

Einführung in die Programmierung mit C

Teil 3

Dieter Kranzlmüller

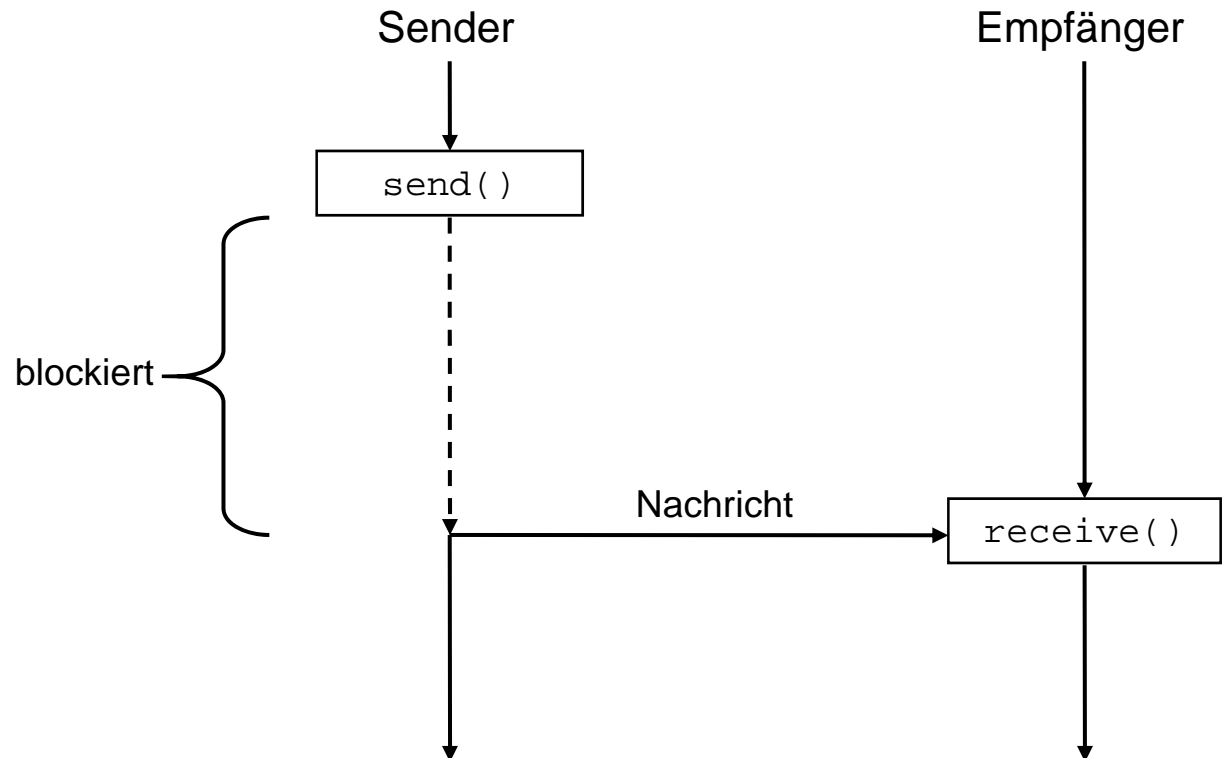
Nils gentschen Felde

Heute: Christian Nietschke



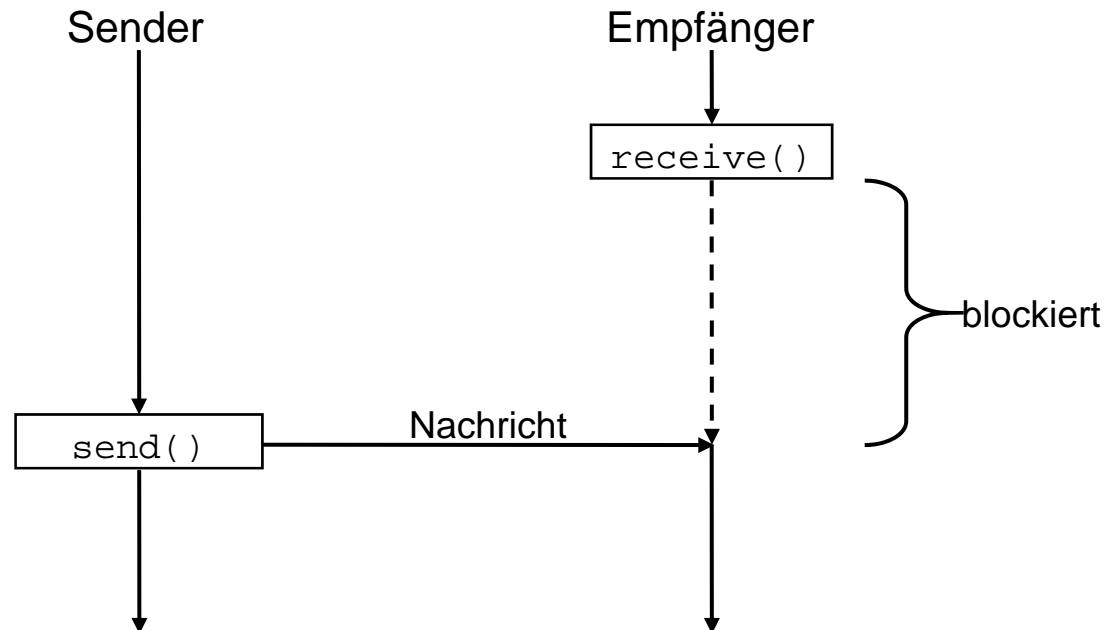
- Blockierendes (synchrones) Senden

- Sender wartet, bis er Antwort auf Nachricht erhält
- Ausbleiben der Nachricht: Timeout/Fehler
- z.B. für RPC



