

# I. Maschinenprogrammierung in Assembler

## Definition: Assemblersprache

Assemblersprachen sind maschinenorientierte Programmiersprachen, die sehr nahe an der Maschinensprache ist.

Ein Assembler übersetzt in ein Objektprogramm:

- 1. Durchgang:**
- Bestimmung der Länge von Maschineninstruktionen
  - Verwaltung eines Adresszählers (Befehle und Daten)
  - Zuordnung symbolischer Adressen
  - Zuordnung von Literalen
  - Verarbeitung einiger Assembler-Instruktionen

- 2. Durchgang:**
- Heranziehung der Symbolwerte (= Speicheradressen)
  - Erzeugung von Maschineninstruktionen
  - Erzeugung von Daten (= Konstanten)
  - Verarbeitung der restlichen Assembler-Instruktionen

Im Folgenden wird die X86-64-Architecture verwendet in der Intel-Syntax

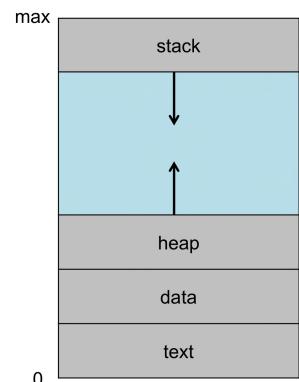
**Protected Mode:** Segmentregister werden automatisch gesetzt

## Arbeitsspeicherstruktur

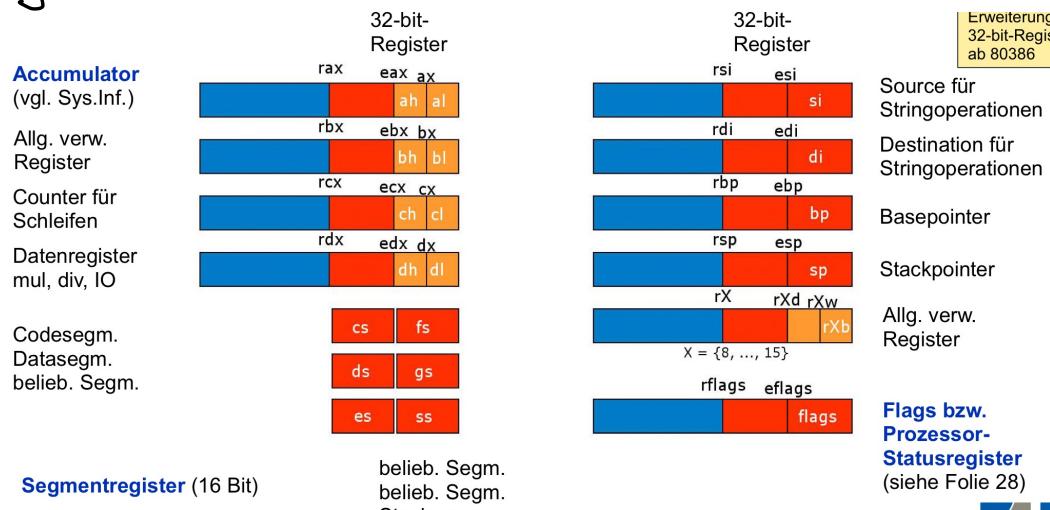
### Speichergröße:

word = 2 Byte  
double word = dword = 4 Byte  
quad word = 8 Byte  
paragraph = 16 Byte

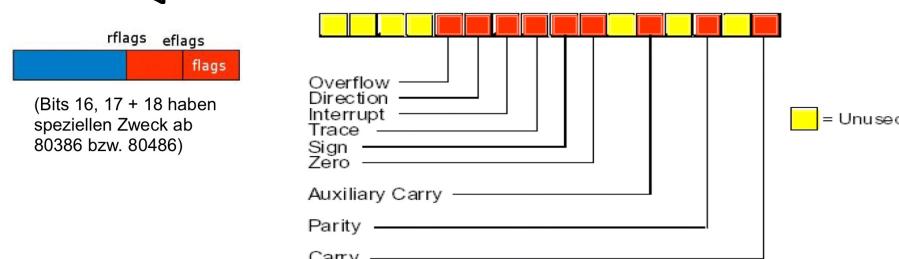
**Bytereihenfolge:** Little-Endian



## Register



## Statusregister:



**Control Flags:**  
(Einfluss auf Ausführung von Instruktionen)

D: Direction, String Operation  
1 von hohen nach niedrigen Adressen  
I: Interrupt, 1 = Interrupt enabled  
T: Trap/Trace, Debugging  
Single Step/Trace Modus

**Status Flags:**  
(Ergebnis der Ausführung von Instruktionen)

C: Carry, Unsigned Op. zu groß oder < 0  
O: Overflow, Signed Op. zu groß oder zu klein  
S: Sign, Vorzeichen 1 = neg. / 0 = pos.  
Z: Zero, 1 = Ergebnis ist null  
A: Aux. Carry, für BCD-Operationen  
P: Parity, 1 = Ergebnis hat gerade Anzahl

# Klassen von Operationen

1. Data movement instructions – **Verschieben von Daten in und von Registern**  
mov, lea, les, push, pop, pushf, popf
2. Conversions – **Konvertierungen Byte, Word, Double, Extended Double**  
cbw, cwd, cwde
3. Arithmetic instructions – **Arithmetik**  
add, inc, sub, dec, cmp, neg, mul, imul, div, idiv
4. Logical, shift, rotate and bit instructions – **Logik, Bit-Operationen**  
and, or, xor, not, shl, shr, rcl, rcr
5. I/O instructions – **Input/Output**  
in, out
6. String instructions – **String Verarbeitung**  
movs, stos, lods
7. Program flow control instructions – **Programmfluss, Sprungbefehle**  
jmp, call, ret, (conditional jumps)
8. Miscellaneous instructions – **Verschiedenes**  
clc, stc, cmc

## Hello-World Beispiel

Der Klassiker: „Hello World!“	Erläuterung
# syscall „write“ %macro write 3 mov rax, 1 mov rdi, %1 mov rsi, %2 mov rdx, %3 syscall %endmacro	Makro: An der aufrufenden Stelle wird Textersetzung durch die Instruktionen des Makros vorgenommen  hier: Aufruf der Systemfunktion write
SECTION .data hello: db „Hello World!“, 10 helloLen: equ \$ - hello	Datenteil des Assembler-Programms, z.B. String-Konstanten, globale Variablen, ... Eigene Adresse – Adresse Label „hello“
global _start	Für den Linker sichtbares Label.
SECTION .text _start: write 1, hello, helloLen  mov rax, 60 mov rdi, 0 syscall	Textteil des Assembler-Programms, das eigentliche Programm selbst. _start ist Label für Start des Hauptprogramms.

## Addressierungsarten

Registeradressierung:

mov rbx, rdi

Unmittelbare Adressierung:

mov rbx, 1000

Direkte Adressierung:

mov rbx, [1000]

Register-indirekte Adressierung:

mov rbx, [rax]

Basis-Register Adressierung:

mov rax, [rsi+10]

Allgemeine Indizierung:

mov register, segreg:[base+index\*scale+disp]

[reg]: Wert in der Speicherstelle, die in Reg abgespeichert ist

# Anwendungen

## if - then - else:

Übertragung in Assembler:

```

        cmp rax, 4711
        jne ungleich

gleich: ...
        jmp weiter

ungleich: ...

weiter: ...
    
```

## for-Schleife:

Übertragung in Assembler:

```

        mov rcx, 100
schleife: add rax, [a]
        loop schleife
    
```

## ohne loop:

Bilde Summe von 100 beliebigen Zahlen (Array):

```

        mov rcx, 0
        mov rax, 0
        mov rbx, array
schleife: add rax, [rbx+rcx*2]
        inc rcx
        cmp 100, rcx
        jne schleife
    
```

loop dekrementiert rcx und hat seinen letzten Durchlauf bei rcx = 1

```

# Initialisiere Index mit 0
# Initialisiere Summe mit 0
# Arrayadresse in Basis
# Array-Durchlauf in 2er Schritten (aufwärts)
# erhöhe Index
# Ende: Index == 99 + 1 ?
# wenn gleich, Schleifenende
    
```

# Calling-Conventions

## Aufruf einer Funktion:

```

        mov rbx, input1
        call get_int
        mov rbx, input2
        call get_int
    
```

## Unterfunktion:

```

# void stars(int low, int high)
stars:
    # set up stack frame
    push ebp
    mov ebp, esp
    # reserve space for local variable i
    sub esp, 4
    ... # rest of function stars

    # reconstruct stack- and base-pointer
    leave

    # return
    ret
    
```

Retten des Basepointers und Aufbau eines neuen Stacks  
Platz für lokale Variablen reservieren

...	
Parameter high	ebp + 12
Parameter low	ebp + 8
Rücksprungadresse	esp + 8
gerettetes ebp	esp + 4
lokale Variable i	esp

## Parameterübergabe

cdecl: Parameterübergabe über den Stack

AMD64 ABI: Parameterübergabe über Register rdi, rsi, rdx, rcx, r8, r9

## Verwendbare Register:

cdecl: eax, ecx, edx

Registers, die unverändert bleiben sollen: ebx, esi, edi, ebp, cs, ds, ss, es

AMD64 ABI: rax, rcx, rdx, rsi, rdi, r8-r11

Registers, die unverändert bleiben sollen: rbx, rsp, rbp, r12-r15

→ Register müssen ggf. wieder hergestellt werden

Rückgabe: über eax bzw. rax

## Befehlsformate

3-Adress-Format: Op-Code Quelle 1 Quelle 2 Ziel

2-Adress-Format: Op-Code Quelle 1 Quelle 2 + Ziel  
(„Überdeckung“: Der zweite Operand wird zerstört)

1-Adress-Format: Op-Code Quelle + Ziel  
(„Überdeckung“: Der Operand wird zerstört;  
ggf. zusätzlich „Implizierung“)

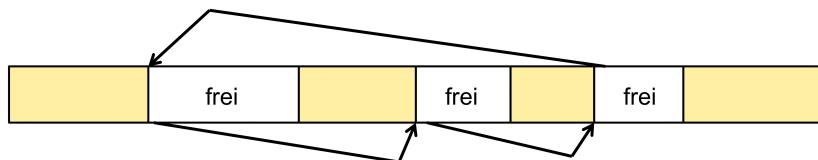
## Speicherverwaltung

Umsetzung durch `malloc()` und `free()`

### Free-Liste

`malloc`/`free` verwalten den freien Speicher in einer zyklisch verkettenen Liste

Jedem freien Speicherblock geht ein Header voraus, der folgendes enthält:  
1. die Größe des freien Speicherblocks (ohne Header)  
2. einen Zeiger auf den nächsten freien Speicherblock  
→ letzter Zeiger zeigt auf den ersten Block



### Alloc

#### Verfahren:

1. Suche in der Free-Liste einen groß genugem Block (first-fit)
2. Wird einer gefunden: Dann schneide benötigten Speicher von hinten ab

Wird keiner gefunden: Neuer Speicher wird vom OS angefordert und in die Free-Liste eingefügt

**Header:** Größe des Blocks, Zeigerfeld wird nicht verwendet

**Rückgabe:** Zeiger auf erstes Byte hinter dem Header

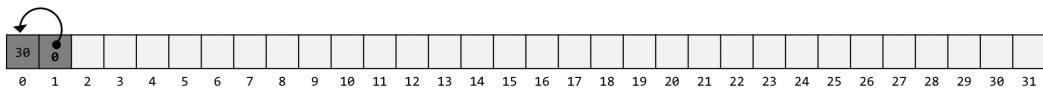
### Free

Mithilfe des Alloc-Headers wird ein neuer Free-Block erzeugt und in die Liste eingefügt.

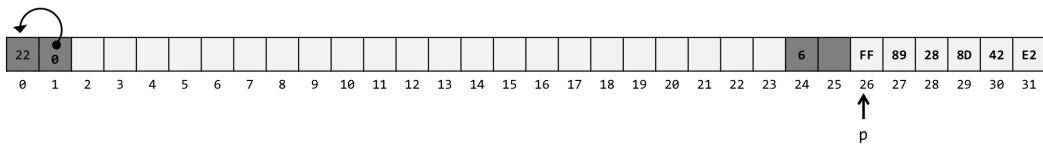
Direkt angrenzende Blöcke werden zusammengefasst.

## Beispiel:

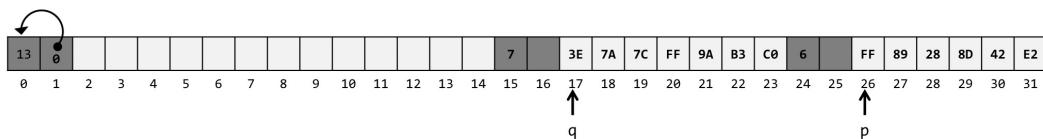
- 32 Byte großer Speicher, 1 Byte lange Adressen (0-31)
- Zu Beginn ein einziger, großer, freier Block:



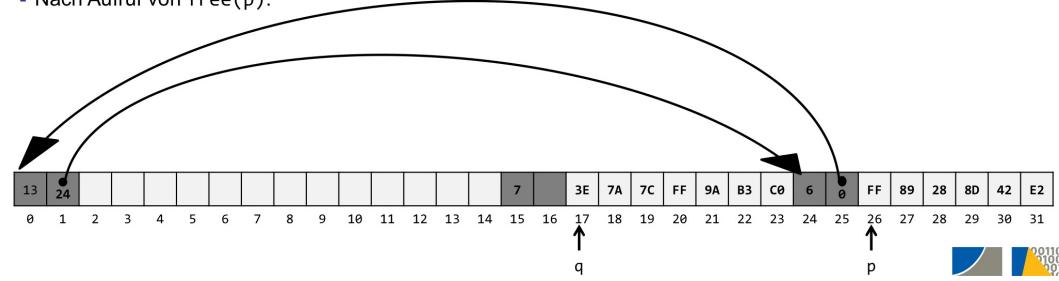
- Nach Aufruf von malloc(6) (Rückgabewert: p) und schreiben von FF 89 28 8D 42 E2 an die Stelle, auf die p zeigt:



- Nach Aufruf von malloc(7) (Rückgabewert: q) und schreiben von 3E 7A 7C FF 9A B3 C0 an die Stelle, auf die q zeigt:



- Nach Aufruf von free(p):



## 2.1 Prozesse

### Prozess im Speicher

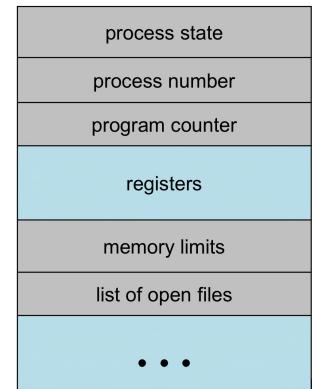
**kernel-mode Prozesse:** können auf alles zugreifen

**User-mode Prozesse:** können nur auf den ihnen zugewiesenen Speicher zugreifen

### Process Control Block (PCB)

Speichert Information über den jeweiligen Prozess für das Betriebssystem ab:

- Process state (more details follow)
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information



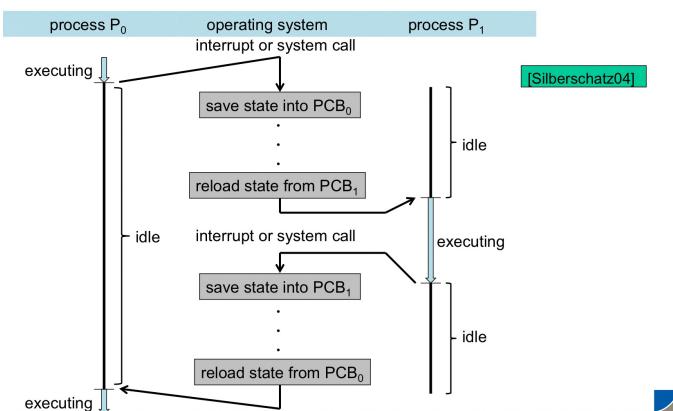
### Context switch

Context switch beschreibt das Wechseln des Prozessors von einem Prozess zum Anderen.

Der Kontext wird durch den jeweiligen PCB dargestellt. Beim Wechsel muss der Zustand des alten Prozesses gespeichert werden und der neue geladen werden.

