thread running: %i\n", i);
42     }
43     my_unlock();
44
45     sleep(1);
46
47     my_lock();
48     {
49         sleep(1);
50         printf("Thread running: %i\n", i);
51     }
52     my_unlock();
53
54     semctl(semid, 0, IPC_RMID, 0);
55     return EXIT_SUCCESS;
56 }
```

Listing B.7: Beispiel: Shared Memory als Textspeicher (System V)

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/types.h>
5 #include <sys/ipc.h>
6 #include <sys/shm.h>
```

```
7
8  #define SHM_SIZE (1024)
9  #define perror_exit(x) do { perror((x)); exit(1); } ↩
       while (0)
10 #define ifpexit(cnd, x) do { if ((cnd)) { perror_exit(( ↩
       x)) } } while (0)
11
12 int main(int argc, char *argv[]) {
13     key_t   key;
14     int     shmid;
15     char*   data;
16     int     mode;
17
18     if (argc > 2) {
19         fprintf(stderr, "usage: shmdemo [data_to_write ↩
               ]\n");
20         exit(1);
21     }
22
23     ifpexit(((key = ftok("sysv_shm_simple.c", 'R')) == ↩
       -1), "ftok");
24     ifpexit(((shmid = shmget(key, SHM_SIZE, 0644 | ↩
       IPC_CREAT)) == -1), "shmget");
25
26     data = shmat(shmid, (void *)0, 0);
27     ifpexit((data == (char *)(-1)), "shmat");
28
29     if (argc == 2) {
30         printf("writing to segment: \"%s\"\n", argv[1]) ↩
               ;
31         strncpy(data, argv[1], SHM_SIZE);
32     } else {
33         printf("segment contains: \"%s\"\n", data);
34     }
35
36     ifpexit((shmdt(data) == -1), "shmdt");
37
38     return  EXIT_SUCCESS;
39 }
```

Listing B.8: Beispiel: Shared Memory mittels Semaphore synchronisiert

```
1 #define _GNU_SOURCE
2 #include <errno.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <sys/ipc.h>
7 #include <sys/sem.h>
```

```
8  #include <sys/shm.h>
9  #include <sys/types.h>
10 #include <unistd.h>
11
12 #define SHM_SIZE (1024)
13
14 #define print_exit(x) do { \
                                \
15         printf("Errno: %s, "x"\n", strerror(errno)); \
                \
16         exit(1); \
                                                        \
17     } while (0)
18
19 #define perror_exit(x) do { perror((x)); exit(1); } \
       while (0)
20
21 #define ifprintexit(cnd, x) do { if ((cnd)) { \
       print_exit(x); } } while (0)
22 #define ifpexit(cnd, x) do { if ((cnd)) { perror_exit(x\
       ); } } while (0)
23
24 static struct sembuf sop_lock[2] = {
25     { 0 /* Nr */, 0 /* Op (wait == 0) */, 0  /* Flags (\
          none) */ },
26     { 0 /* Nr */, 1 /* Op (+1) */, SEM_UNDO /* Flags (\
          undo changes on crash) */ },
27 };
28
29 static struct sembuf sop_unlock[2] = {
30     { 0 /* Number */, -1 /* Op (sub one) */, IPC_NOWAIT\
           /* Flags: Do not block */ },
31 };
32
33 struct sshm { // shared memory, synchronized via \
       semaphore
34     int semid;
35     key_t key;
36     int shmid;
37     void* shm;
38 };
39
40 void sshm_init(struct sshm* s) {
41     memset(s, 0, sizeof(*s));
42     s->semid = semget(123456L, 1, IPC_CREAT | 0666);
43     ifpexit((s->semid < 0), "semget");
44
45     s->key = ftok("sysv_shm_sync.c", 'R');
```

```
46      ifpexit((s->key == -1), "ftok");
47
48      s->shmid = shmget(s->key, SHM_SIZE, 0644 | IPC_CREAT);
49      ifpexit((s->shmid == -1), "shmget");
50  }
51
52  void sshm_att(struct sshm* s) {
53      s->shm = shmat(s->shmid, (void*)0, 0);
54      ifpexit((s->shm == (char *)(-1)), "shmat");
55      memset(s->shm, 0, SHM_SIZE);
56  }
57
58  void sshm_det(struct sshm* s) {
59      ifpexit((shmdt(s->shm) == -1), "shmdt");
60      s->shm = NULL;
61  }
62
63  void sshm_clear(struct sshm* s) {
64      ifpexit((semctl(s->semid, 0, IPC_RMID, 0) < 0), "semctl");
65      ifpexit((shmctl(s->shmid, IPC_RMID, NULL) < 0), "shmctl");
66      memset(s, 0, sizeof(*s));
67  }
68
69  void sshm_op(struct sshm* s, struct sembuf* op, int shmflg) {
70      ifprintexit((semop(s->semid, op, shmflg) < 0), "semop error");
71  }
72
73  void sshm_lock(struct sshm* s) {
74      sshm_op(s, sop_lock, 2);
75  }
76
77  void sshm_unlock(struct sshm* s) {
78      sshm_op(s, sop_unlock, 1);
79  }
80
81  int main(int argc, char** argv) {
82      struct sshm memory;
83
84      if (argc == 1) {
85          fprintf(stderr, "usage: %s [data_to_write]\n", argv[0]);
86          exit(1);
87      }
```

```
88
89      sshm_init(&memory);
90
91      int pid = fork();
92      if (pid == 0) {
93          // sender
94          sshm_att(&memory);
95
96          size_t ptr = 0;
97          for (int i = 0; i < argc; ++i) {
98              sleep(1);
99              sshm_lock(&memory);
100             // printf("Server, writing: '%s'\n", argv[i
                ]);
101             strcpy((char*) memory.shm + ptr, argv[i]);
102             // printf("Server, text: '%s'\n", (char*)
                memory.shm);
103             ptr += strlen(argv[i]);
104             sshm_unlock(&memory);
105         }
106     } else {
107         // receiver
108         sshm_att(&memory);
109
110         int lastlen = 0;
111
112         for (int i = 0;; ++i) {
113             sshm_lock(&memory);
114             sleep(1);
115             int currlen = strlen(memory.shm);
116             if (currlen == lastlen && currlen != 0) {
117                 break;
118             } else {
119                 lastlen = currlen;
120             }
121             printf("%i: %s\n", i, (char*) memory.shm);
122             sshm_unlock(&memory);
123         }
124     }
125
126     sshm_det(&memory);
127     return EXIT_SUCCESS;
128 }
```