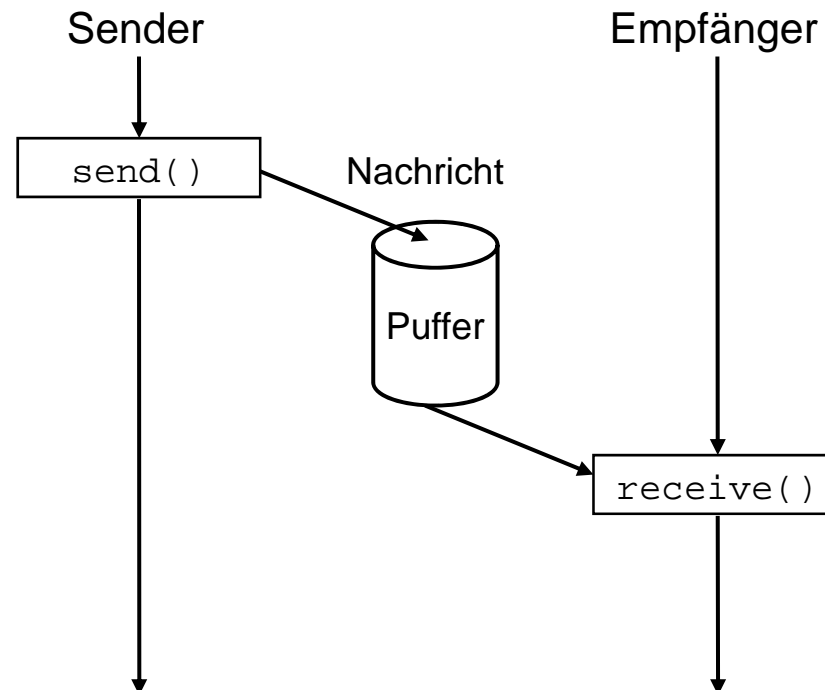
• Blockierendes (synchrones) Empfangen

- Warten auf Empfangen der Nachricht
- Blocking read/synchronous receive
- Prozess wartet, bis Daten verfügbar sind, arbeitet diese ab, wartet wieder