Context switch wird durch das OS durchgeführt:

- System call** – process calls operating system procedure  
**Interrupt handler** – device controller signals interrupt



### Process state

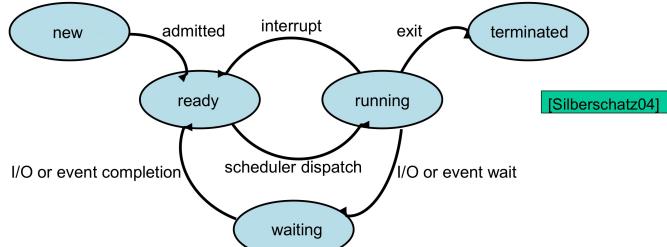
**Status:** new: The process is being created

running: Instructions are being executed

waiting: The process is waiting for some event to occur

ready: The process is waiting to be assigned to a processor

terminated: The process has finished execution

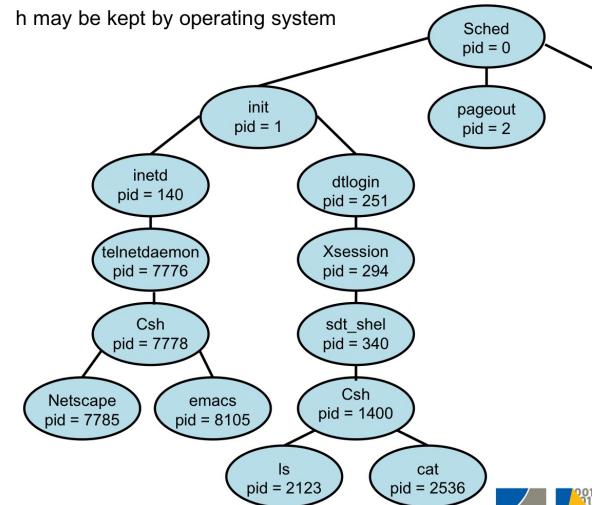


# Implementation in Unix

## Prozess Hierarchy

Prozesse werden mit einer ID identifiziert (**pid**)

Jeder Prozess wird von einem parent-Prozess gestartet  
→ Hierarchie wird beibehalten



## Process Creation

### Optionen:

#### Resource sharing

- Parent and children share all resources
- Children share subset of parent's resources
- Parent and child share no resources

#### Execution

- Parent and children execute concurrently
- Parent waits until children terminate

#### Address space

- Child duplicate of parent
- Child has a program loaded into it

## system()-Funktion

Erstellt einen Sub-Prozess, der die Standard-shell ausführt, die einen Befehl ausführt

```
#include <stdlib.h>

int main()
{
    int return_value;
    return_value = system("ls -1 /");
    return return_value;
}
```

## fork()-Funktion

Erstellt einen child-Prozess, der eine Kopie des parent-Prozess darstellt.

Return values:

parent-Prozess: child-Prozess pid  
child-Prozess: Null

```
int main()
{
    pid_t pid;

    /* fork another process */
    pid = fork(); ←
    if (pid < 0) { /* error occurred*/
        fprintf(stderr, "Fork Failed");
        exit(-1);
    }
    else if (pid == 0) { /* child process */
        execl("/bin/ls", "ls", NULL); ←
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
        exit(0);
    }
}
```

## exec()-Funktion

Führt ein Sub-Programm aus, was den aktuellen Prozess ersetzt

List of arguments specified as null-terminated strings, last argument must be null pointer

- int execl(const char \*path, const char \*arg, ...);
- int execlp(const char \*file, const char \*arg, ...);
- int execle(const char \*path, const char \*arg, ..., char \* const envp[]);

List of arguments specified as array of null-terminated strings

- int execv(const char \*path, char \*const argv[]);
- int execvp(const char \*file, char \*const argv[]);

## Process state

Prozess ID:

parent-Prozess ID:

```
pid_t getpid(void);
pid_t getppid(void);
```

## Process termination

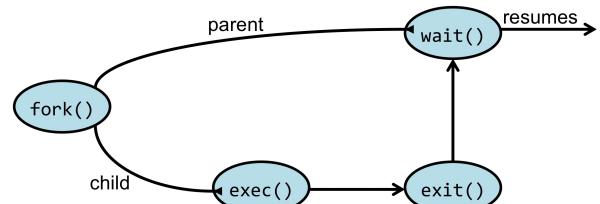
exit() - Funktion

Signalisiert normale Beendigung mit Rückgabewert der success oder failure angibt.

abort() - Funktion

Signalisiert nicht normale Beendigung.  
Prozess wurde vorzeitig beendet oder durch den parent-Prozess beendet.

```
int main()
{
    int child_status;
    /* The argument list to pass to the "ls" command. */
    char *arg_list[] = {
        "ls",
        /* argv[0], the name of the program. */
        "-l",
        "/",
        NULL
    };
    /* Spawn a child process running the "ls" command. Ignore the returned child process ID. */
    spawn("ls", arg_list);
    /* Wait for the child process to complete. */
    wait(&child_status);
    if (WIFEXITED(child_status))
        printf("The child process exited normally, with exit code %d\n",
               WEXITSTATUS(child_status));
    else
        printf("The child process exited abnormally\n");
    return 0;
}
```



## Warten auf process termination

wait() - Funktion

Wartet auf die Beendigung des Child-Prozesses

WIFEXITED macro: Bestimmen des Child-Prozess exit-code

WEXITSTATUS macro: Extrahieren des exit-codes

Asynchron:

periodically call `wait3()` or `wait4()` with non-blocking mode

Another approach is to use the signal `SIGCHLD`

- Parent process is notified when a child process terminates
- Parent specifies a signal handler, which stores the child process's termination status, and then cleans up the child

# Signale

Mit Signalen können Prozesse untereinander kommunizieren oder manipuliert werden. Erhält ein Prozess ein Signal, wird die Ausführung gestoppt und das Signal bearbeitet

## Signal types

Signal	Value	Action	Comment
SIGHUP	1	Term	Death of controlling process
SIGINT	2	Term	Interrupt from Keyboard
SIGQUIT	3	Core	Quit from Keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm
SIGTERM	15	Term	Termination signal
SIGUSR1	30, 10, 16	Term	User-defined signal 1
SIGUSR2	31, 12, 17	Term	User-defined signal 2
SIGCHLD	20, 17, 18	Ign	Child stopped or terminated
SIGCONT	19, 18, 25	Cont	Continue if stopped
SIGSTOP	17, 19, 23	Stop	Stop process
SIGTSTP	18, 20, 24	Stop	Stop typed at tty
SIGTTIN	21, 21, 26	Stop	tty input for background process
SIGTTOU	22, 22, 27	Stop	tty output for background process

## Signal handler

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

sig_atomic_t sigusr1_count = 0; ← (ATOMIC)
void handler(int signal_number)
{
    ++sigusr1_count;
}

int()
{
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = &handler; ←
    sigaction(SIGUSR1, &sa, NULL);

    /* Do some lengthy stuff here. */
    /* ... */

    printf("SIGUSR1 was raised %d times\n", sigusr1_count);
    return 0;
}
```

## Zombie Prozesse

Entsteht, wenn der parent-Prozess nicht wait() aufruft. So wird der child-Prozess nicht richtig beendet und nicht hinter ihm aufgeräumt. Erst, wenn der parent-prozess terminiert, wird der child-Prozess beendet.

## Process tools in Linux

**ps**: Zeigt alle aktuellen Prozesse

**pstree**: zeigt alle aktuellen Prozesse in einer Baum-Struktur

**top**: zeigt alle Prozesse (kann interaktiv angeordnet werden)

**htop**: erweitertes top

**kill**: Signal zu einem Prozess senden

**nice**: Ändere scheduling priority vor dem Start

**renice**: Ändere scheduling priority während der Laufzeit eines Prozesses

**&**: Starte Prozess im Hintergrund

**bg**: Schiebe Prozess in den Hintergrund

**fg**: Schiebe Prozess in den Vordergrund

## 2.2 Threads

### Definition: Threads

Threads erlauben mehrere parallele Abläufe innerhalb eines Prozesses. Dabei teilen sich Threads Adressräume, Speicher usw., weshalb auf Konsistenz geachtet werden muss. Im Gegensatz zu Prozessen muss seltener der Kontext gewechselt werden und die Kommunikation zwischen Threads ist deutlich einfacher.

### Thread Typen

#### Kernel Threads

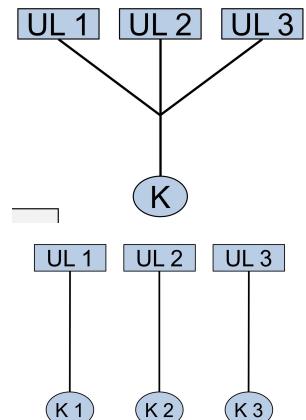
Ist Teil des Betriebssystems und Threads werden vom Kernel erstellt. Dadurch können die Vorteile von Multicore-Processing genutzt werden.

#### User Thread

Wird auf user-level durch eine Library implementiert, wodurch der Kernel von außen nur einen Prozess sieht. Diese sind zwar einfach zu erstellen, können aber keine Multiprocessors nutzen

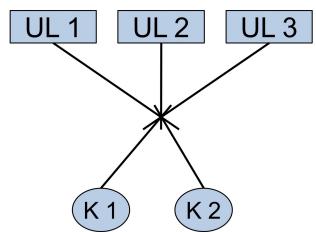
##### Many-to-One

Viele User threads werden von einem Kernel Thread ausgeführt.



##### One-to-One

Jeder User Thread ist genau einem Kernel Thread zugewiesen.



##### Many-to-Many

Mehrere User Threads sind mehreren Kernel Threads zugewiesen, wobei es weniger Kernel Threads als User Threads gibt.  
→ flexibelstes Modell

### Implementation von Threads in Linux

Linux unterscheidet nicht groß zwischen Threads und Prozessen (One-to-One)

#### clone()-Funktion

Erstellt einen neuen Thread als child-Prozess, der den Adressraum usw. mit dem Parent-Prozess teilt. Funktion wird von User Thread Implementationen verwendet.

# POSIX Threads

**POSIX:** "Portable Operating System Interface"

Library, die das standard Interface bietet zum arbeiten mit Threads. Im Folgenden wird ausschließlich mit dieser Library und pthreads gearbeitet.

**Import:** <pthread.h>

## Thread Creation

**Funktion:** int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*start\_routine)(void\*), void \*arg);

**Argumente:**  
**thread** – Data structure representing the created thread (= thread identifier)  
**attr** – The attributes of the new thread. If set to NULL, then the default attributes will be used.  
**start\_routine** – Pointer to the thread function executed by the newly created thread  
**arg** – Arguments passed to the start routine

**Rückgabe:** 0 if OK, error code otherwise

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

pthread_t workThreadID;

void *workThreadFunction(void *thread_arg) {
    printf("Your work thread says: Hello world!\n");
    return ((void *) 0);
}

int main(void) {
    pthread_create(&workThreadID, NULL, workThreadFunction, NULL);
    printf("Your main thread says: Hello world!\n");
    sleep(2);
    return 0;
}
```

## Thread Function

Funktion, die von einem neuen Thread gestartet wird. Die Rückgabe dieser Funktion terminiert den Thread.

**Allgemeine Signatur:** void\* **threadFunctionName** (void\*)

→ Um mehrere Argumente zu übergeben kann ein Structure mit den Parametern definiert werden

## Thread Attributes

Mit Attribute kann das grundlegende Verhalten der Threads angepasst werden.

**Initialisierung:** int pthread\_attr\_init(pthread\_attr\_t \*attr);

**Zerstörung:** int pthread\_attr\_destroy(pthread\_attr\_t \*attr);

**Zugriff:** int pthread\_attr\_setAttrName ( pthread\_attr\_t \*attr, AttrType t );  
int pthread\_attr\_getAttrName ( pthread\_attr\_t \*attr, AttrType \*t );

**Detach state:** int pthread\_attr\_setdetachstate (pthread\_attr\_t \*attr, int detachstate);  
int pthread\_attr\_getdetachstate(const pthread\_attr\_t \*attr, int \*detachstate);  
detachstate – either PTHREAD\_CREATE\_DETACHED or PTHREAD\_CREATE\_JOINABLE

**Detach a running Thread:** int pthread\_detach(pthread\_t thread);

**Thread Scheduling:** pthread\_attr\_setschedpolicy / pthread\_attr\_getschedpolicy  
pthread\_attr\_setscope / pthread\_attr\_getscope  
pthread\_attr\_setinheritsched / pthread\_attr\_getinheritsched

**Stack Information:** pthread\_attr\_setstacksize / pthread\_attr\_getstacksize  
pthread\_attr\_setstackaddr / pthread\_attr\_getstackaddr

```
int main(void) {  
    pthread_attr_t wtAttr;  
  
    pthread_attr_init(&wtAttr);  
    pthread_attr_setdetachstate(&wtAttr, PTHREAD_CREATE_DETACHED);  
    pthread_create(&workThreadID, &wtAttr, workThreadFunction, NULL);  
    pthread_attr_destroy(&wtAttr);  
  
    [...]  
}
```

## Joining Threads

Funktion blockiert einen Thread bis ein anderer Thread terminiert. Funktioniert nur bei joinable Threads. Detached Threads können nicht mehr joinable gemacht werden.

**Funktion:** int pthread\_join(pthread\_t thread, void \*\*value\_ptr);

**Argumente:** thread – Identifier of the target thread  
value\_ptr – The result value passed to pthread\_exit by the terminating thread

```
void* workThreadFunction(void* thread_arg) {  
    int answer = 42;  
    return ((void*) answer);  
}  
  
int main(void) {  
    int threadResult;  
    pthread_create(&workThreadID, NULL, workThreadFunction, NULL);  
    pthread_join(workThreadID, (void*) &threadResult);  
    printf("Thread result: %d\n", threadResult);  
}
```

return value is integer,  
but casted to pointer type!

result available in  
integer variable

## Thread Ending

1. Implizit: Return von der Thread Funktion
2. Explizit: void pthread\_exit(void \*value\_ptr);  
value\_ptr - The return value of the thread
3. Thread wird von einem anderen Thread im gleichen Prozess  
gecancelt: int pthread\_cancel(pthread\_t thread);

**Cancelability:** Jeder Thread bestimmt selber wie und wann er cancelt werden kann.

int pthread\_setcancelstate(int state, int \*oldstate);

Enable cancelling by setting state to PTHREAD\_CANCEL\_ENABLE

Disable cancelling by setting state to PTHREAD\_CANCEL\_DISABLE

**Cancel Typen:** int pthread\_setcanceltype(int type, int \*oldtype);

**Deferred (synchronous):** setting type to PTHREAD\_CANCEL\_DEFERRED

Thread kann nur an vom Programmierer bestimmten Cancellation points gecancelt werden:

void pthread\_testcancel(void);

**Asynchronous Cancelable:** setting type to PTHREAD\_CANCEL\_ASYNCHRONOUS

Thread kann an jedem Punkt gecancelt werden

```
void* workThreadFunction(void* thread_arg) {
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
    // Do fancy stuff...
    pthread_testcancel();
    // Do even more fancy stuff...
}

int main(void) {
    pthread_create(&workThreadID, NULL, workThreadFunction, NULL);
    pthread_cancel(workThreadID);
    pthread_join(workThreadID, NULL);
}
```

## Thread Synchronization

Threads arbeiten asynchron zueinander, aber greifen auf die selben Daten zu im undefinierter Reihenfolge, was zu Fehlern führen kann.

### Definition: Critical Section

Teil des Codes, der zwischen Threads geteilte Ressourcen nutzt, die nur von einem Thread zu jeder Zeit genutzt werden sollen.

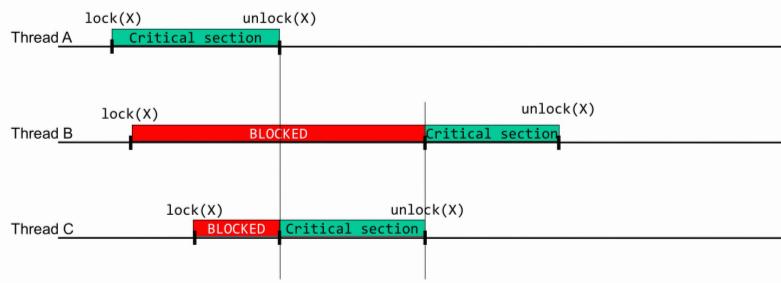
### Tanenbaum's Bedingungen für eine gute Lösung:

1. No two threads may be inside their critical section simultaneously.
2. No assumptions are to be made about speeds or the number of CPUs.
3. No thread running outside its critical region may block other threads.
4. No thread should have to wait forever to enter its critical region.

## Mutex Variables

**Prinzip:** Critical Sections sind durch Locks geschützt, die von Threads vor dem Eintreten in die Critical Section gesetzt werden.