Listing B.9: Beispiel: Shared Memory als Textspeicher (POSIX)

```
1 #include <stdlib.h>
2 #include <string.h>
3 #include <fcntl.h>
```

```
4 #include <sys/shm.h>
5 #include <sys/stat.h>
6 #include <sys/mman.h>
7 #include <unistd.h>
8 #include <stdio.h>
9
10 #define SHM_SIZE (1024)
11 #define NAME ("/SHARED_MEM_EXAMPLE_SIMPLE")
12
13 int main(int argc, char** argv) {
14     int i      = 0;
15     void* ptr = NULL;
16     int shm    = shm_open(NAME, O_CREAT | O_RDWR, 0666);
17
18     /* configure the size of the shared memory segment */
19     ftruncate(shm, SHM_SIZE);
20
21     ptr = mmap(0, SHM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm, 0);
22     if (ptr == MAP_FAILED) {
23         printf("Map␣failed\n");
24         exit(1);
25     }
26
27     if (argc == 1) {
28         printf("Memory:␣%s\n", (char*) ptr);
29     } else {
30         memset(ptr, 0, SHM_SIZE);
31         for (i = 0; i < argc; ++i) {
32             sprintf(ptr, "%s␣", argv[i]);
33             ptr += strlen(argv[i]) + 1;
34         }
35     }
36
37     munmap(ptr, SHM_SIZE);
38
39     return EXIT_SUCCESS;
40 }
```

Listing B.10: Beispiel: Ausgeben der Quellcodedatei mit “mmap”

```
1 #include <sys/stat.h>
2 #include <sys/mman.h>
3 #include <errno.h>
4 #include <string.h>
5 #include <fcntl.h>
6 #include <stdio.h>
7 #include <stdlib.h>
```

```c
#include <ctype.h>

#define ifpexit(cnd, x) do { if ((cnd)) { perror((x)); exit(1); } } while (0)

int main () {
    struct stat s;
    int         status;
    size_t      size;
    char*       mapped;
    int         i;
    int         fd = open("./mmap_readfile.c", O_RDONLY);

    ifpexit(fd < 0, "open");

    status = fstat(fd, & s);
    ifpexit(status < 0, "stat");
    size = s.st_size;

    mapped = mmap(0, size, PROT_READ, MAP_PRIVATE, fd, 0);
    ifpexit(mapped == MAP_FAILED, "mmap");

    printf("%s", mapped);
    munmap(mapped, size);
    return 0;
}
```