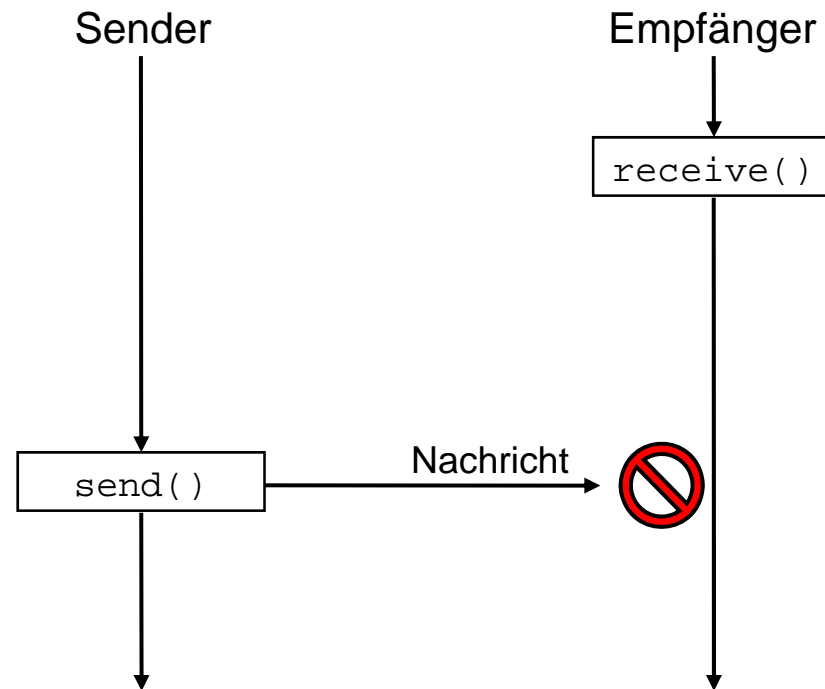
• Nicht-blockierendes Senden

- Zeitlich versetzte Kommunikation
- Sende-Prozess blockiert nicht



• Nicht-blockierendes Empfangen

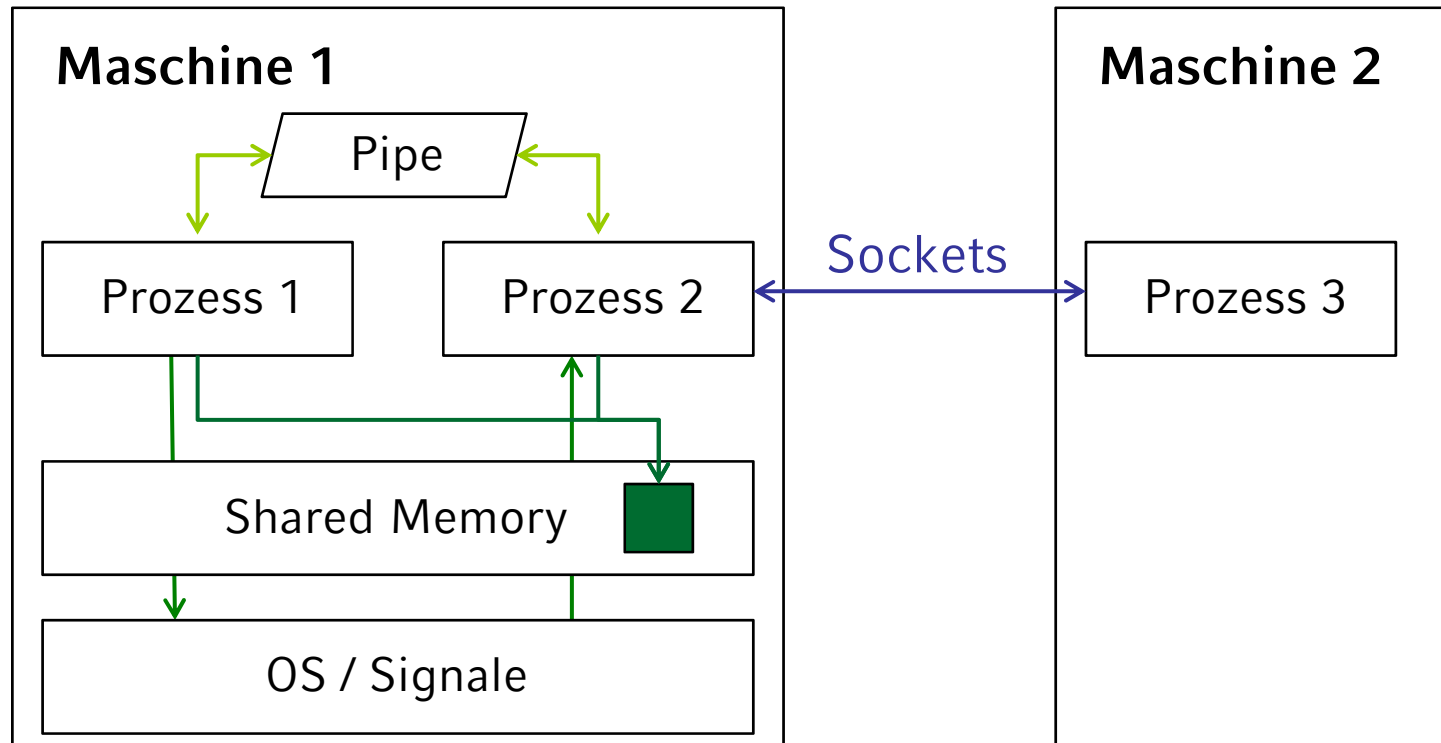
- non-blocking read/asynchronous receive
- Fehler, wenn keine Daten im Eingangsbuffer liegen
- Prozess prüft regelmäßig, ob Datenvorhanden sind



	Blockierendes Empfangen	Nicht-blockierendes Empfangen
Blockierendes Senden	Synchrone Kommunikation (Rendezvous-Konzept)	keine zuverlässige Kommunikation
Nicht-blockierendes Senden	+ Nachrichtenbuffer -> Asynchrone Kommunikation	keine zuverlässige Kommunikation

- Protokoll definiert Regeln für den Informationsaustausch
- Teile einer Protokolldefinition:
 - Verwendete Codes
 - Nachrichtenlänge
 - Nachrichtenformat
 - Form der Adressierung
 - Bestätigungen
 - Fehlercodes
 - u.a.

➤ Kapitel 9



- Basiert auf Datenströmen (Streams)
 - Strom: Objekt, in das Informationen geschrieben und gelesen werden können
- Pipe
 - Zwei-Wege-Datenstrom
 - Unidirektionale Kommunikation
 - Nachrichtenreihenfolge bleibt erhalten (FIFO)
 - Zwei Arten: Named und unnamed Pipes

- Eigenschaften

➤ Kapitel 9.1.1

- Aufbau von unnamed Pipes nur zwischen "verwandten" Prozessen
- Existiert anonym und nur so lange wie Prozess(e) existieren
- Kindprozesse erben Pipes des Elternprozesses

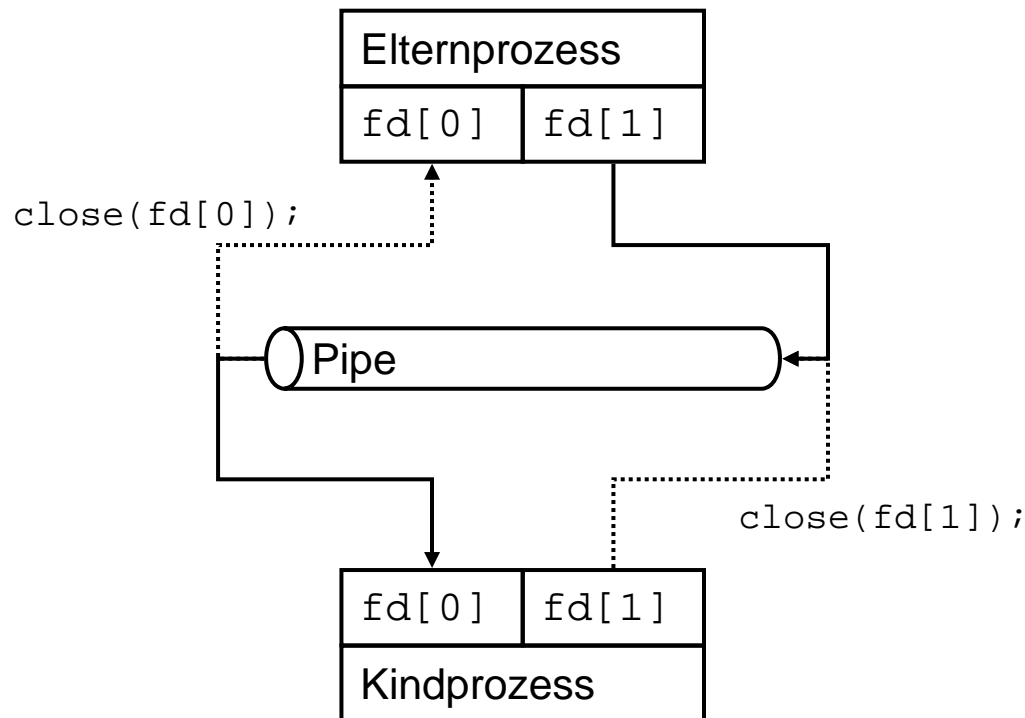
```
int pipe(int fd[2]);  
// fd[0]: Filedeskriptor zum Lesen (Leseseite)  
// fd[1]: Filedeskriptor zum Schreiben (Schreibseite)  
// Rückgabe bei Erfolg 0, im Fehlerfall -1
```

```
close(fd[n]);  
// Schließen eines Pipe-Filedeskriptors
```