Ist bereits ein Lock gesetzt, wird der Thread blockiert bis der andere Thread den Lock freigibt, wenn er die Critical Section verlässt.  
→ Bei den wartenden Threads ist kein FIFO garantiert



## Mutexes

**Data structure:** pthread\_mutex\_t – Mutex variable data type

**Initialisierung:**

**Dynamisch:** int pthread\_mutex\_init (pthread\_mutex\_t \*mutex, pthread\_mutexattr\_t \*attr);

**Statisch:** pthread\_mutex\_t mutexVar = PTHREAD\_MUTEX\_INITIALIZER;

**Zerstörung:** int pthread\_mutex\_destroy (pthread\_mutex\_t \*mutex);

**Mutex locken:**

int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);

→ Blockiert, falls Mutex bereits gelockt ist

int pthread\_mutex\_trylock(pthread\_mutex\_t \*mutex);

→ gibt an, ob das Mutex gelockt werden konnte

0 – Mutex has been locked successfully.

EBUSY – Mutex has not been locked, because it was already locked.

**Mutex unlocken:** int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);

## Mutex Attributes

**Data structure:** pthread\_mutexattr\_t – Mutex attributes data type

**Initialisierung:** int pthread\_mutexattr\_init(pthread\_mutexattr\_t \*attr);

**Zerstörung:** int pthread\_mutexattr\_destroy(pthread\_mutexattr\_t \*attr);

## Mutex Type setzen:

```
int pthread_mutexattr_settype (pthread_mutexattr_t *attr, int type);
int pthread_mutexattr_gettype (const pthread_mutexattr_t *attr, int *type);
```

## Mutex Types

PTHREAD\_MUTEX\_NORMAL: Wird ein Mutex zweimal vom gleichen Thread versucht zu locken, geht es in ein Deadlock

PTHREAD\_MUTEX\_ERRORCHECK: Relocking eines locked Mutexes oder unlocking eines unlocked Mutex gibt einen Fehler

PTHREAD\_MUTEX\_RECURSIVE: Mehrere Locks vom gleichen Thread sind möglich. Muss genauso oft unlockt werden.

PTHREAD\_MUTEX\_DEFAULT: Rekursives Locking und unlock errors

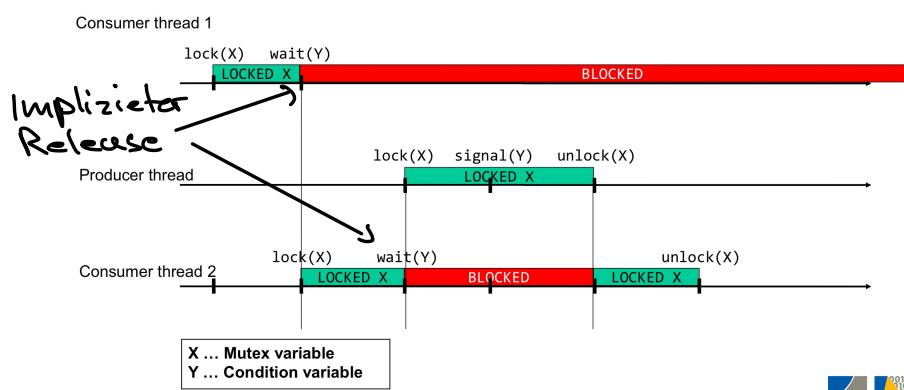
## Beispiel:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void.addValue(int value) {
    struct ListNode* newNode = (struct ListNode*) malloc(sizeof(struct ListNode));
    newNode->value = value;
    pthead_mutex_lock(&mutex);
    (*)
    newNode->succ = firstNode;
    (*)
    firstNode = newNode;
    pthead_mutex_unlock(&mutex);
}
```

## Condition Variables

**Prinzip:** Ein Thread soll auf eine condition Variable (Event) warten und blockiert werden bis ein anderer Thread die Condition Variable setzt

Vor dem Warten auf die Condition Variable wird ein Mutex gelockt. Der Lock wird implizit aufgehoben, wenn auf die Condition Variable gewartet wird. Mutex wird gelockt von dem signalisierenden Thread und nach dem Signal wieder freigesetzt. Da das Warten auf die Condition Variable vorbei ist, wird das Mutex wieder gelockt.



## Condition Variable

**Data structure:** pthread\_cond\_t – Condition variable data type

**Initialisierung:**

**Dynamisch:** int pthread\_cond\_init(pthread\_cond\_t \*cond, const pthread\_condattr\_t \*attr);

**Statisch:** pthread\_cond\_t condVar = PTHREAD\_COND\_INITIALIZER;

**Zerstörung:** int pthread\_cond\_destroy(pthread\_cond\_t \*cond);

## Condition Variable Attribute

**Data structure:** pthread\_condattr\_t – Condition attributes data type

**Initialisierung:** int pthread\_condattr\_init(pthread\_condattr\_t \*attr);

**Zerstörung:** int pthread\_condattr\_destroy(pthread\_condattr\_t \*attr);

→ normalerweise bleiben alle Attribute auf dem Default Value

## Warten auf Condition Variables

### 1. int pthread\_cond\_wait (pthread\_cond\_t \*cond, pthread\_mutex\_t \*mutex);

cond – Condition variable to wait on

mutex – Mutex variable associated with cond

Precondition: mutex must be locked by the calling thread

Effect: Atomically releases mutex and blocks the calling thread on the condition variable cond until cond "is signalled"

Postcondition: mutex is again locked by the calling thread

**Beispiel:**

```
pthread_mutex_lock(&mutex);
pthread_cond_wait(&cond, &mutex);
// Signal has been received
// Do actual work here...
pthread_mutex_unlock(&mutex);
// ... or here
```

### 2. int pthread\_cond\_timedwait (pthread\_cond\_t \*cond, pthread\_mutex\_t \*mutex, const struct timespec \*abstime);

abstime – Absolute time value limiting how long the calling thread can remain blocked waiting for the signal Returns ETIMEDOUT if abstime is passed without cond being signaled

**Beispiel:**

```
int result;
struct timespec tspec;
struct timeval tval;

pthread_mutex_lock(&mutex);
result = gettimeofday(&tval, NULL);
tspec.tv_sec = tval.tv_sec;
tspec.tv_nsec = tval.tv_usec * 1000;
tspec.tv_sec += 5;
result = pthread_cond_timedwait(&cond, &mutex, &tspec);
if (result == ETIMEDOUT) {
    printf("Timeout!\n");
} else {
    printf("Signal received!\n");
}
pthread_mutex_unlock(&mutex);
```

Nur ein Thread durch ein Signal wecken

→ so kann unter anderem sichergestellt werden, dass der richtige Thread gewechselt wird

```
pthread_mutex_lock(&threadFlagMutex);
while(!threadFlag) {
    pthread_cond_wait(&threadFlagCondition, &threadFlagMutex);
}
pthread_mutex_unlock(&threadFlagMutex);
// Condition fulfilled - Do work!
```

## Signalisieren

Mindestens einen Thread entblocken:

```
int pthread_cond_signal(pthread_cond_t *cond);
```

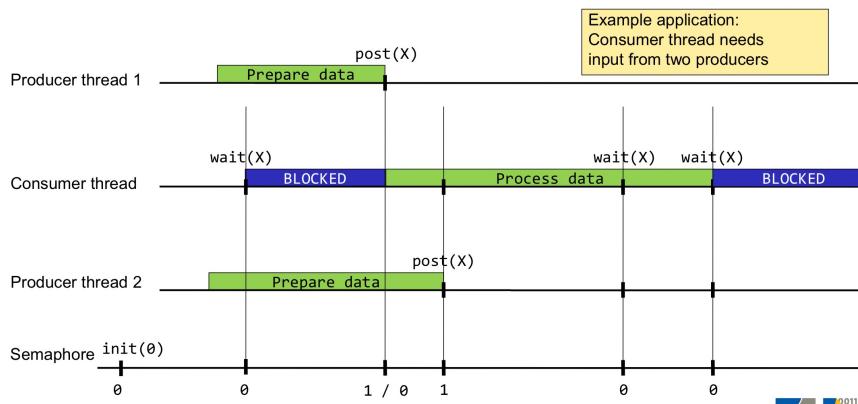
Alle Threads entblocken:

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

## Semaphoren

**Prinzip:** Die Anzahl der Threads die auf eine gemeinsame Ressource zugreifen soll durch einen globalen Counter begrenzt werden.

Ein Counter wird von einem Thread dekrementiert vor dem Aufruf der Critical section. Ist der Counter bereits bei 0, wird der Thread blockiert und wartet. Nach dem Aufruf wird der Counter wieder um eins inkrementiert.



## Semaphores

**Data structure:** `sem_t` – Semaphore data type

**Initialisierung:** `int sem_init(sem_t *sem, int pshared, unsigned int value);`

sem	– The semaphore
pshared	– Indicates whether the semaphore is shared among processes
value	– The initial value of the semaphore

**Zerstörung:** `int sem_destroy(sem_t *sem);`

### Warten:

1. `int sem_wait(sem_t *sem);`

→ Blockiert bis die Semaphore gelockt werden kann

2. `int sem_trywait(sem_t *sem);`

→ gibt an, ob die Semaphore gelockt werden konnte

- |        |   |
|--------|---|
| 0      | – Semaphore has been locked successfully.                       |
| EAGAIN | – Semaphore has not been locked, because it was already locked. |

**Freigeben:** `int sem_post(sem_t *sem);`

```
void* workThread(void* thread_arg) {
    sem_wait(&workThreadSem);
    // Critical section
    sem_post(&workThreadSem);
}

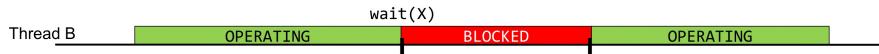
void init() {
    sem_init(&workThreadSem, 0, 1);
    [...]
}
```

→ implementiert Mutual Exclusion

## Barriers

**Prinzip:** Threads sollen aufeinander warten und erst fortfahren, sobald eine gewisse Anzahl an Threads die Barrier erreicht haben.

Dazu wird eine Nummer an Threads nBarr definiert und jeder Thread inkrementiert einen Counter bis nBarr erreicht wird und alle Threads fortfahren können.



If (counter < nBarr) then block calling thread  
init nBarr = 3      If (counter == nBarr) then unblock all threads and re-initialise counter with 0

## Pthread barriers

**Data structure:** pthread\_barrier\_t – Barrier data type  
pthread\_barrierattr\_t – Barrier attributes data type

**Initialisierung:** int pthread\_barrier\_init(pthread\_barrier\_t \*restrict barrier, const pthread\_barrierattr\_t \*restrict attr, unsigned count);  
count – The **number of threads** that must call pthread\_barrier\_wait for the threads to become unlocked

**Zerstörung:** int pthread\_barrier\_destroy(pthread\_barrier\_t \*barrier);

**Warten:** int pthread\_barrier\_wait(pthread\_barrier\_t \*);

```
void* workThread(void* thread_arg) {
    // Do a lot of work here
    pthread_barrier_wait(&allThreadsFinishedBarrier);
    printf("All threads finished!\n");
}

void init() {
    int i;
    pthread_barrier_init(&allThreadsFinishedBarrier, NULL, 5)
    for (i=0; i<5; i++) {
        pthread_create(&threadID[i], NULL, workThreadFunction, NULL);
    }
    [...]
}
```

thread(s) waiting here

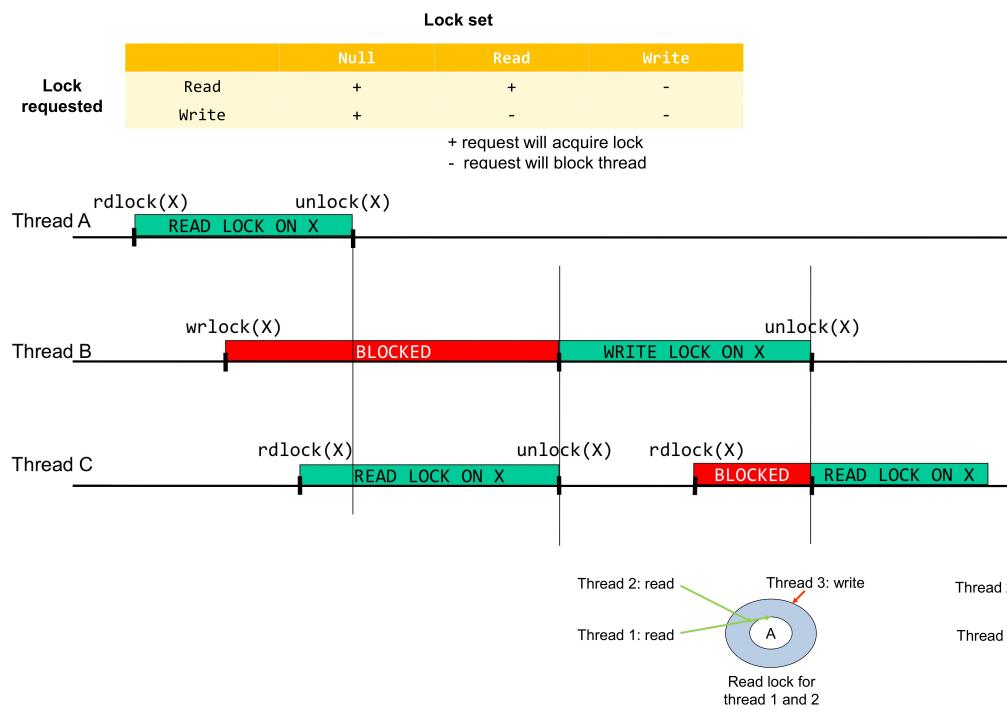
init with 5

## Read-Write Locks

**Prinzip:** Mehrere Threads können ohne Gefahr mit Read-Zugriffen auf die selben Daten zugreifen. Nur Write-Zugriffe können problematisch sein.

Read-Zugriffe sollen immer möglich sein, wenn andere Threads auch nur Read-Zugriffe auf die Daten haben und keiner einen Write-Zugriff.

Write-Zugriffe hingegen werden nur zugelassen, wenn kein anderer Thread auf die Daten zugreift



## Pthread Read-Write Locks

**Data structures:** pthread\_rwlock\_t  
pthread\_rwlockattr\_t

- Read-write lock data type
- Read-write lock attributes data type

**Initialisierung:**

**Dynamisch:** int pthread\_rwlock\_init(pthread\_rwlock\_t \*rwlock, const pthread\_rwlockattr\_t \*attr);

**Statisch:** pthread\_rwlock\_t rwlock = PTHREAD\_RWLOCK\_INITIALIZER;

**Zerstörung:** int pthread\_rwlock\_destroy(pthread\_rwlock\_t \*rwlock);

**Acquiring a read lock**

```
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
```

**Acquiring a write lock**

```
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
```

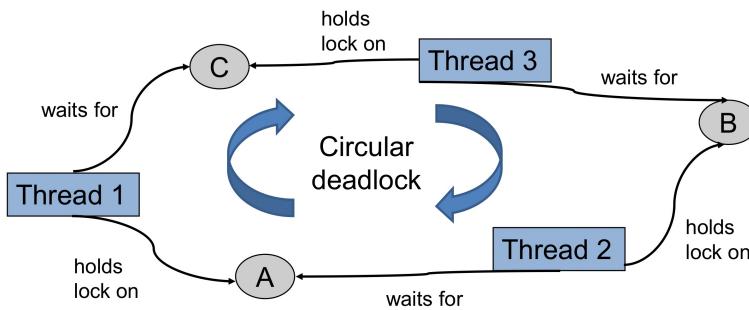
**Releasing a lock**

```
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

# Deadlocks

## Definition: Deadlock

Ein Deadlock tritt auf, wenn zwei oder mehr Threads sich gegenseitig warten und so gegenseitig ihr Fortlaufen blockieren.



## Deadlock Bedingungen

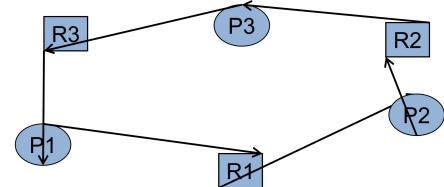
Alle Bedingungen müssen für ein Deadlock erfüllt sein.

1. **Mutual Exclusion:** Ressourcen können nur von einem Thread zu jeder Zeit genutzt werden.
2. **Non preemption:** Ressourcen können nur vom haltenden Thread freigegeben werden und ihm nicht entzogen werden.
3. **Hold & Wait:** Threads halten die Ressourcen auch während sie warten
4. **Circular Wait:** Es gibt einen geschlossenen Kreis von Threads, die jeweils auf Ressourcen warten, die vom Vorgänger gehalten werden.