<http://pubs.opengroup.org/onlinepubs/009695399/functions/<Funktionsname>.html>

IEEE Std 1003.1-2001 ... defines a standard operating system interface and environment, including a command interpreter (or "shell"), and common utility programs to support applications portability at the source code level.

- Pipe zur Interprozesskommunikation
(nach (!) fork())



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

int main() {
    pid_t pid;
    int fd[2], n = 5;
    char string[] = "abc\n\n";

    if (pipe(fd) < 0) {
        perror ("Fehler beim Einrichten der Pipe.");
        exit(EXIT_FAILURE);
    }

    if ((pid = fork ()) < 0) {
        perror ("Fehler bei fork().");
        exit(EXIT_FAILURE);
    }
}
```

```
else if (pid > 0) {
    /*im Elternprozess */
    close(fd[0]); //Leseseite schließen
    if ((write (fd[1], string, n)) != n) { //In Schreibseite schreiben
        perror ("Fehler bei write().");
        exit (EXIT_FAILURE);
    }
    /* Warten auf den Kindprozess */
    if ((waitpid (pid, NULL, 0)) < 0) {
        perror ("Fehler beim Warten auf Kindprozess.");
        exit (EXIT_FAILURE);
    }
}
else {
    /*im Kindprozess */
    close(fd[1]); //Schreibseite schließen
    n = read (fd[0], string, PIPE_BUF); //Leseseite auslesen
    if ((write (STDOUT_FILENO, string, n)) != n) {
        perror ("Fehler bei write().");
        exit (EXIT_FAILURE);
    }
}
exit (EXIT_SUCCESS);
}
```

• Eigenschaften

➤ Kapitel 9.3

- Kann ähnlich wie eine normale Datei behandelt werden: Named Pipeline nutzt das Dateisystem (spezielle FIFO-Datei)
- Ist System-persistent und existiert über Lifetime der Prozesse hinaus → Muss explizit gelöscht werden
- **Kommunizierende Prozesse brauchen keine gemeinsamen Vorfahren**
- **Mehrere Prozesse können in die named Pipe schreiben**

```
mkfifo(pfad, zugriffsrechte);  
// Rückgabe bei Erfolg 0, im Fehlerfall -1  
// Öffnen einer Named Pipe  
fd = open(pfad, modus);  
// Schließen einer Named Pipe  
close(fd);
```

- Normalerweise sind die Speicherbereiche zweier Prozesse streng getrennt!
→ Es ist ein gesonderte Speicherbereich notwendig, das **Shared Memory Segment**
- Vom BS-Kern verwalteter Speicherbereich
- Kann von mehreren Prozessen gelesen und beschrieben werden
- Synchronisation der Zugriffe erforderlich

➤ Kapitel 9.7

Schritt	Funktion	Aufgabe
1	<code>shmget ()</code>	Gemeinsamen Speicher anlegen/öffnen → Prozess erzeugt Shared Memory-Segment, legt Größe und Name/Schlüssel fest
2	<code>shmat ()</code>	Erzeugtes Segment an den Adressraum des Erzeuger-Prozesses anhängen (shared memory attach)
3		Anderer Prozess kann das Segment mitbenutzen, wenn er den Schlüssel kennt
-	<code>shmctl ()</code>	Eigenschaften des Shared Memory Segments verändern
4	<code>shmdt ()</code>	Speicherbindung wieder aufheben (shared memory detach)

Schritt 1 - Neues Segment anlegen oder auf bestehendes zugreifen

```
int shmget(key_t key, int size, int flag)
// Rückgabewert: -1 im Fehlerfall oder die Shared Memory ID
// key: Schlüssel für Segment oder IPC_PRIVATE
// size: Größe in Byte
// flag: Setzt Berechtigungen, z.B. IPC_CREAT | 0644
```

Schritt 2 – Segment an Adressraum des Erzeugerprozesses anhängen

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
// Rückgabewert: Pointer auf die Anfangsadresse im Adressraum
// shmid: ID des Segments (Rückgabewert von shmget())
// shmaddr: Adresse für Einbindung (0 = automatisch)
// shmflg: Zugriffsberechtigungen, SHM_RDONLY ist read only
```

Schritt 3 – Gemeinsamen Speicher nutzen

[Optional] – Eigenschaften des gemeinsamen Speichers verändern

```
int shmctl(int shmid, int kommando, struct shmid_ds *buf);  
// shmid: ID des Segments (Rückgabewert von shmget())  
// kommando: Konstanten, um Zugriffsrechte usw. zu ändern  
// buf: Zieldatenstruktur für Statistiken usw.
```

Schritt 4 – Speicheranbindung wieder entfernen

```
int shmdt(const void *shmaddr);  
// Rückgabewert: -1 im Fehlerfall, ansonsten 0  
// shmadr: Adresse des Segments
```