## Resource-Allocation Graphs

**Knoten:** 1. Prozesse / Threads als Kreise  
2. Ressourcen als Rechtecke

**Kanten:** Process → Resource: Process is waiting for resource  
Resource → Process: Resource is held by process



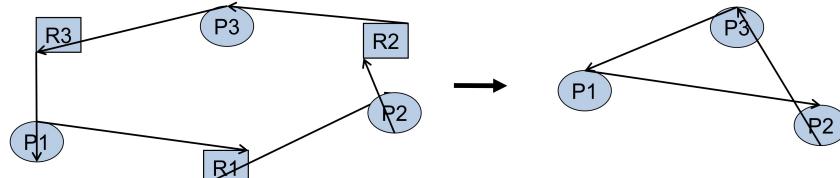
→ Cycle ⇒ Deadlock

## Process Wait-For Graphs

**Knoten:** Prozesse

**Kanten:** P1 → P2: Process P1 waits for Process P2

**Conversion:**



## Handling Deadlocks

### Deadlock Prevention

Eine der Bedingungen soll nie erfüllt sein

1. Threads dürfen immer nur eine Ressource auf einmal halten. (Bed. 3)
  2. Threads werden gezwungen alle Ressourcen freizugeben, wenn sie eine Ressource anfordern, die nicht direkt geliefert werden kann. (Bed. 2)
  3. Reihenfolge angeben in welcher die Ressourcen angefragt werden können (Bed. 4)
- Thread Synchronisation kann die ersten drei Bedingungen verhindern und sollte immer angewendet werden.

### Deadlock avoidance

Übergeordnetes System (OS) braucht Information darüber welcher Thread welche Ressourcen benötigt und kann so entscheiden welche Zugriffe gemacht werden können

### Deadlock detection

Periodische Überprüfung des Systems um Deadlocks zu erkennen und diese zu beheben.

### Suspecting deadlocks

Ahnlich wie Deadlock detection, nur dass Deadlocks erkannt werden sollen bevor diese auftreten.

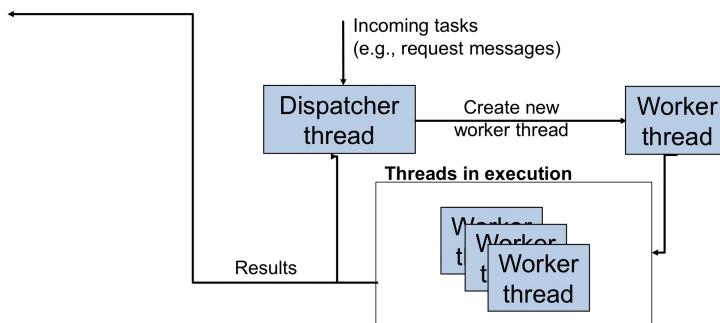
# Threading Mechanismen

## Worker Threads

Komplexere Aufgaben werden an Worker Threads im Hintergrund von einem Dispatcher Thread delegiert, sodass der Dispatcher Thread im Vordergrund nicht blockiert.

Die Worker Threads terminieren nach Abschluss der Aufgabe und geben ihr Ergebnis an den Dispatcher Thread.

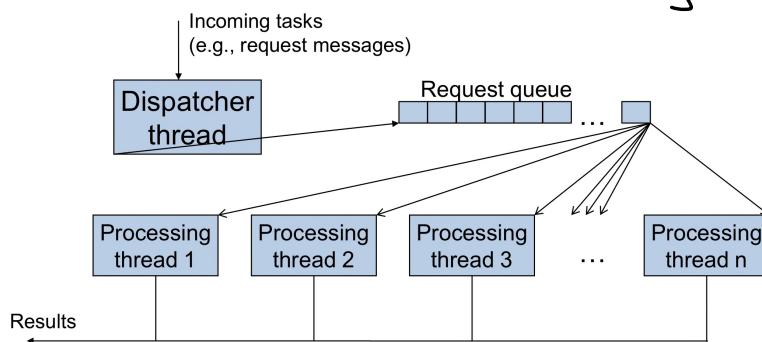
→ Parallelismus benötigt Thread Synchronisation



## Thread Pools

Threads werden beim Start erstellt um wiederkehrende (kurze) Aufgaben zu erledigen. So müssen nicht immer wieder aufwändig neue Threads erstellt werden, sondern Aufgaben können direkt an Threads delegiert werden.

→ effizientes Multi threading



## Thread - Safe

Code/libraries sind thread safe, wenn sie parallel in mehreren Threads korrekt funktionieren

### Ensure re-entrance of functions

- Only use **purely local state** – no access of shared data
- Partial execution of the function only affects the stack of the thread

### Use mutual exclusion

- Serialize access to critical sections using thread synchronization primitives

### Use thread-local data

- Create copies of data for each thread and operate only on this thread-local data

### Use atomic operations for accessing shared data

### Share memory by communication

## 2.3 Interprocess Communication IPC

### Definition: Persistence

Persistence definiert wie lange ein IPC Objekt existiert.

#### 1. Process persistence

IPC-Objekte bleiben so lange bestehen, wie die zugreifenden Prozesse ihre Verbindung halten.

#### 2. Kernel persistence

IPC-Objekte werden beim kernel-reboot gelöscht oder explizit gelöscht.

#### 3. Filesystem persistence

IPC-Objekte bleiben bestehen bis sie explizit gelöscht werden.

Type of IPC	Persistence
Pipe	process
FIFO (named pipes)	process
Posix mutex	process
Posix condition variable	process
Posix read-write lock	process
fcntl record locking	process
Posix message queue	kernel
Posix named semaphore	kernel
Posix memory-based semaphore	process
Posix shared memory	kernel
System V message queue	kernel
System V semaphore	kernel
System V shared memory	kernel
TCP socket	process
UDP socket	process
Unix domain socket	process

→ IPC sollen i.A. keinen Reboot überstehen

## Filehandle zwischen Prozessen

#### 1. Child-Prozesse erhalten Kopien von den Filehandles des Parent-Prozessen (fort.)

→ Interaktion zwischen Child und Parent

#### 2. exec: Handles bleiben offen

# Kommunikation mit Files

Prozesse können über Files kommunizieren durch Benutzung der Files API. Diese können gleichzeitig oder zeitlich versetzt zugreifen

## File API

Alle Funktionen arbeiten auf Files referenziert über einen File Descriptor.

### Open/Create Files

```
int open(const char *pathname, int oflag, ... /* mode_t mode */ );
```

**oflag:** Optionen zum Öffnen und Erstellen

- O\_RDONLY, O\_WRONLY, O\_RDWR: read only, write only, read/write access:  
exactly one of these must be specified
- O\_APPEND: write appends to end of file
- O\_TRUNC: truncate file to length 0
- O\_CREAT: create file if not existing; if specified, third parameter
- mode: access permissions, cf. next slide
- O\_EXCL: error if O\_CREAT is specified and file already exists

```
int creat(const char *path, mode_t mode);
```

→ Äquivalent zu open

**mode:** S\_IRUSR, S\_IWUSR, S\_IXUSR: read, write, execute for user

S\_IRGRP, S\_IWGRP, S\_IXGRP: read, write, execute for group

S\_IROTH, S\_IWOTH, S\_IXOTH: read, write, execute for other

### Lesen einer File

```
ssize_t read(int filedes, void *buf, size_t nbytes);
```

**Return:**

0: end of file

-1: error

### Schreiben einer File

```
ssize_t write(int filedes, const void *buf, size_t nbytes);
```

### Schließen einer File

```
int close(int filedes);
```

### Entfernen des Filenames

```
int unlink(const char *path);
```

→ Keine opens mehr möglich

### Filepointer verschieben

```
off_t lseek(int filedes, off_t offset, int whence);
```

→ Bewegt Filepointer für read/write Operationen

**whence:** Position zu der relativ verschoben werden soll

SEEK\_SET: beginning of file

SEEK\_CUR: current position

SEEK\_END: end of the file

**Return:** Absolute Position

```
int main (int argc, char* argv[])
{
    /* The file to which to append the timestamp. */
    char* filename = argv[1];
    /* Get the current timestamp. */
    char* timestamp = get_timestamp ();
    /* Open the file for writing. If it exists, append to it;
     * otherwise, create a new file. */
    int fd = open (filename, O_WRONLY | O_CREAT | O_APPEND, 0666);
    /* Compute the length of the timestamp string. */
    size_t length = strlen (timestamp);
    /* Write the timestamp to the file. */
    write (fd, timestamp, length);
    /* All done. */
    close (fd);
    return 0;
}
```

(Source: Advanced Linux Programming by CodeSourcery LLC, published by New Riders)

File Permissions: [https://en.wikipedia.org/wiki/File\\_system\\_permissions](https://en.wikipedia.org/wiki/File_system_permissions)

```
% cat tsrl
Thu Feb 1 2
% ./tsmeta
% cat tsfil
Thu Feb 1 2
Thu Feb 1 2
```

## Portable (Buffered) File API

Gebufferte Streams, sollte vorzüglich genutzt werden

### Korrespondierende Funktionen:

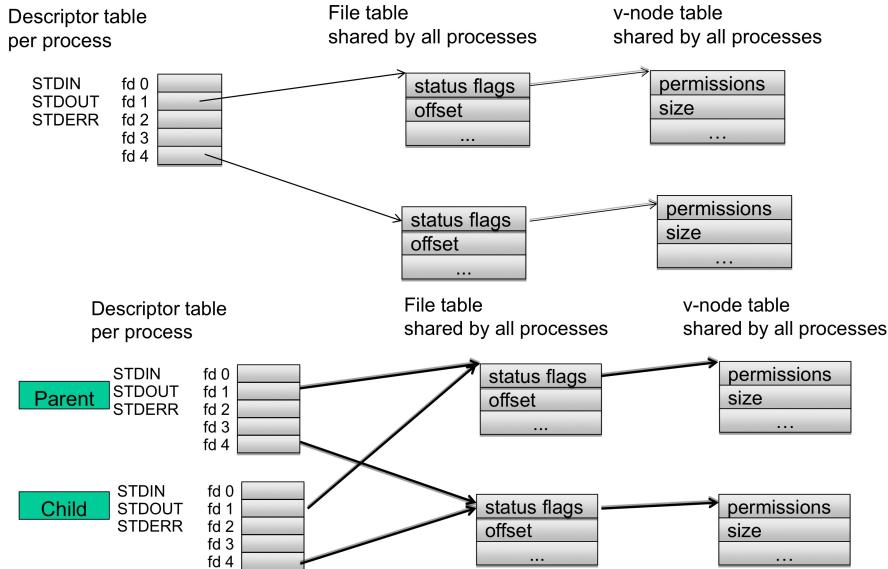
fopen, fread, fwrite, fseek corresponding calls

- Call open, read, write, lseek internally

```
int main (int argc, char* argv[])
{
    /* The file to which to append the timestamp. */
    char* filename = argv[1];
    /* Get the current timestamp. */
    char* timestamp = get_timestamp ();
    /* Open the file for writing. If it exists, append
     * otherwise, create a new file. */
    FILE fp = fopen (filename, "w+");
    /* Compute the length of the timestamp string. */
    size_t length = strlen (timestamp);
    /* Write the timestamp to the file. */
    fwrite (timestamp, sizeof(char), length, fp);
    /* All done. */
    fclose (fp);
    return 0;
}
```

## File Management Structures

Jeder Prozess hat eine Descriptor table mit allen offenen File Descriptors. Jede File besitzt einen eigenen table mit Informationen zum File Descriptor und jede File einen table mit Metadaten.



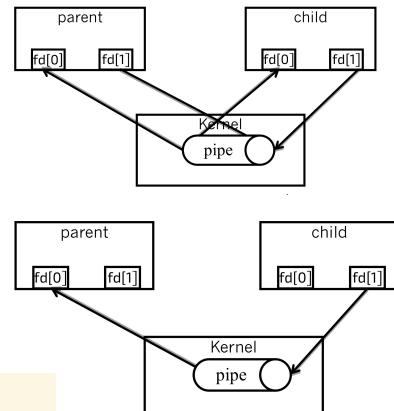
→ File Offset wird geteilt

# Pipes

**Prinzip:** Zwei verbundene File Descriptors von Prozessen mit gleichen Vorgängen. Dies funktioniert nur Half-Duplex, Information fließt nur in eine Richtung

## Pipe Creation

```
int pipe(int fd[2]);  
fd[0]: open for reading  
fd[1]: open for writing
```



1. Erstelle Pipe
2. Forke einen Child-Prozess
3. Child- und Parent-Prozess schließen jeweils einen File Descriptor

## Pipe read/write

### read/write from File API

All writers close pipe

- read returns 0 (=EOF)
- > Reader process must close write file descriptor

All readers close pipe

- Signal SIGPIPE is generated on write

```
int main(void) {  
    int n;  
    int fd[2];  
    pid_t pid;  
    char line[MAXLINE];  
  
    if (pipe(fd) < 0)  
        err_sys("pipe error");  
    if ((pid = fork()) < 0) {  
        err_sys("fork error");  
    } else if (pid > 0) { /* parent */  
        close(fd[0]);  
        write(fd[1], "hello world\n", 12);  
        close(fd[1]);  
    } else { /* child */  
        close(fd[1]);  
        n = read(fd[0], line, MAXLINE);  
        write(STDOUT_FILENO, line, n);  
    }  
    exit(0);  
}
```

## popen und pclose

### Kombiniert pipe + fork + system

```
FILE *popen(const char *cmdstring, const char *type);
```

→ Cmdstring wird als neuer Prozess ausgeführt und der input bzw. output wird umgeleitet

- type:**
- "r": STDOUT of child is redirected; parent can read
  - "w": STDIN of child is redirected; parent can write

```
int pclose(FILE *fp);  
→ schließt pipe
```

## Duplicate existing file descriptors

```
int dup(int filedes);  
→ Dupliziert einen file descriptor und gibt den Neuen aus  
int dup2(int filedes, int filedes2);  
→ Schließt filedes2 und kopiert filedes nach filedes2
```

## FIFOs (Named Pipes)

**Prinzip:** Pipes mit Namen, die im File System erstellt werden, die von beliebigen Prozessen in Half-Duplex

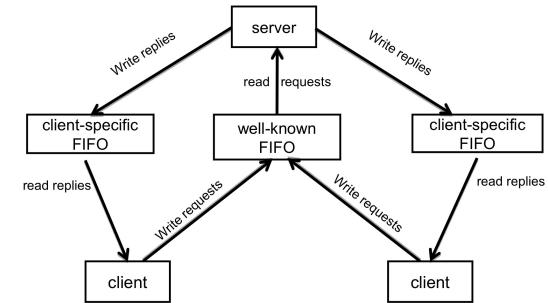
### FIFO Creation

int **mkfifo**(const char \*pathname, mode\_t mode);

**Mode:**

- O\_RDONLY, O\_WRONLY, O\_RDWR: read only, write only, read/write access; exactly one of these must be specified
- O\_APPEND: write appends to end of file
- O\_TRUNC: truncate file to length 0
- O\_CREAT: create file if not existing; if specified, third parameter
- mode: access permissions, cf. next slide
- O\_EXCL: error if O\_CREAT is specified and file already exists

→ Implies O\_CREAT | O\_EXCL



### FIFO Open

Open for either O\_RDONLY or O\_WRONLY  
→ blockiert bis beide Seiten geöffnet sind

### FIFO Reading / Writing

read/write operations for pipe

### FIFO Closing / Deleting

**Close:** wie bei Files  
→ File bleibt bestehen, Inhalt wird gelöscht

**Deletion:** int **unlink**(const char \*path);

### Non-Blocking Operations

: use flag O\_NONBLOCK for open

```
/* Set file descriptor to non-blocking */
int flags;
/* Get current flags */
if ((flags = fcntl(filedes, F_GETFL, 0)) < 0) {
    /* Handle error */
}

/* set non-blocking flag */
flags |= O_NONBLOCK; ←

if (fcntl(filedes, F_SETFL, flags) < 0) {
    /* Handle error */
}
```

```
/* Set file descriptor to blocking */
int flags;
/* Get current flags */
if ((flags = fcntl(filedes, F_GETFL, 0)) < 0) {
    /* Handle error */
}

/* clear non-blocking flag */
flags &= ~O_NONBLOCK; ←

if (fcntl(filedes, F_SETFL, flags) < 0) {
    /* Handle error */
}
```

Operation	Existing opens of pipe or FIFO	Blocking	Non-Blocking
Open FIFO for reading	Open for writing	Returns OK	Returns OK
	Not open for writing	Blocks until opened for writing	Returns OK
Open FIFO for writing	Open for reading	Returns OK	Returns OK
	Not open for reading	Blocks until opened for reading	Returns ENXIO
Read empty pipe or FIFO	Open for writing	Blocks until data available or all writers close	Returns EAGAIN
	Not open for writing	Returns 0 (EOF)	Returns 0 (EOF)
Write to pipe or FIFO	Open for reading	Returns OK	Returns OK
	FIFO/pipe full	Blocks until space available	Returns EAGAIN
	Not open for reading	Generate SIGPIPE	Returns EPIPE

# Posix Message Queues

#include <mqueue.h>

**Prinzip:** Message Queues nutzen linked list von Messages.  
So können Prozesse indirekt miteinander kommunizieren und müssen nicht gleichzeitig miteinander kommunizieren und ist daher mehr record orientated

Funktioniert über eigene API und File descriptors  
→ kernel persistence

**Open Message Queues:** mqd\_t mq\_open(const char \*name, int oflag);

**Create Message Queues:** mqd\_t mq\_open(const char \*name, int oflag, mode\_t mode, struct mq\_attr \*attr);

→ oflag und mode ist genauso wie bei Files

## Attribute

```
#include <mqueue.h>
mqd_t mq_getattr(mqd_t mqdes, struct mq_attr *attr);
mqd_t mq_setattr(mqd_t mqdes, struct mq_attr *newattr, struct mq_attr
*oldattr);

struct mq_attr {
    long mq_flags;           /* Flags: 0 or O_NONBLOCK */
    long mq_maxmsg;          /* Max. # of messages on queue */
    long mq_msgsize;         /* Max. message size (bytes) */
    long mq_curmsgs;         /* # of messages currently in queue */
};
```

**Sending Messages:** int mq\_send(mqd\_t mqdes, const char \*msg\_ptr, size\_t msg\_len, unsigned int msg\_prio);

→ Message wird nach Messages mit gleicher Priorität eingefügt und vor Messages mit tieferer priority (msg\_prio)

**Receive Messages:** ssize\_t mq\_receive(mqd\_t mqdes, char \*msg\_ptr, size\_t msg\_len, unsigned int \*msg\_prio);

→ Receives the oldest of the highest priority messages and removes it from the queue

→ msg\_prio saves the priority of the received message

## Send:

```
#include "unipipc.h"

int main(int argc, char **argv)
{
    mqd_t mqd;
    void *ptr;
    size_t len;
    uint_t prio;

    if (argc != 4)
        err_quit("usage: mqsend <name> <#bytes> <priority>");

    len = atoi(argv[2]);
    prio = atoi(argv[3]);

    mqd = Mq_open(argv[1], O_WRONLY);
    ptr = Calloc(len, sizeof(char));
    Mq_send(mqd, ptr, len, prio);
    exit(0);
}
```

Error checks hide  
function names'  
(see [Stevens1999])  
  
e.g. Mq\_open(..)  
function that calls  
error checks

send „err  
(Calloc s

## Receive:

```
int main(int argc, char **argv) {
    int c;
    mqd_t mqd;
    ssize_t n;
    uint_t prio;
    void *buff;
    struct mq_attr attr;

    if (argc != 2) err_quit("usage: mqreceive <name>");

    mqd = Mq_open(argv[1], O_RDONLY);
    Mq_getattr(mqd, &attr); /* obtain maximum message size */

    buff = Malloc(attr.mq_msgsize);

    n = Mq_receive(mqd, buff, attr.mq_msgsize, &prio);
    printf("read %ld bytes, priority = %u\n", (long)n, prio);

    exit(0);
}
```

Error checks hidden in upper  
function names'  
(see [Stevens1999] ch. 1.6)

Example adapted from [Ste

**Close Message Queues:** int mq\_close(mqd\_t mqdes);

**Delete Message Queues:** int mq\_unlink(const char \*name);

## Notification with Message Queue

**Prinzip:** Ein Prozess registriert sich auf der Message Queue. Notification passiert, wenn eine Message an die Queue gesendet wird und so diese von empty zu non-empty springt. Die Message wird dann direkt vom blockierenden Prozess entfernt, die Registrierung aufgehoben und der Prozess arbeitet weiter.

**Notify:** int mq\_notify(mqd\_t mqdes, const struct sigevent \*notification);

→ if Notification == NULL: Prozess Registration wird entfernt

**Attribute:**

```
union sigval {  
    int sival_int; /* Data passed with notification */  
    void *sival_ptr; /* Integer value */  
};  
struct sigevent {  
    int sigev_notify; /* Notification method */  
    int sigev_signo; /* Notification signal */  
    union sigval sigev_value; /* Data passed with notification */  
    void (*sigev_notify_function) (union sigval); /* Function for thread  
                                         notification */  
    void *sigev_notify_attributes; /* Thread function attributes */  
};
```

**Notification Methoden:** SIGEV\_NONE:  
no notification occurs

SIGEV\_SIGNAL:

signal notification->sigev\_signo is sent

SIGEV\_THREAD:

new thread with function notification->sigev\_thread\_function  
is created

```
#define die(msg) { perror(msg); exit(EXIT_FAILURE); }  
static void tfunc(union sigval sv); /* forward declaration */  
  
int main(int argc, char *argv[]) {  
    mqd_t mqdes;  
    struct sigevent not;  
  
    (*) assert(argc == 2);  
  
    mqdes = mq_open(argv[1], O_RDONLY);  
    if (mqdes == (mqd_t)-1) die("mq_open");  
  
    not.sigev_notify = SIGEV_THREAD;  
    not.sigev_notify_function = tfunc;  
    not.sigev_notify_attributes = NULL;  
    not.sigev_value.sival_ptr = &mqdes; /* Arg. to thread func. */  
    if (mq_notify(mqdes, &not) == -1) die("mq_notify");  
  
    pause(); /* Process will be terminated by thread function */  
}  
  
static void tfunc(union sigval sv) { /* Thread start function */  
    struct mq_attr attr;  
    ssize_t nr;  
    void *buf;  
    mqd_t mqdes = *((mqd_t *) sv.sival_ptr);  
  
    /* Determine max. msg size; allocate buffer to receive msg */  
  
    if (mq_getattr(mqdes, &attr) == -1) die("mq_getattr");  
    buf = malloc(attr.mq_msgsize);  
    if (buf == NULL) die("malloc");  
  
    nr = mq_receive(mqdes, buf, attr.mq_msgsize, NULL);  
    if (nr == -1) die("mq_receive");  
  
    printf("Read %ld bytes from MQ", (long) nr);  
    free(buf);  
    exit(EXIT_SUCCESS); /* Terminate the process */  
}
```

- Register notification via command line argument (argv[1] is MQ name)
- Perform notification by creation of new thread

(\*) assert - abort the program if assertion is false

#include <assert.h>  
void assert(scalar expression);

- Output message
- Terminate process

## Sockets

**Prinzip:** Full-Duplex Kommunikation zwischen Sockets, die über Ports identifiziert werden.

### Unix Domain Sockets

Gleiche API wie bei Internet Domain Sockets

**Data Structure:**

```
struct sockaddr_un {  
    sa_family_t sun_family;  
    char sun_path[108];  
};
```

```
#include "apue.h"  
#include <sys/socket.h>  
#include <sys/un.h>  
  
int main(void) {  
    int fd, size;  
    struct sockaddr_un un;  
  
    un.sun_family = AF_UNIX;  
    strcpy(un.sun_path, "foo.socket");  
    if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) < 0)  
        err_sys("socket failed");  
    size = offsetof(struct sockaddr_un, sun_path) + strlen(un.sun_path);  
    if (bind(fd, (struct sockaddr *)&un, size) < 0)  
        err_sys("bind failed");  
    printf("UNIX domain socket bound\n");  
    exit(0);  
}
```

"apue.h"  
comes from <http://www.apuebook.com>  
Resources to Stevens et al. A  
Programming in the Unix Env

## Posix Shared Memory

Identische oder geteilter Speicher von Threads über den  
kommuniziert wird. Thread Synchronisation erforderlich.

**Möglichkeiten:**

- Posix shared memory
- Memory-mapped file
- /dev/zero anonymous memory-mapping
- Kernel persistence

# IPC Synchronisation: Posix Semaphores

Shared-memory based Semaphores: wird weiter oben gezeigt

## Named Semaphores

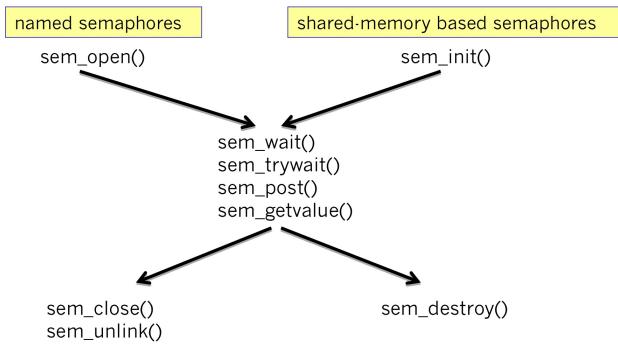
**Prinzip:** Können genutzt werden um Prozesse zu synchronisieren und so externe Daten, shared memory von Prozessen, zu schützen

**Open Named Semaphores:** `sem_t *sem_open(const char *name, int oflag);`

**Create Named Semaphores:** `sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);`

**Close Named Semaphores:** `int sem_close(sem_t *sem);`

**Removal Named Semaphores:** `int sem_unlink(const char *name);`



## Signals

**Prinzip:** Signale, spezifiziert durch ihre Nummer, können explizit genutzt werden um asynchron Notifications zwischen Prozessen zu senden oder Notifications vom System zu senden.

Wenn ein Prozess ein Signal erhält, stoppt die Ausführung und das Signal wird bearbeitet. Danach wird die Ausführung ggf. fortgeführt

**Unreliable Signals:** in älteren Unix-Versionen Signal Handler deregistrieren und ggf. reregistrieren sich nach einem Signal Call. In dem kurzen Zeitfenster können Signale verloren gehen.

## Terminology

Signal is *generated* (or *sent* to a process) when event occurs

Signal is *delivered* if associated action is taken

Signal is *pending* in between

Signal delivery can be *blocked*

- Only delivery, not generation can be blocked
- Signal is pending until unblocked or action changed to ignore
- Set of *blocked* signals is called *signal mask*

**Signal action** (also called disposition) can be

- *default action* (often process termination)
- *catch*, i.e., custom handler
- *ignore*

## Interrupted System Calls

System Calls können mit Fehler EINTR returnen, wenn ein Signal während der Ausführung erzeugt wird  
→ Manche Implementation starten automatisch neu

## Reentrant Functions

Funktion müssen reentrant sein, wenn die Ausführung nach dem Signal Handler fortgesetzt werden soll. Das heißt sie muss im Code korrekt fortgeführt werden müssen.

## Generation of Signals

```
int kill(pid_t pid, int signo);
```

```
int raise(int signo);
```

→ *Raise*: sende Signal zum gleichen Prozess

**Pid:** pid>0: send to process with ID==pid

pid==0: send to process group of current process

pid<0: send to process group abs(pid)

pid=1: send to all processes

## Multiple Signals:

Verhalten ist implementationsabhängig:

1. Signal is delivered multiple times (queuing)

2. Signal is delivered only once

- **Most implementations deliver only once**

→ Andere Signale können Signal Handler unterbrechen

## Signal Catching

Ein Signal Handler bearbeitet ein ankommendes Signal und blockiert weitere gleiche Signale. Eine Signal Mask kann auch weitere Signale blockieren, während der Ausführung des Signal Handlers

**Signal Handler:** void `handler(int signo, siginfo_t *info, void *context);`

**Info:** zusätzliche Information

```
struct siginfo {
    int     si_signo;      /* signal number */
    int     si_errno;      /* if nonzero, errno value from <errno.h> */
    int     si_code;       /* additional info (depends on signal) */
    pid_t   si_pid;       /* sending process ID */
    uid_t   si_uid;       /* sending process real user ID */
    void   *si_addr;      /* address that caused the fault */
    int     si_status;    /* exit value or signal number */
    long    si_band;      /* band number for SIGPOLL */
    /*possibly other fields also */
};
```

Signal	Code	Reason
SIGILL Illegal Instruction	ILL_ILLOPC	illegal opcode
	ILL_ILLOPN	illegal operand
	ILL_ILLADR	illegal addressing mode
	ILL_ILLTRP	illegal trap
	ILL_PRVOPC	privileged opcode
	ILL_PRVREG	privileged register
	ILL_COPROC	coprocessor error
	ILL_BADSTK	internal stack error
SIGFPE Floating Point Exception	FPE_INTDIV	integer divide by zero
	FPE_INTOVF	integer overflow
	FPE_FLTDIV	floating-point divide by zero
	FPE_FLTOVF	floating-point overflow
	FPE_FLTUND	floating-point underflow
	FPE_FLTRES	floating-point inexact result
	FPE_FLTINV	invalid floating-point operation
	FPE_FLTSUB	subscript out of range
SIGPOLL Asynchronous IO	POLL_IN	data can be read
	POLL_OUT	data can be written
	POLL_MSG	input message available
	POLL_ERR	I/O error
	POLL_PRI	high-priority message available
	POLL_HUP	device disconnected

## Signal Handler verändern

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

If act != NULL, action is changed

If oact != NULL, old action is copied

```
struct sigaction {
    void        (*sa_handler)(int); /* addr of signal handler, */
                                    /* or SIG_IGN, or SIG_DFL */
    sigset_t    sa_mask;          /* additional signals to block during handler */
    int         sa_flags;         /* signal options; examples discussed above*/
    /* alternate handler */
    void        (*sa_sigaction)(int, siginfo_t *, void *);
};
```

## Beispiel

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

int main (int argc, char *argv[])
{
    struct sigaction act;

    memset (&act, '\0', sizeof(act));

    /* Use the sa_sigaction field because the handles has two additional parameters */
    act.sa_sigaction = &hdl;

    /* The SA_SIGINFO flag tells sigaction() to use the sa_sigaction field, not sa_handler. */
    act.sa_flags = SA_SIGINFO;

    if (sigaction(SIGINT, &act, NULL) < 0) {
        perror ("sigaction");
        return 1;
    }

    while (1)
        sleep (10);

    return 0;
}

static void hdl (int sig, siginfo_t *siginfo, void *context)
{
    printf ("Sending PID: %ld, UID: %ld\n",
           (long)siginfo->si_pid, (long)siginfo->si_uid);
}
```

## Blocking Signals

```
int sigprocmask(int how, const sigset_t *set, sigset_t *oset);  
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oset);
```

→ Signal mask wird verändert

- How:
  - SIG\_BLOCK: additionally block specified signals
  - SIG\_SETMASK: replace old mask by new mask
  - SIG\_UNBLOCK: unblock specified signals

set == NULL → no modification, how is irrelevant

oset != NULL → old signal mask is copied

Only signals that can be ignored can also be blocked

- not SIGKILL and SIGSTOP

Blocking of SIGFPE, SIGILL, SIGSEGV or SIGBUS

- Programming errors; execution cannot be continued
- Behavior undefined unless explicitly generated by kill()

## Pending Signals

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

→ Behalte das Set von pending Signals

## Signal Sets

```
int sigemptyset(sigset_t *set);  
int sigfillset(sigset_t *set);  
int sigaddset(sigset_t *set, int signo);  
int sigdelset(sigset_t *set, int signo);  
▪ All return 0 for OK, -1 on error
```

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

- Returns 1 for true, 0 for false, -1 for error

### Beispiel:

```
void pr_mask(const char *str) {  
    sigset_t    sigset;  
    int         errno_save;  
  
    errno_save = errno; /* we can be called by signal handlers */  
    if (sigprocmask(0, NULL, &sigset) < 0)  
        err_sys("sigprocmask error");  
  
    printf("%s", str);  
    if (sigismember(&sigset, SIGINT))  printf("SIGINT ");  
    if (sigismember(&sigset, SIGQUIT)) printf("SIGQUIT ");  
    if (sigismember(&sigset, SIGUSR1)) printf("SIGUSR1 ");  
    if (sigismember(&sigset, SIGALRM)) printf("SIGALRM ");  
    /* remaining signals can go here */  
  
    printf("\n");  
    errno = errno_save;  
}
```

string parameter

... will be  
list of signal  
mask

(Adapted from

## Waiting for Signals