Kritischer Bereich

- Ein kritischer Bereich ist der Teil in einem Prozess, der auf gemeinsam genutzte Betriebsmittel zugreift.
- In ihm darf sich zu einem Zeitpunkt nur ein Prozess/Thread aufhalten

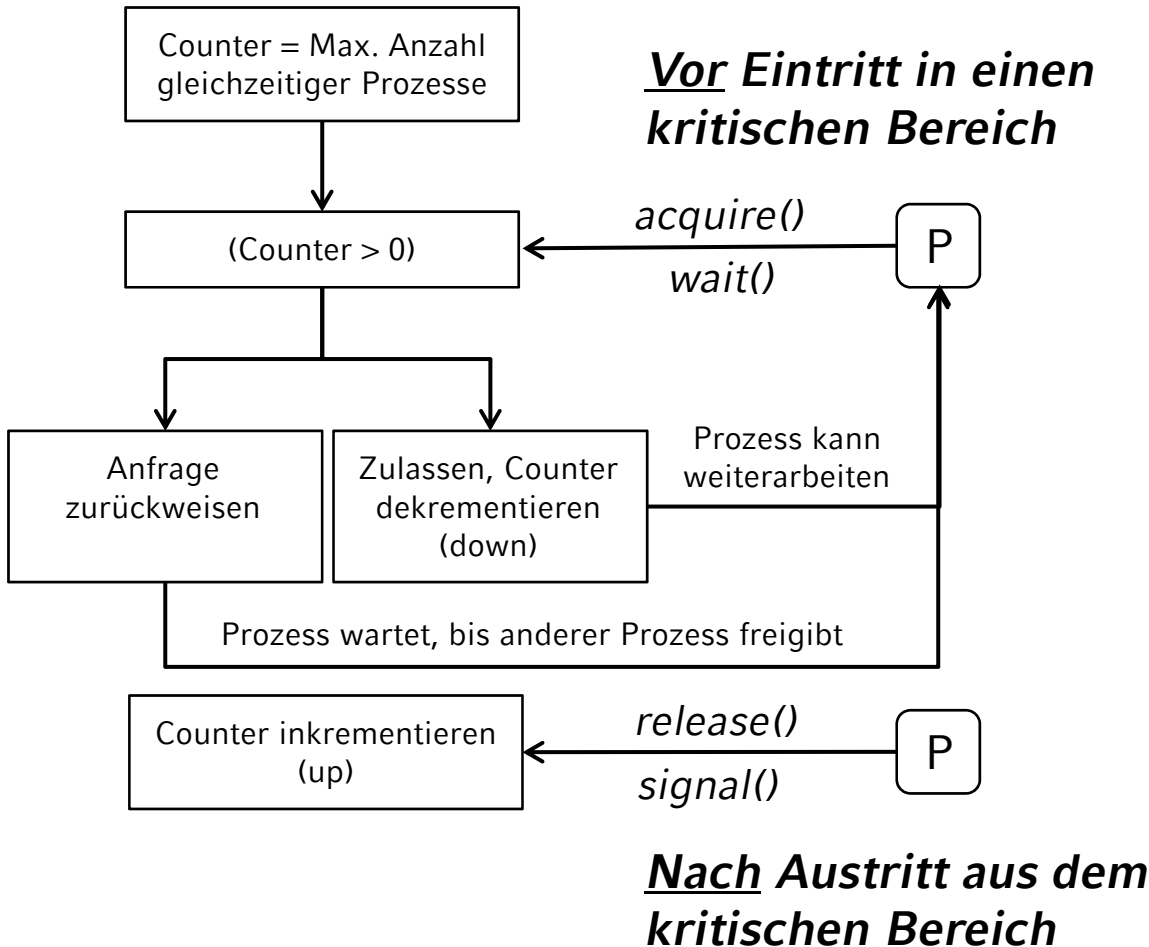
Race Condition

Die Korrektheit einer Berechnung hängt vom relativen Timing mehrere Threads zur Laufzeit ab

- Datenstruktur
- Überwacht, wie viele Akteure im kritischen Bereich sind bzw. wie viele noch hinein dürfen
- Besteht meist aus einem ganzzahligen Zähler und einer Akteur-Warteschlange

Semaphore

Prozesse



**Synchronisiert Zugriff
auf exklusive
Ressourcen**

- Unix-Semaphore ist allgemeine Semaphore (Zählsemaphore)
- Anlegen einer Gruppe von Semaphoren durch einen einzigen Aufruf
- Für eine Gruppe von Semaphoren kann eine Operationsfolge definiert werden, die logisch-atomar ausgeführt wird

➤ Kapitel 9.5

Semaphoren-Array anlegen oder auf ein bestehendes zugreifen

```
int semget(key_t key, int nsems, int semflg);  
// Rückgabewert: -1 im Fehlerfall, Identifikator des Arrays  
// key: Schlüssel für die Gruppe, IPC_PRIVATE (automatisch)  
// nsems: Anzahl der Semaphoren in der Gruppe  
// semflg: IPC_CREAT | 0644
```


Steuerungsfunktionen auf Semaphoren-Gruppe ausführen

```
int semctl(int semid, int semnum, int cmd, union semun args);  
// Rückgabewert: -1 im Fehlerfall, ansonsten 0  
// semid: ID des Semaphoren-Arrays (von semget)  
// semnum: Anzahl der Semaphoren im Array  
// cmd: Flag aus sys/sem.h, z.B. SETALL oder GETALL  
// args: Input bzw. Output-Vektor, abhängig von cmd
```