```
int pause(void);
```

```
int sigsuspend(const sigset_t *sigmask);
```

→ Threads halten bis ein Signal mit Handler erhalten werden  
→ `sigsuspend` ersetzt die Signal Mask bis zum `return`

```
int sigwait(const sigset_t *set, int *sig);
```

- Selects an already pending signal from set and returns
- Returns 0 if OK, -1 on error
- Signal number is stored in \*sig
- Original signal action is not taken (contrary to `sigsuspend`)
- Signals in set must be part of signal mask (blocked)
- Call blocks if no signal is pending
- If multiple signals are in set and pending, order is not defined

## Signal und Threads

Maske gilt separat für jeden Thread

**Synchrone Signale:** Wird vom Thread selber generiert und von ihm auch behandelt, wenn nicht maskiert

**Asynchrone Signale.** Generiert von außerhalb und wird von genau einem Thread bearbeitet.

- Thread:
1. Be blocked in sigwait for the signal
  2. Does not block the signal (is not in signal mask)

## Dedicated Signal Handler

- All threads block all signals
- Dedicated handler thread uses sigwait() function
- Additional advantage: sigwait() allows synchronous programming
- Easier to understand and write correctly

```
thr_sigsetmask(mask);
while( (signo = sigwait(mask)) > 0) {
    /*handle signal type */
}
```

## Inter-thread Signalling

int **pthread\_kill**(pthread\_t thread, int sig);

Signal *sig* is delivered to thread *thread*

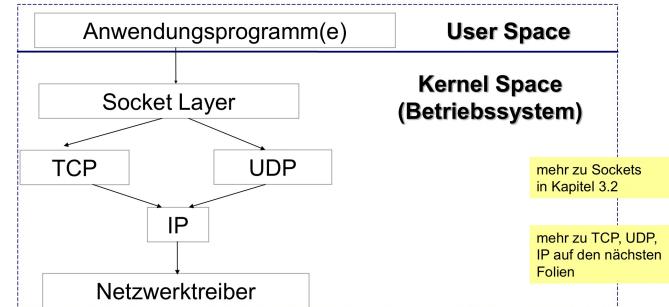
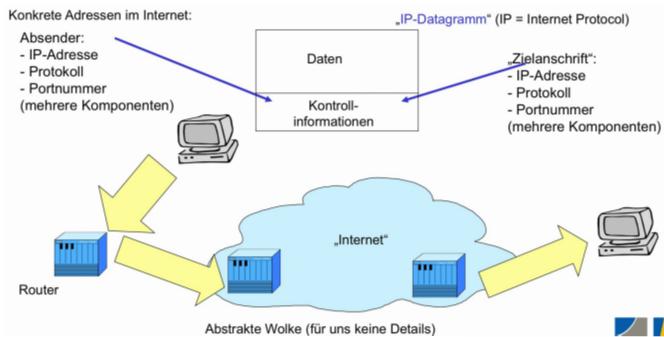
Returns 0 if OK, -1 on error

If *sig* is 0

- Error checking is performed
- No signal is sent
- Can be used to check validity of thread argument

# 3. Netzwerkprogrammierung

## Modell des Internets

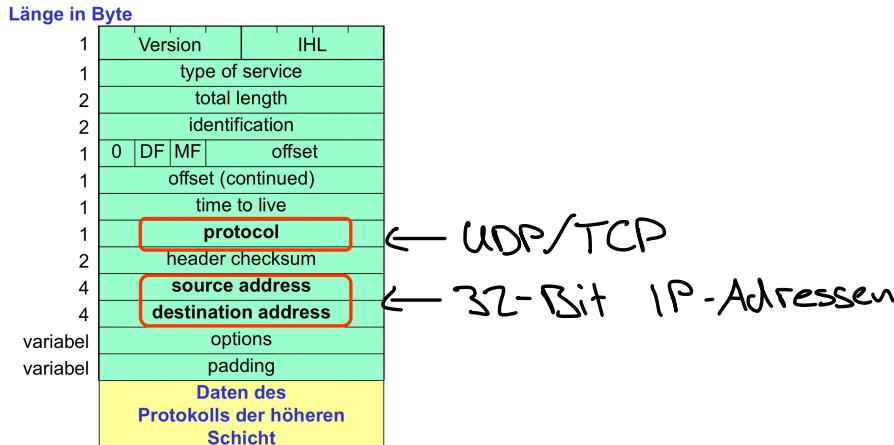


## Internet Protocol (IP)

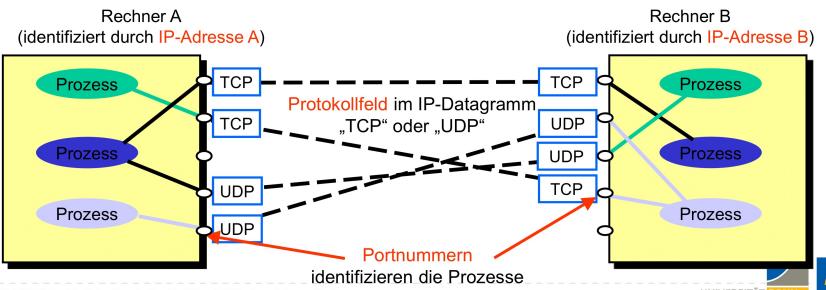
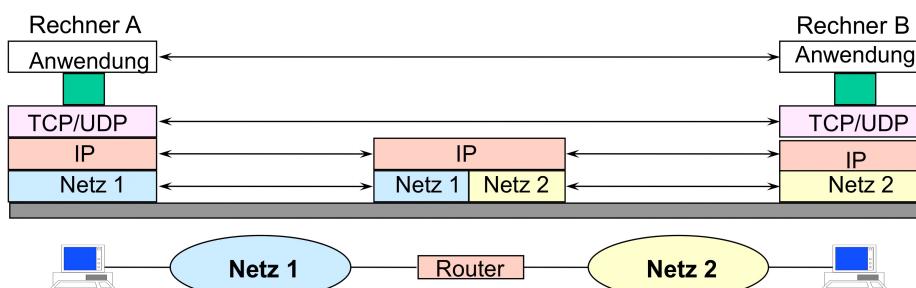
IP ist ein **Netzwerkprotokoll**, was die Basis des Internets bildet.  
IP ist verantwortlich für das Routing (Wegfindung zum Ziel) und alle anderen Netzwerkprotokolle bauen auf IP auf.

**IP-Datagramm:** Datenpakete mit begrenzter Länge (max. 64 KByte), die über IP versendet werden.

→ keine Zusicherung über den Versand von Datagrammen

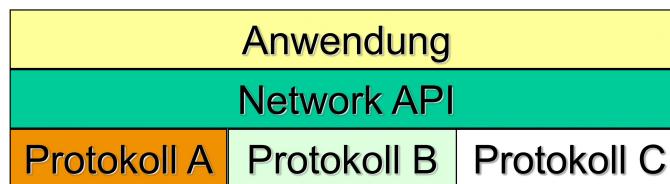


## Anwendung



## Network API

Die Network API wird vom Betriebssystem bereitgestellt und bietet ein Interface für Netzwerkfunktionen, die mit beliebigen Protokollen und Adressdarstellungen verwendet werden können.

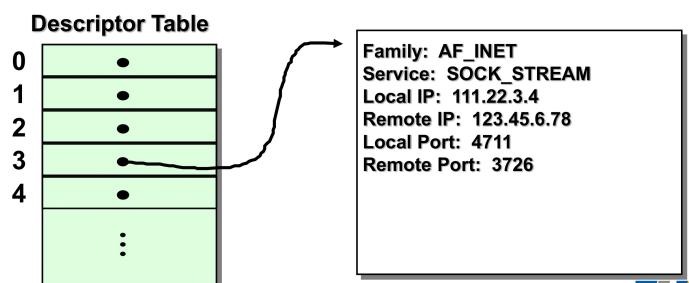


## Sockets

Sockets, die eine generische Network API darstellen, sind eine abstrakte Repräsentation eines kommunizierenden Endpunktes. Sie sind eine Datenstruktur implementiert durch eine Menge von Systemfunktionen.

## Socket Descriptor

Normaler Unix-File-Descriptor mit speziellem Inhalt der Datenstruktur



## Protokollfamilie

1. Internet Domain Sockets (Schlüssel AF\_INET)  
→ implementiert durch IP-Adressen und Ports  
Schlüssel AF\_INET6 für IP Version 6
2. Unix Domain Sockets (Schlüssel AF\_UNIX oder AF\_LOCAL)
3. Novell IPX (Schlüssel AF\_IPX)
4. AppleTalk Datagram Delivery Protocol (Schlüssel AF\_APPLETALK)

## Socket Typen

1. Stream (Schlüssel SOCK\_STREAM)  
→ TCP
2. Datagram (Schlüssel SOCK\_DGRAM)  
→ UDP
3. Direktzugriff auf das Netzwerk (Schlüssel SOCK\_RAW)  
→ übergibt Transportprotokolle

	AF_INET	AF_INET6	AF_LOCAL AF_UNIX	AF_ROUTE	AF_KEY
SOCK_STREAM	TCP	TCP	Ja		
SOCK_DGRAM	UDP	UDP	Ja		
SOCK_RAW	IPv4	IPv6		Ja	Ja

## Erzeugung des Sockets

```
int socket(int domain, int type, int protocol);
```

Domain: Bestimmt Protokollfamilie AF\_INET, AF\_INET6, AF\_LOCAL etc.

Type: Typ des Sockets SOCK\_STREAM oder SOCK\_DGRAM

Protocol: normalerweise 0

Return: Socketdescriptor

## Binden einer Adresse

```
int bind(int sockfd, const struct sockaddr *myaddr,  
        socklen_t addrlen);
```

sockfd: Socket der an die Adresse gebunden wird

myaddr: Zeiger auf die Adresse

addrlen: Längenangabe der Adresse

## Socket-Adressen

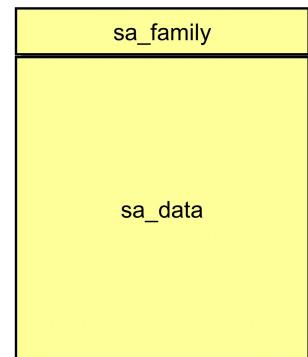
### Generische data structure

```
struct sockaddr {  
    sa_family_t sa_family; /* address family */  
    char sa_data[14]; /* up to 14 bytes of  
                      direct address */  
};
```

sa\_family gibt die Adressfamilie an (AF\_INET, AF\_LOCAL, ...)

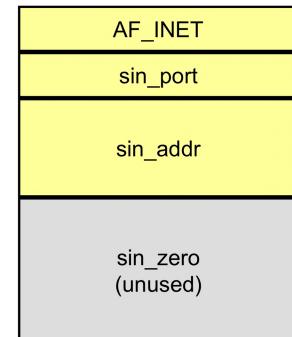
sa\_family\_t entspricht uint16\_t bzw. unsigned short

sa\_data[] kann 14 weitere beliebige Bytes beinhalten  
(abhängig von der konkreten Adressfamilie)



### data structures für IPv4

```
struct in_addr {  
    in_addr_t s_addr; /* 32 bit IPv4 address */  
    /* network byte order! */  
};  
  
struct sockaddr_in {  
    sa_family_t sin_family; /* here: AF_INET */  
    in_port_t sin_port; /* 16-bit TCP/UDP port number */  
    /* network byte order! */  
    struct in_addr sin_addr; /* 32 bit IPv4 address */  
    /* network byte order! */  
    char sin_zero[8]; /* unused, total size 16 bytes */  
};
```



### Adresswahl:

IP-Adresse	Portnummer	Ergebnis
INADDR_ANY INADDR_ANY	0 konkret, > 0	BS wählt IP-Adresse und Port automatisch BS wählt IP-Adresse, Prozess wählt Port selbst
konkrete IP-Adresse konkrete IP-Adresse	0 konkret, > 0	Prozess wählt IP-Adresse selbst, BS wählt Port autom. Prozess wählt IP-Adresse und Port selbst

### Beispiel:

```
int fd, err;  
struct sockaddr_in addr;  
  
fd = socket(AF_INET, SOCK_DGRAM, 0);  
if (fd<0) { ... }  
  
addr.sin_family = AF_INET;  
addr.sin_port = htons(4711);  
addr.sin_addr.s_addr = htonl(INADDR_ANY);  
  
err = bind(fd, (struct sockaddr *)&addr, \  
           sizeof(struct sockaddr_in));  
if (err<0) { ... }
```

## Schließen des Sockets

`int close(int sockfd);`

→ sonst wird der Port nicht freigegeben

## Typdefinitionen

Datentyp	Beschreibung	Zugeh. Headerfile
<code>int8_t</code> <code>uint8_t</code> <code>int16_t</code> <code>uint16_t</code> <code>int32_t</code> <code>uint32_t</code>	<code>signed 8-bit Integer</code> <code>unsigned 8-bit Integer</code> <code>signed 16-bit Integer</code> <code>unsigned 16-bit Integer</code> <code>signed 32-bit Integer</code> <code>unsigned 32-bit Integer</code>	<code>&lt;sys/types.h&gt;</code> <code>&lt;sys/types.h&gt;</code> <code>&lt;sys/types.h&gt;</code> <code>&lt;sys/types.h&gt;</code> <code>&lt;sys/types.h&gt;</code> <code>&lt;sys/types.h&gt;</code>
<code>sa_family_t</code>	address family of socket address structure, normally <code>uint16_t</code>	<code>&lt;sys/socket.h&gt;</code>
<code>socklen_t</code>	length of socket address structure, normally <code>uint32_t</code>	<code>&lt;sys/socket.h&gt;</code>
<code>in_addr_t</code> <code>in_port_t</code>	IPv4 address, normally <code>uint32_t</code> TCP or UDP port, normally <code>uint16_t</code>	<code>&lt;netinet/in.h&gt;</code> <code>&lt;netinet/in.h&gt;</code>

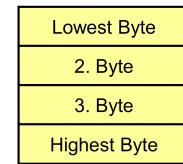


## Byte Order

Die Reihenfolge in der Bytes abgespeichert bzw. gesendet werden unterscheidet sich von System zu System.

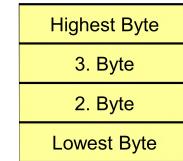
### Little-Endian-Byte-Order:

kleinstwertiges Byte wird zuerst abgespeichert/gesendet



### Big-Endian-Byte-Order:

Höchstwertiges Byte wird zuerst abgespeichert/gesendet



### Host Byte Order:

architekturabhängige Darstellung

### Network Byte Order:

Byte Order der Netzwerkfunktionen, die architektur-unabhängig sind!

### TCP, UDP, IP: Big-Endian-Byte-Order

→ muss ggf. umgewandelt werden

## Umwandlung Network Byte Order

```
uint16_t htons(uint16_t hvalue); /* Host to Network, 16 bits */
uint32_t htonl(uint32_t hvalue); /* Host to Network, 32 bits */
uint16_t ntohs(uint16_t hvalue); /* Network to Host, 16 bits */
uint32_t ntohl(uint32_t hvalue); /* Network to Host, 32 bits */
```

## Darstellung von IP-Adressen

Dotted-Decimal-Schreibweise: Schreibweise zur besseren Leserlichkeit

### Umwandlung:

```
in_addr_t inet_addr(const char *dotted); /* Dotted to Network */
char *inet_ntoa(struct in_addr network); /* Network to Dotted */
```

## Hilfsfunktionen zur Adressbestimmung

Gewünschte Information aus dem IP-Datagramm des Clients (Quelle= Client, Destination = Server)	TCP Server	UDP Server
Quell-IP-Adresse	accept() oder auch getpeername()	recvfrom()
Quell-Portnummer	accept() oder auch getpeername()	recvfrom()
Destination-IP-Adresse	getsockname()	recvmsg()
Destination-Portnummer	getsockname()	getsockname()

## Server - Strukturen

**Iterativer Server:** Server verarbeitet zu jeder Zeit genau eine Client-Auffrage  
→ kurze Anfragen

**Nebenläufiger Server:** Server kann mehrere Client-Aufgaben gleichzeitig bearbeiten  
→ längere, komplexere Anfragen

**Verbindungslos:** weniger overhead und keine Begrenzung an Clients

**Verbindungsorientiert:** einfacher zu programmieren, da das Transportprotokoll viele Aufgaben übernimmt

- Designansätze:**
1. Ein Child-Prozess/Thread pro Client
  2. **Prefork:** Prozesse/Threads werden auf Clientanfragen vorbereitet
  3. Select Loop

**State:** Information, die ein Server über eine Client-Interaktion aufreht erhält

**Statelessness:** Server merkt sich keine Information über Client-Interaktionen

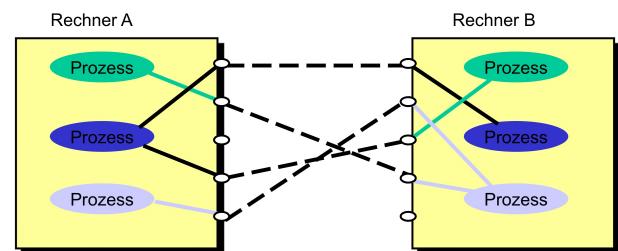
**Statefullness:** Server merkt sich den Zustand der Interaktion mit einem Client

# User Datagram Protocol (UDP)

UDP ist ein Transportprotokoll, welches IP benutzt, um Datagramme verbindungslos zwischen Endgeräten zu verschicken. Kommuniziert wird über Ports. UDP bietet dabei keinerlei Garantien über das Ankommen der Datagramme, bietet aber eine minimale und einfache Implementierung.

## UDP-Datagramm:

Format des UDP Datagramms:

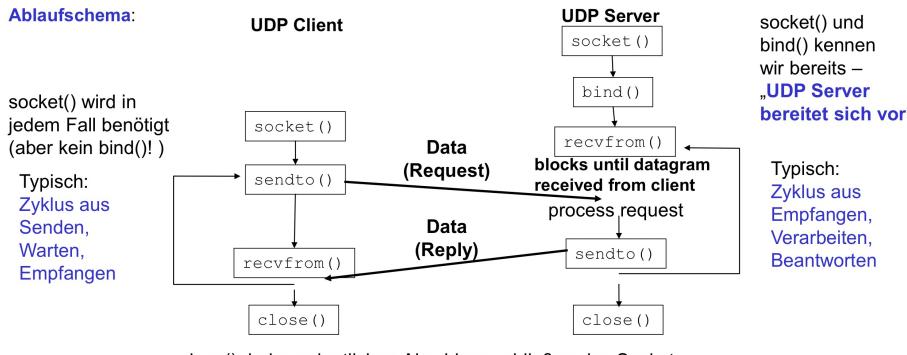


## „Well-known“ Ports

Decimal	Description
0	Reserved
7	Echo
9	Discard
11	Active Users
13	Daytime
15	Who is up or NETSTAT
17	Quote of the Day
19	Character Generator
<b>22</b>	<b>ssh (secure shell)</b>
37	Time
42	Host Name Server
43	Who is
<b>53</b>	<b>Domain Name Server</b>
67	Bootstrap Protocol Server
68	Bootstrap Protocol Client
69	Trivial File Transfer
<b>80</b>	<b>World Wide Web HTTP</b>
111	Sun Microsystems RPC
123	Network Time Protocol
161	SNMP net monitor
162	SNMP traps
512	UNIX comsat
513	UNIX rwho daemon
514	system log
525	Time daemon

## Kommunikation über Sockets

Ablaufschema:



`close()`, beim ordentlichen Abschluss schließen des Sockets

## Senden

```
int sendto(int sockfd, const void *buff, size_t nbytes,
           int flags, const struct sockaddr *to, socklen_t tolen);
```

## Parameter:

- sockfd File Descriptor des Sockets
- buff Zeiger auf den Puffer mit den zu sendenden Daten
- nbytes Länge der zu sendenden Daten in Bytes
- flags Optionen, standard ist 0 (weitere ggf. später)
- to Zeiger auf Socket-Adress-Struktur mit der Zieladresse (Port + IP-Adr.)
- tolen Länge der Adress-Struktur

```

char msg[64];
int err;
struct sockaddr_in dest;

strcpy(msg,"hello, world!");

dest.sin_family = AF_INET;
dest.sin_port = htons(4711);
dest.sin_addr.s_addr = inet_addr("130.37.193.13");

err = sendto(sockfd, msg, strlen(msg)+1, 0, \ 
(struct sockaddr*)&dest, sizeof(struct sockaddr_in));

if (err<0) { ... } Casting der Adress-Struktur
    
```

## Empfangen

```
int recvfrom(int sockfd, void *buff, size_t nbytes,
             int flags, struct sockaddr *from, socklen_t *fromlen);
```

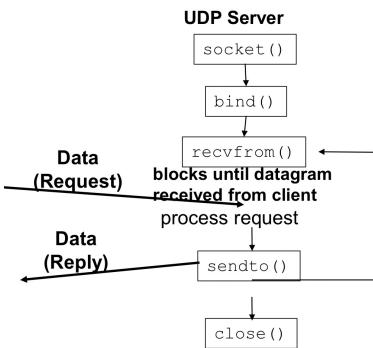
→ blockiert bis ein Datagramm empfangen wurde

### Parameter:

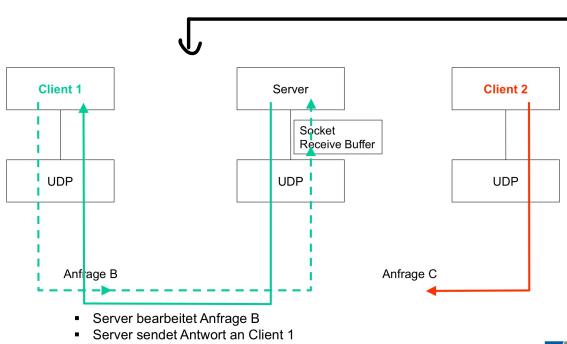
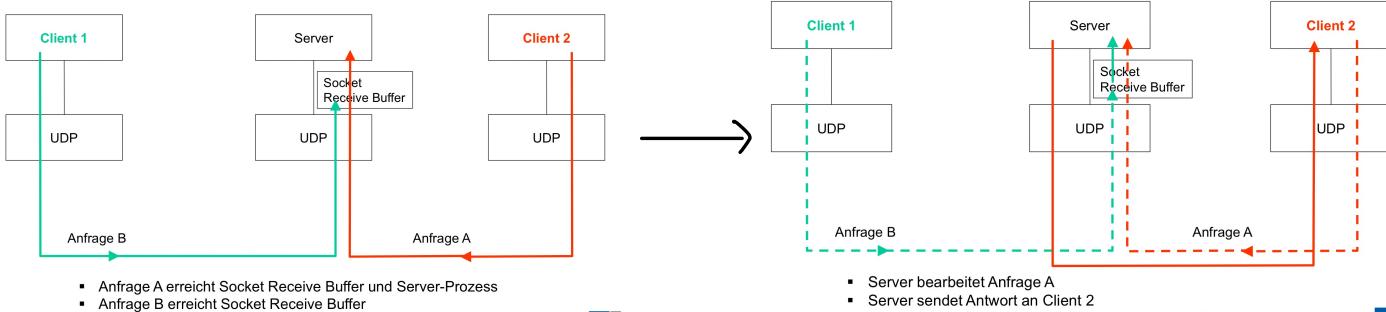
- sockfd File Descriptor des Sockets
- buff Zeiger auf einen Puffer, in den empfangene Daten geschrieben werden
- nbytes Länge des Puffers
- flags Optionen, standard ist 0 (weitere ggf. später)
- from Zeiger auf Socket-Adress-Struktur mit der Quelladresse (Port + IP-Adr.)
- fromlen Zeiger auf Integer mit Länge der Quell-Adress-Struktur

```
char msg[64];
int len, flen;
struct sockaddr_in from;
flen = sizeof(struct sockaddr_in);
len = recvfrom(sockfd, msg, sizeof(msg), 0, &from, &flen);
if (len<0) { ... }
printf("Received %d bytes from host %s port %d: %s", len,
inet_ntoa(from.sin_addr), ntohs(from.sin_port), msg);
```

## Iterativer UDP-Server



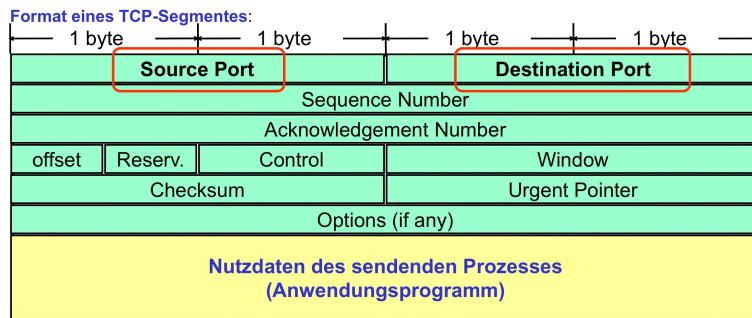
1. Blockieren bis Client-Anfrage empfangen (Client-Adresse aus recvfrom())
2. Verarbeiten der Anfrage
3. Senden einer Antwort an die entsprechende Client-Adresse
4. Weiter bei 1., warte auf nächste Anfrage



# Transmission Control Protocol (TCP)

TCP ist ein verbindungsorientiertes Transportprotokoll. Vor der Kommunikation zwischen zwei Geräten, muss eine Verbindung aufgebaut werden. Über die Verbindung kann full-duplex kommuniziert, indem Daten als Byteströme versendet werden. TCP sorgt dabei zu, dass die Daten korrekt und vollständig ankommen.

## TCP-Segment



## "Well-known" Ports

Decimal	Description
0	Reserved
1	TCP Multiplexer
5	Remote Job Entry
7	Echo
9	Discard
11	Active Users
13	Daytime
15	Network status program
17	Quote of the Day
19	Character Generator
20	File Transfer Protocol (data)
21	File Transfer Protocol
22	ssh (secure shell)
23	Telnet
25	SMTP (Mail Transfer)
37	Time
42	Host Name Server
43	Who is

Decimal	Description
53	Domain Name Server
77	Any private RJE service
79	Finger
80	World Wide Web HTTP
93	Device Control Protocol
95	SUPDUP Protocol
101	NIC Host Name Server
102	ISO-TSAP
103	X.400 Mail Service
104	X.400 Mail Sending
111	Sun Microsystems RPC
113	Authentication Service
117	UUCP Path Service
119	USENET News Transfer Protocol
129	Password Generator Protocol
139	NETBIOS Session Service
160-223	Reserved

# Kommunikation über Sockets

## Ablaufschema:

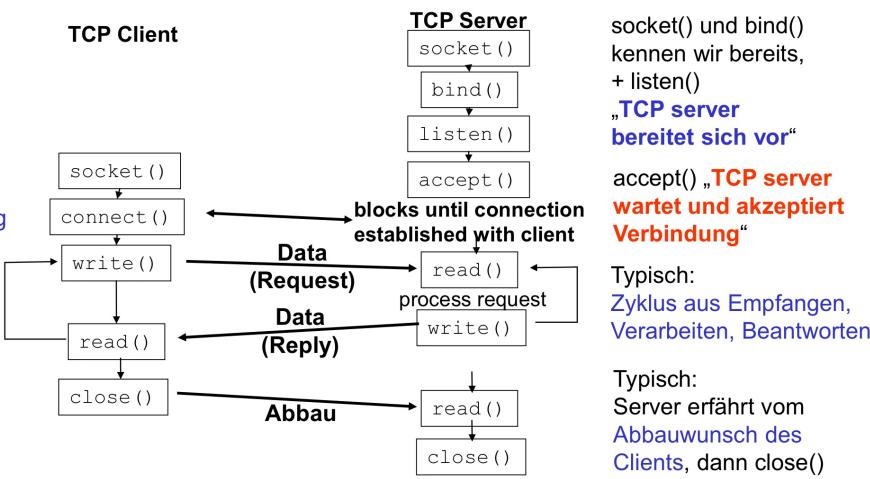
socket() wird in jedem Fall benötigt (aber kein bind()!)

connect() initiiert Aufbau einer Verbindung

Typisch:  
Zyklus aus Senden, Warten, Empfangen

Typisch:  
close() beim Client initiiert Abbau der Verbindung

Erinnerung:



socket() und bind()

kennen wir bereits,

+ listen()

„TCP server

bereitet sich vor“

accept() „TCP server wartet und akzeptiert Verbindung“

Typisch:  
Zyklus aus Empfangen, Verarbeiten, Beantworten

Typisch:  
Server erfährt vom Abbauwunsch des Clients, dann close()

## Verbindungsauftreten

### Server

#### 1. listen()-Funktion

Socket wird in einen passiven Zustand gesetzt  
→ ist nur verantwortlich für einkommende Verbindungsauftretewünsche

`int listen(int sockfd, int backlog);`

**Parameter:** sockfd Socket File Descriptor  
backlog Anzahl der gleichzeitig möglichen Verbindungsauftreteworgänge  
(typischer Wert: 5)

**Handshake:** Mehrere Schritte beim Verbindungsauftreten zwischen Server und Client

#### 2. accept()-Funktion

Socket nimmt Verbindungsauftretewünsche entgegen und blockiert solange bis eine Verbindung hergestellt wurde

`int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);`

**Parameter:** sockfd Socket File Descriptor  
addr Zeiger auf Adress-Struktur, in der die **Adresse des Initiators** des Verbindungsauftretens abgelegt wird (→ darin konkrete IP-Adresse + Port!)  
addrlen Zeiger auf Länge der Adress-Struktur

Dort auf

**Return:** Deskriptor eines neu kreierten Sockets über den kommuniziert wird

### Client

#### Connect()-Funktion

`int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);`

**Parameter:** sockfd Socket File Descriptor  
serv\_addr Zeiger auf **Adress-Struktur eines Servers**, zu dem eine Verbindung werden soll  
(→ darin konkrete IP-Adresse + Port!)  
addrlen Länge der Adress-Struktur

## Beispiel:

```
int sockfd, newsock, res;
sockaddr_in client_addr;
socklen_t addrlen;

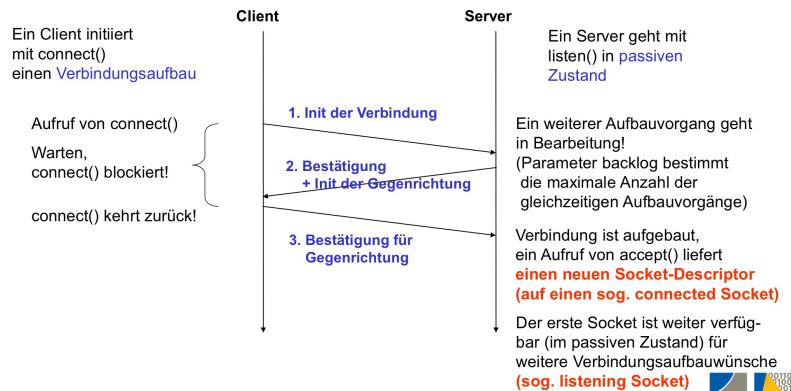
... /* the socket sockfd has been created
     and bound to a port number */

res = listen(sockfd, 5); ← Standard-Aufruf
if (res<0) { ... }

addrlen = sizeof(struct sockaddr_in);

newsock = accept(sockfd, (struct sockaddr *) &client_addr, &addrlen);
if (newsock<0) { ... } ← Neuer Socket-Descriptor kann ab jetzt
else {                                                 benutzt werden!
    printf("Received connection from %s!\n",
           inet_ntoa(client_addr.sin_addr));
}

} Hilfsfunktion zur Ausgabe der
IP-Adresse als Dotted-Decimal
```



## Kommunikation

Client und Server sind gleichberechtigt in der Kommunikation.  
Senden und Empfangen von Daten muss untereinander  
abgestimmt werden

### Daten Senden

```
ssize_t write(int sockfd, const void *buff, size_t count);
```

**Parameter:** sockfd File Descriptor des Sockets  
buff Zeiger auf den Puffer mit den zu sendenden Daten  
count Länge der zu sendenden Daten in Bytes

**Return:** Anzahl gesendeter Bytes  
→ erneut senden, wenn kleiner als gewünschte Anzahl

### Daten Empfangen