Semaphoren-Werte innerhalb einer Gruppe verändern

```
int semop(int semid, struct sembuf *sops, size_t nsops);  
// Rückgabewert: -1 im Fehlerfall, ansonsten 0  
// semid: ID des Semaphoren-Arrays (von semget)  
// nsops: Anzahl der Strukturen im Array  
// sops: Pointer auf ein Array mit Strukturen aus  
Semaphoren-Operationen
```

**Führt
Operationen
atomar aus**



```
struct sembuf {  
    short sem_num;  
    short sem_op;  
    short sem_flg;  
};
```

sem_num: Nummer des Semaphors in der Gruppe

sem_op: auszuführende Operation (-1 = wait(), 1 = signal())

sem_flg: Flags zur Steuerung der Operation

semaphore.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define NSEMS 3
#define WAIT -1
#define SIGNAL 1

int main() {
    /*Semaphorengruppe mit NSEMS (=3) Semaphoren anlegen*/
    int id = semget(IPC_PRIVATE, NSEMS, IPC_CREAT | 0770);
    printf("Identifizier der Gruppe: %i\n", id);

    /*Alle Semaphore der Gruppe initialisieren*/
    unsigned short init[NSEMS];
    int i;
    for(i=0; i<NSEMS; i++) {
        init[i] = 1;
    }
    semctl(id, NSEMS, SETALL, init);
}
```

semaphore.c

```
/*Werte der Semaphore abfragen*/
unsigned short out[NSEMS];
semctl(id, NSEMS, GETALL, out);
printf("Werte des 1. und 2. Semaphors: %i, %i\n", out[0], out[1]);

/*wait()/down()-Operation auf erstem Semaphor der Gruppe ausführen*/
struct sembuf wait;
wait.sem_num = 0;
wait.sem_op = WAIT;
wait.sem_flg = 0;

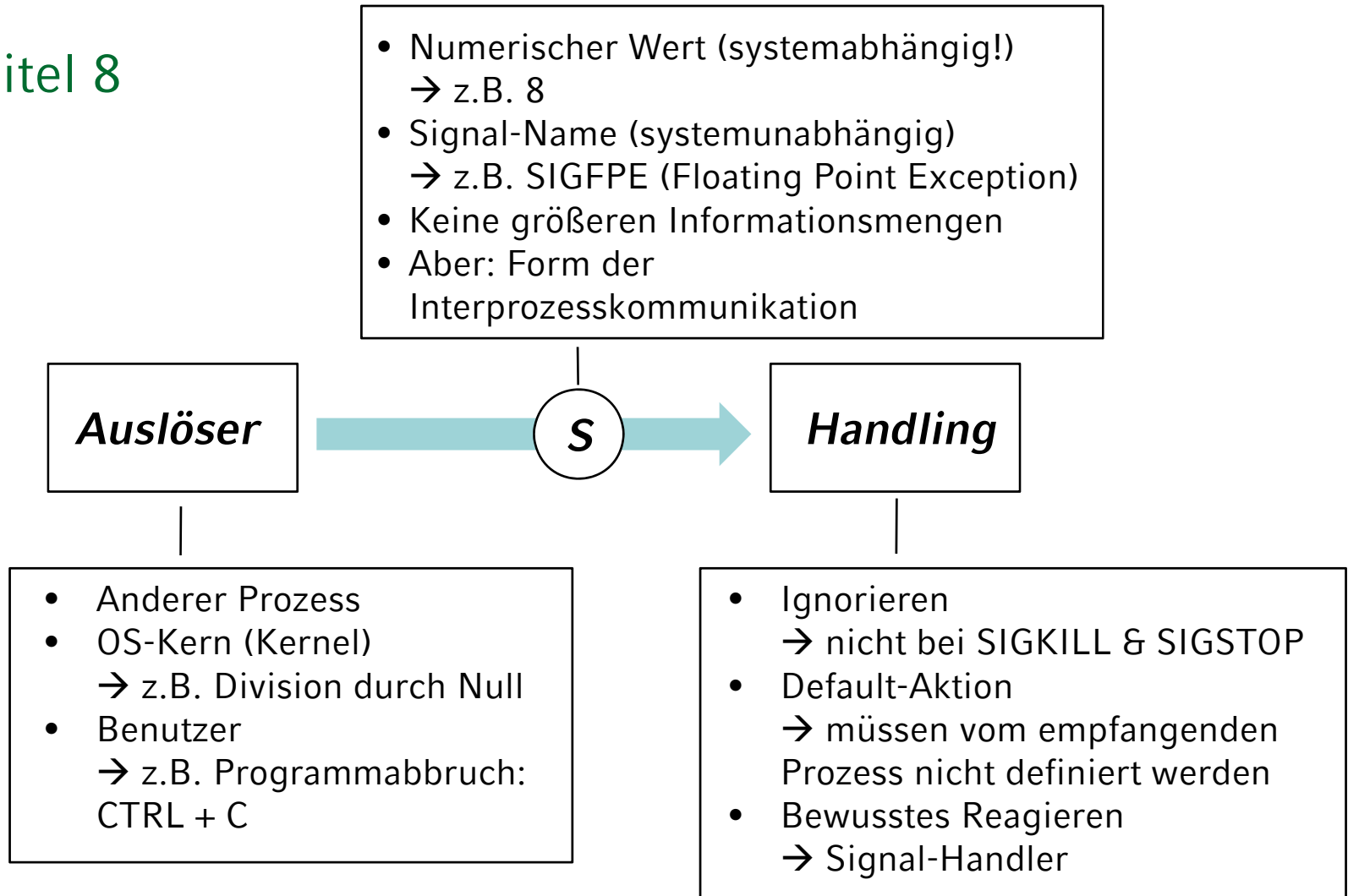
semop(id, &wait, 1);