```
ssize_t read(int sockfd, void *buff, size_t count);
```

→ blockiert bis Daten empfangen werden

**Parameter:** sockfd File Descriptor des Sockets  
buff Zeiger auf einen Puffer, in den empfangene Daten geschrieben werden  
count Länge des Puffers

### Schließen (Alternative)

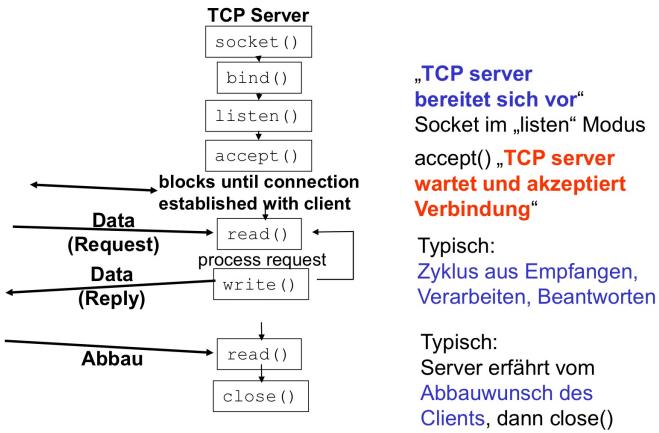
```
int shutdown(int sockfd, int howto);
```

→ erlaubt Schließen von Lese- und Schreibrichtung einzeln

**Parameter:** sockfd File Descriptor des Sockets  
howto Variante des Schließens:

- SHUT\_RD Schließen der Leserichtung
- SHUT\_WR Schließen der Schreibrichtung
- SHUT\_RDWR Schließen beider Richtungen

# Iterativer TCP-Server



„TCP server bereitet sich vor“  
Socket im „listen“ Modus  
accept() „TCP server wartet und akzeptiert Verbindung“

Typisch:  
Zyklus aus Empfangen, Verarbeiten, Beantworten

Typisch:  
Server erfährt vom Abbauwunsch des Clients, dann close()

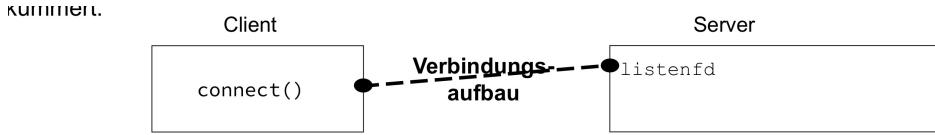
```

int sock_listen_fd, newfd;
...
/* sock_listen_fd vorbereitet */

while (1) {
    newfd = accept(sock_listen_fd, ...);
    treat_request(newfd);
    close(newfd);
}
    
```

# Nebenläufiger TCP-Server mit fork()

Für jede Client-Aufgabe wird ein neuer Prozess erstellt

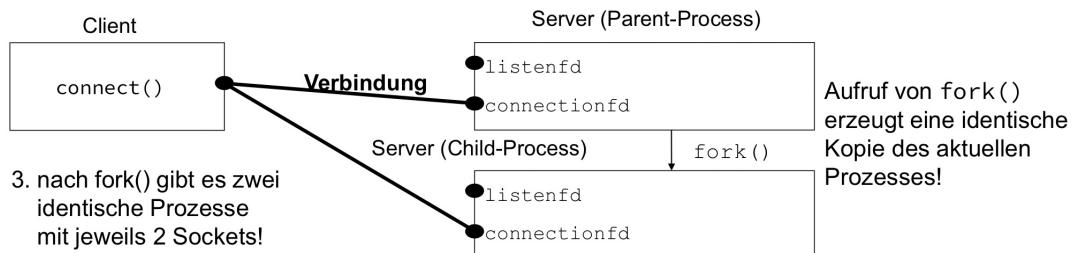


1. der Client initiiert connect(), der Server wartet auf Rückkehr von accept()

Vgl. Kap 2.1. Prozesse  
SysProg

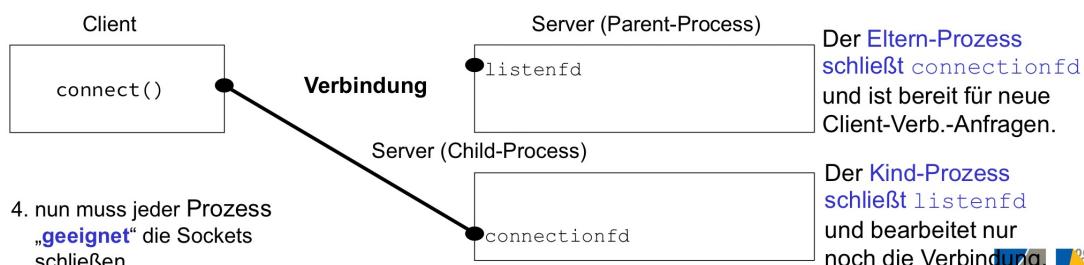


2. nach Rückkehr von accept() läuft die Verbindung auf dem neuen Socket



3. nach fork() gibt es zwei identische Prozesse mit jeweils 2 Sockets!

Aufruf von fork()  
erzeugt eine identische Kopie des aktuellen Prozesses!



4. nun muss jeder Prozess „geeignet“ die Sockets schließen

Der Eltern-Prozess schließt connectionfd und ist bereit für neue Client-Verb.-Anfragen.

Der Kind-Prozess schließt listenfd und bearbeitet nur noch die Verbindung.

## Beispiel:

```

int main() {
    int sock_listen_fd, newfd, child_pid;
    ...
    /* sock_listen_fd vorbereitet */

    while (1) {
        newfd = accept(sock_listen_fd, ...);

        if (newfd<0) /* Fehlerbehandlung */
            child_pid = fork();

        if (child_pid==0) {
            close(sock_listen_fd);
            treat_request(newfd);
            close(newfd);
            exit(0);
        }
        else { close(newfd); }
    }
}
    
```

## Preforked nebenläufiger TCP-Server

Teure `fork()`-Operationen werden vorab vom Parent-Prozess durchgeführt. So kann auch die Client-Anzahl beschränkt werden.

Auf manchen Betriebssystemen müssen `accept()` Aufrufe synchronisiert werden.

```
void recv_requests(int fd) { /* An iterative server */
    int f;

    while (1) {
        f=accept(fd,...);
        treat request(f);
        close(f);
    }
}

int main() {
    int fd;
    ...           /* Listening Socket fd vorbereitet */

    for (int i=0;i<NB PROC;i++) { /* Create NB_PROC children */
        if (fork()==0) recv_requests(fd);
    }
    while (1) pause(); /* The parent process does nothing */
}
```

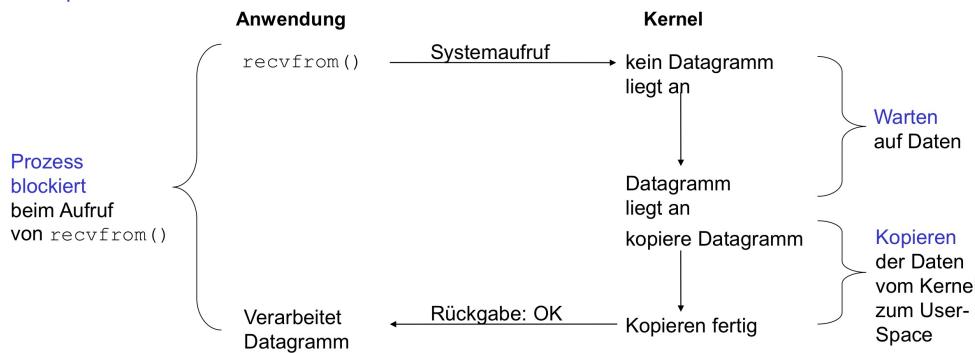
Parent generiert 10 Child-Prozesse  
Parent macht danach gar nichts mehr!

# Input/Output Multiplexing

**Prinzip:** Ein Programm soll gleichzeitig Eingaben aus verschiedenen Quellen bearbeiten können.

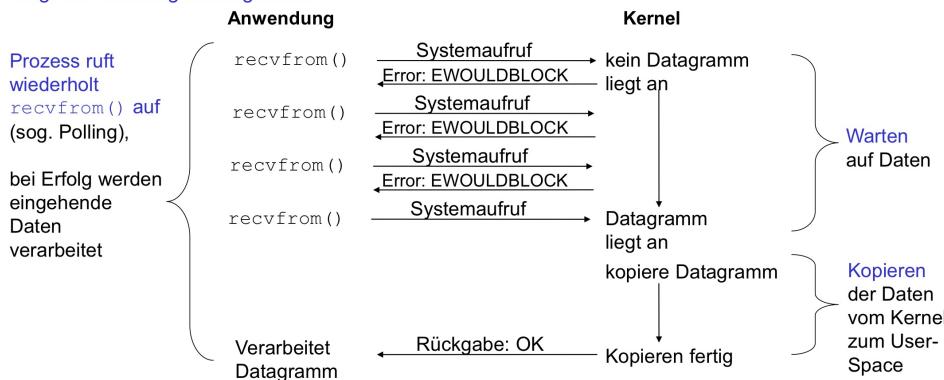
## Blockierender I/O

1. Warten auf das „Eintreffen“ von Daten (Tastatur, Netzwerk, ...)
2. Kopieren der Daten vom Kernel zum Prozess



## Nicht-blockierender I/O

Zur Vorbereitung muss zunächst ein **Socket** (oder ein allg. File Descriptor) in den sog. **non-blocking Mode gesetzt** werden.



## Non-blocking mode:

```
int flags, err;
/* sockfd sei Descriptor eines TCP-Sockets */

flags = fcntl(sockfd, F_GETFL, 0);
err = fcntl(sockfd, F_SETFL, flags | O_NONBLOCK);
```

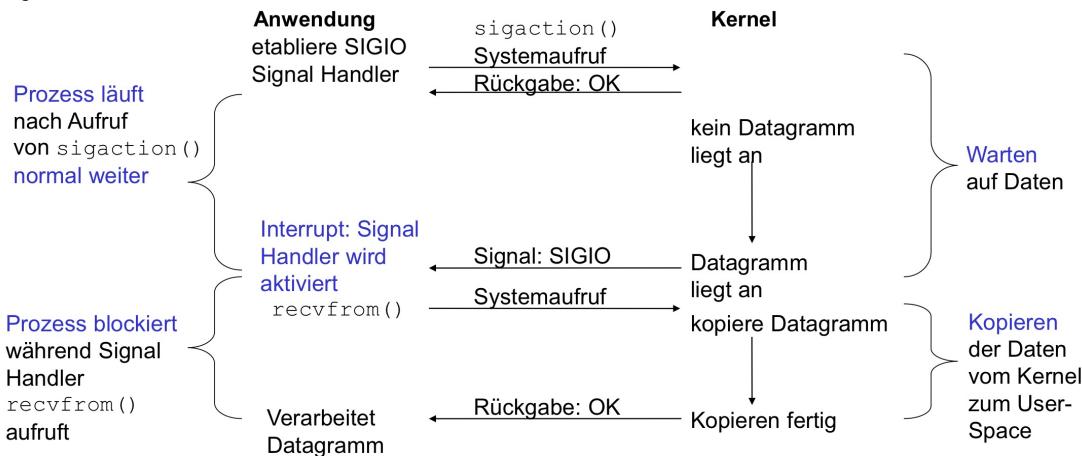
## Beispiel:

```
while (!done) {
    if ((n=read(STDIN_FILENO,...)<0))
    {
        if (errno!=EWOULDBLOCK)
            /* ERROR */
        else write(sockfd,...)
    ...
    if ((n=read(sockfd,...)<0))
    {
        if (errno != EWOULDBLOCK)
            /* ERROR */
        else write(STDOUT_FILENO,...)
    ...
}
```

**Busy Waiting:** Der Prozess ist in einer Endlosschleife solange er wartet und verbraucht unnötig CPU-Ressourcen

# Signal-gesteuerter I/O

Zur Vorbereitung muss zunächst der Socket für signal-driven I/O vorbereitet und ein sog. Signal Handler für das Signal SIGIO aktiviert werden.



## Alternativen

1. Signal Handler ruft Daten ab
  2. Signal Handler füllt Hauptroutine mit Daten abzurufen
- Busy Waiting  
→ sehr komplex für TCP

## I/O-Multiplexing mit Hilfsfunktion `select()`

### Select()-Funktion

Stellt universelle Funktionalitäten für I/O-Multiplexing mehrerer Eingabeketten zur Verfügung unter Angabe aller File Descriptors, die beachtet werden sollen überprüft, ob Daten an einem File Descriptor anliegen.

```
int select(int maxfdp1, fd_set *readfds, fd_set *writefds,  
          fd_set *exceptfds, struct timeval *timeout);
```

#### Parameter:

- |                        |  |
|------------------------|--|
| <code>maxfdp1</code>   | die Nummer des höchsten File Descriptors plus 1 „max fd p 1“ = Anzahl der zu prüfenden File Descriptoren |
| <code>readfds</code>   | Liste von File Descriptoren, die zum Lesen überprüft werden  |
| <code>writefds</code>  | Liste von File Descriptoren, die zum Schreiben überprüft werden  |
| <code>exceptfds</code> | Liste von File Descriptoren, die auf Exceptions überprüft werden   |
| <code>timeout</code>   | Zeitintervall, nach dem <code>select()</code> in jedem Fall zurückkehrt                                  |

#### Return:

- >0 Anzahl der Descriptoren im Zustand „ready for I/O“
- = 0 Timeout (ohne bereite Descriptoren)
- = -1 Fehler

#### Timeout

#### data structure:

```
struct timeval {  
    long tv_sec /* seconds */;  
    long tv_usec /* microseconds */;  
};
```

#### Möglichkeiten:

1. unendlich warten (blockierend)
2. Warte gar nicht (nicht-blockierend)
3. Warten einer spezifischen Zeit

## Manipulation von File-Descriptor Sets

File Descriptor Sets werden als Bit-Flags implementiert, wobei jedes Bit angibt, ob der File Descriptor beinhaltet ist

Hilfsfunktion: FD\_ZERO(fd\_set \*set); /\* clears all bits \*/

FD\_SET(int fd, fd\_set \*set); /\* turns on bit fd \*/

FD\_CLR(int fd, fd\_set \*set); /\* turns off bit fd \*/

FD\_ISSET(int fd, fd\_set \*set); /\* checks if bit fd is set \*/

→ immer mit FD\_ZERO initialisieren

## Ablaufschema für select()

1. Löschen der Menge (Bit-Flags) mit FD\_ZERO
2. zu überprüfende Descriptoren hinzufügen mit FD\_SET
3. eigentlicher Aufruf von select()
4. nach Rückkehr von select() Rückgabewert überprüfen (>0, 0, -1)  
und ggf. mit FD\_ISSET ein oder mehrere bereite File Descriptoren bearbeiten

## Socket File Descriptor bereit

1. Ein Socket wird bereit zum Lesen (eingetragen in Menge readfds)

- es liegen Daten zum Lesen an (1 Datagramm UDP, x Bytes TCP)
- der Leseteil eines TCP-Sockets wurde geschlossen  
d.h. der Kommunikationspartner hat seinen Schreibteil geschlossen  
=> Aufruf von read() wird 0 zurückliefern (vgl. vorne)
- ein TCP listening Socket hat einen komplettierten Verbindungsaufbauwunsch  
=> Aufruf von accept() wird erfolgreich sein
- ein Socket-Error liegt vor, read() liefert -1, errno gibt weiteren Aufschluss  
(im Falle eines Socket-Errors wird readable und writeable markiert, s.u.)

2. Ein Socket wird bereit zum Schreiben (eingetragen in Menge writefds)

- der Socket Sendepuffer hat genügend Platz zum Schreiben weiterer Daten  
(aus diesem Puffer werden zu sendende Daten an den Kernel übergeben. Bei nicht-blockierendem Schreiben könnte man diesen Puffer überfluten!)
- der Schreibteil eines Sockets ist geschlossen
- nach einem nicht-blockierenden Aufruf von connect() wurde der Verbindungsauftbau abgeschlossen, erfolgreich oder fehlerhaft
- ein Socket-Error liegt vor, write() liefert -1, errno gibt weiteren Aufschluss  
(im Falle eines Socket-Errors wird readable und writeable markiert, s.u.)

3. Beim Socket liegt eine Ausnahmebedingung vor, Exception  
(eingetragen in Menge exceptionfds)

Selten benutzer Fall:

- am TCP-Socket liegen sog. „Out-of-band“ Daten an  
(wichtige Daten, die „normale“ Daten überholen sollen)  
(mehr Informationen für Interessierte im Stevens, Ch. 24)

## Beispiel:

```
int res, fd1=5, fd2=8;
char buf[1024];
fd_set rset;

while (1) {
    FD_ZERO(&rset);
    FD_SET(fd1, &rset);
    FD_SET(fd2, &rset);

    res = select(9, &rset, NULL, NULL, NULL);
    if (res<=0) { ... }

    if (FD_ISSET(fd1,&rset)) {
        bzero(buf,1024);
        res = read(fd1,buf,1024);
        if (res<0) { ... }
        if (res==0) printf("Received EOF on fd1!\n");
        else printf("Received data on fd1: %s\n", buf);
    }

    if (FD_ISSET(fd2,&rset)) { ... }
}
```

Blockiere nur Abfrage

Jeder mögliche  
Socket muss  
überprüft werden