/*Werte der Semaphore erneut abfragen*/
semctl(id, NSEMS, GETALL, out);
printf("Wert des 1. und 2. Semaphors: %i, %i\n", out[0], out[1]);

/*Semaphorengruppe löschen*/
semctl(id, NSEMS, IPC_RMID, 0);

return EXIT_SUCCESS;
}
```

➤ Kapitel 8



Systemseitige Signalbehandlung/Signalverarbeitung

1. Signal(ereignis) tritt ein
2. Hinterlegung des Signals im Prozesstabelleneintrag des empfangenden Prozesses → ***pending signal***
3. Signal wird im Kontext des Prozesses verarbeitet, wenn der Prozess wieder die CPU-Kontrolle erhält (Scheduling)
→ genauer: vor dem Wechsel vom Kernel in den User Mode

SIGCHLD	Wird an den Elternprozess geschickt, wenn einer seiner Kindprozesse terminiert
SIGCONT	Wird an einen angehaltenen Prozess gesendet, wenn dieser seine Ausführung fortsetzen soll
SIGFPE	Wird bei einem arithmetischen Fehler (z.B. Division durch 0) geschickt
SIGINT	Wird allen aktiven "Vordergrundprozessen" geschickt, wenn die Unterbrechungstaste/-tastenkombination (i.d.R. STRG + C) gedrückt wird
SIGKILL	Beendet einen Prozess
SIGPIPE	Wird einem Prozess geschickt, der versucht, in eine Pipe zu schreiben, zu der es keinen offenen Lese-Filedeskriptor gibt
SIGSTOP	Hält einen laufenden Prozess an

Ausgewählte Signale (signal.h) → insgesamt ca. 30 Signale

- Zusammenhang zwischen Signalen und Kindprozessen
 - Kindprozess erbt Signal-Handler des Elternprozesses
 - Zombie-Prozesse ohne Verwendung von `wait()/waitpid()` verhindern: Signal `SIGCHLD` im Elternprozess ignorieren

Das Signalkonzept im C-Standard

- `void (*signal(int sig, void (*func)(int)))(int);`
Signal-Handler festlegen, d.h. auf Signale reagieren
- `int kill(pid_t pid, int sig);`
Senden von Signalen an Gruppe anderer Prozesse, festgelegt durch pid
- `int raise(int sig);`
Senden von Signalen an den ausführenden Prozess
- Bemerkung:
- `raise(signr);` ist identisch mit `kill(getpid(), signr);`

Beispiel: SIGINT – Varianten der Default-Behandlung

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

void mysighandler(int signalKey) {
    switch(signalKey) {
        case SIGINT:
            // Programmode
            break;
    }
}

int main(int argc, char *argv[]) {
    // Eigenen Signal-Handler verwenden
    signal(SIGINT, mysighandler);

    //Signal ignorieren
    signal(SIGABRT, SIG_IGN);

    // Reguläre Signalbehandlungsfunktion
    // verwenden (default signal handler)
    signal(SIGINT, SIG_DFL);
}
```

Das Signalkonzept im POSIX-Standard

➤ Kapitel 8.2

- Signalmengen
- Einrichten und Erfragen von Signal-Handlern mit `sigaction()`
- Vermeidet einige Nachteile, die C-Standard Implementierung hat
- Grundlegende Idee: Signalmengen mit eigenem Datentyp `sigset_t`

1. Leeres Signal-Set erzeugen

```
int sigemptyset(sigset_t *set);
```

2. Signalmenge manipulieren

```
int sigfillset(sigset_t *set);
```

Fügt alle vorhandenen Signale zur Signalmenge hinzu

```
int sigaddset(sigset_t *set, int signo);
```

Fügt das gegebene Signal signo zum Signalset hinzu

```
int sigdelset(sigset_t *set, int signo);
```

Löscht das angegebene Signal aus der Signalmenge

```
int sigismember(const sigset_t *set, int signo);
```

Prüfen, ob das gegebene Signal in der Menge enthalten ist

3. Auf Signale reagieren

```
int sigaction(  
    int sig,  
    const struct sigaction *restrict act,  
    struct sigaction *restrict oact);
```

Der aufrufende Prozess kann auslesen bzw. festlegen, welche Aktionen zu einem Signal zugewiesen ist

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

- sa_handler: Adresse des Signal-Handlers, SIG_IGN, SIG_DFL
- sa_mask: Zusätzliche Menge an zu blockierende/sperrende Signale während der Signalbehandlung
- sa_flags: Signalooptionen, z.B. SA_SIGINFO, SA_NOCLDSTOP, SA_RESTART

- Informationen zwischen (entfernten) Rechnern/Prozessen austauschen ➤ Kapitel 11
- Definition: Socket
 - Kommunikationsendpunkt auf Softwareebene
 - spezifische Schnittstelle zwischen einem Anwendungsprogramm
- Socket-Arten
 - Stream-Socket (SOCK_STREAM): Zuverlässiger, verbindungsorientierter Bytestrom mit Sequencing und Fehlerkorrektur
 - Datagramm-Socket (SOCK_DGRAM): Unzuverlässige, verbindungslose Paketübertragung ohne Sequencing und ohne Fehlerkorrektur

- Socket-Adressierung

- Socket-Domäne: vom Socket verwendete Protokollfamilie
→ Nur Sockets der gleichen Domäne können miteinander kommunizieren
- Domänen:
 - PF_UNIX, PF_LOCAL: Unix-Adresse → Nachteil: keine entfernte Kommunikation
 - PF_INET: IP-Adresse (IPv4: 32 Bits) + Portnummer (16 Bits)
 - PF_INET6: IP-Adresse (IPv6: 128 Bits) + Portnummer (16 Bits)

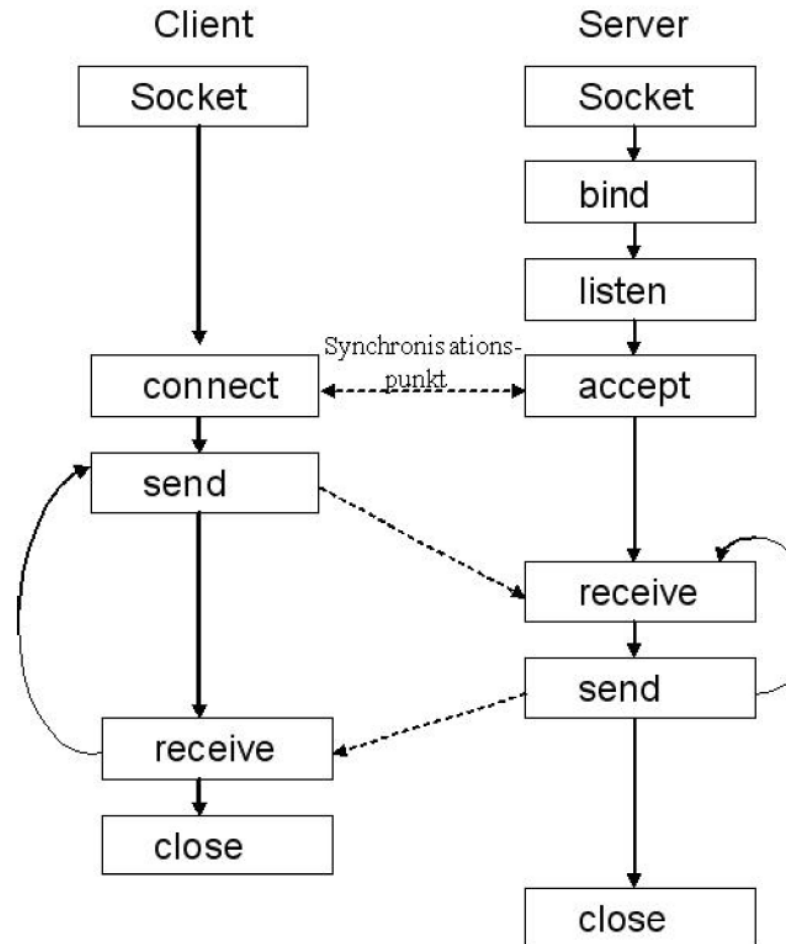
- Wir betrachten: Sockets der Internet-Domäne (Internet-Sockets)

- Die Struktur struct sockaddr_in

```
struct sockaddr_in {  
    short sin_family;  
    unsigned short sin_port;    // Portnummer  
    struct in_addr sin_addr;    // Struktur für IP-Adresse  
    char sin_zero[8];          // zum Auffüllen der Struktur,  
                                // damit alle Socket-Adressen  
                                // mind. 16 Bytes groß sind  
};  
  
struct in_addr {  
    unsigned long saddr;        // IP-Adresse  
};
```

- wird durch Einbinden von netinet/in.h verfügbar
- Client/Server-Prinzip
 - Client und Server sind Rollen
 - Server: bietet (mindestens) einen Dienst an, wartet passiv auf eine Anfrage
 - Client: sucht aktiv nach einem Server, dessen Dienst er benötigt

Systemaufrufe für Sockets



Funktionssignaturen:
siehe man-Pages

Beispiel: Iterativer Socket-Server (socksrv.c)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main() {

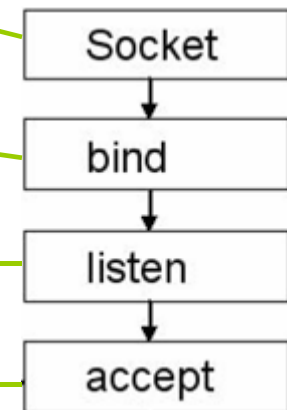
    // 1. Socket anlegen
    int sock = socket(PF_INET, SOCK_STREAM, 0);

    // 2. Binden einer Adresse an den Socket
    struct sockaddr_in server;
    server.sin_family = PF_INET;
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = htons(4711);
    bind(sock, (struct sockaddr *) &server, sizeof(server));

    // 3. Auf Anfragen warten
    listen(sock, 5);

    // 4. Verbindungen akzeptieren
    struct sockaddr_in client;
    int fd, client_len;
    client_len = sizeof(client);
    fd = accept(sock, (struct sockaddr *) &client, &client_len);

    return 0;
}
```



- Weiterführende Informationen zu Prozessen, Pipes, Shared Memory, Semaphoren, Signalen, Sockets:
- www.cs.cf.ac.uk/Dave/C/
- pubs.opengroup.org/onlinepubs/009695399/
- openbook.galileocomputing.de/unix_guru/node393.html

- Datum: 27. Oktober 2014, 18:30 Uhr
- Einlass: 18:15 Uhr
- Raum: A240 und M218,
LMU Hauptgebäude
- Erlaubte Hilfsmittel: keine
- Bearbeitungszeit: 60 Minuten
- Bitte Studenten- und amtlichen Lichtbildausweis mitbringen! Ohne Dokumente ist keine Teilnahme möglich!