Koordination durch Kommunikation

Interprozesskommunikation (engl. *inter-process communication*, IPC)

Interaktion von Prozessen ist zwingend, um in einem Mehrprozesssystem Fortschritte in der Programmverarbeitung zu erreichen, und zwar:

implizit innerhalb des Betriebssystems

- nebenläufige Ausführung mehrfädiger Systemprogramme:
  - asynchrone Programmunterbrechungen
  - verdrängende Prozesseinplanung
  - ggf. auch SMP (engl. *symmetric multiprocessing*)
- die Prozesse konkurrieren um die Betriebsmittelzuteilung

explizit innerhalb des Anwendungssystems

- arbeitsteilige Ausführung eines Programms durch mehrere Prozesse
  - paralleles/verteiltes Programm
- die Prozesse kooperieren bei der gemeinsamen Programmausführung

Nebenläufigkeit „*Considered Harmful*“

Kritischer Abschnitt (engl. *critical section*)

Rückblick: nebenläufiges Zählen (S. III-51)

- `wheel++` ist nicht immer eine unteilbare Operation
  - diese Operation der Ebene $_5$ ist nur „scheinbar elementar“
  - ggf. bildet sie eine Sequenz von Elementaroperationen der Ebene $_4$
  - in dem Fall wäre sie eine teilbare Operation und damit kritisch
- unterbrechungsbedingte Überlappungs(d)effekte können die Folge sein

Warteschlangen, z.B., stellen andere potentielle „Brennpunkte“ dar

- oft ist eine beliebige Permutation der Zugriffsoperationen möglich
  - eintragen überlappt austragen bzw. sich selbst, und umgekehrt
- auch die Auslegung der Datenstruktur „Schlange“ ist von Bedeutung

☞ „Untiefen“ dieser Art gibt es einige in Betriebssystemen. . .

Einreihung in eine einfach verkettete Liste

„Elementaroperation“ zum Anhängen eines Kettenglieds

```
typedef struct chainlink {
    struct chainlink *link;
} chainlink;

void chain (chainlink **next, chainlink *item) {
    *next = (*next)->link = item;
}
```

Die Funktion hängt ein neues Kettenglied hinter `*next` ein: `(*next)->link = item`. Der Zuweisungswert ist gleichfalls die Adresse des Kettenglieds, an dem das nächste Kettenglied angehängt werden soll. Diese Adresse wird vermerkt: `*next =` *Wert*. Damit ist `next` Einfügezeiger in eine einfach verkettete Liste.

`gcc -O6 -fomit-frame-pointer -S chain.c`. Auf der Assembliersprachenebene ist erkennbar, dass die Einfügeoperation als Folge von Einzelschritten auf der CPU zur Ausführung kommt. Im Falle der nicht-sequentiellen Ausführung ist nach jedem Einzelschritt mit einer asynchronen Programmunterbrechung zu rechnen, die eventuell die Verdrängung des laufenden Prozesses bewirkt und einen anderen Prozess startet, der dieselbe Funktion zeitlich überlappend (und damit nebenläufig) ausführen könnte.

x86

```
chain:
   movl 4(%esp), %ecx
   movl 8(%esp), %edx
   movl (%ecx), %eax
   movl %edx, (%eax)
   movl %edx, (%ecx)
   ret
```

PPC

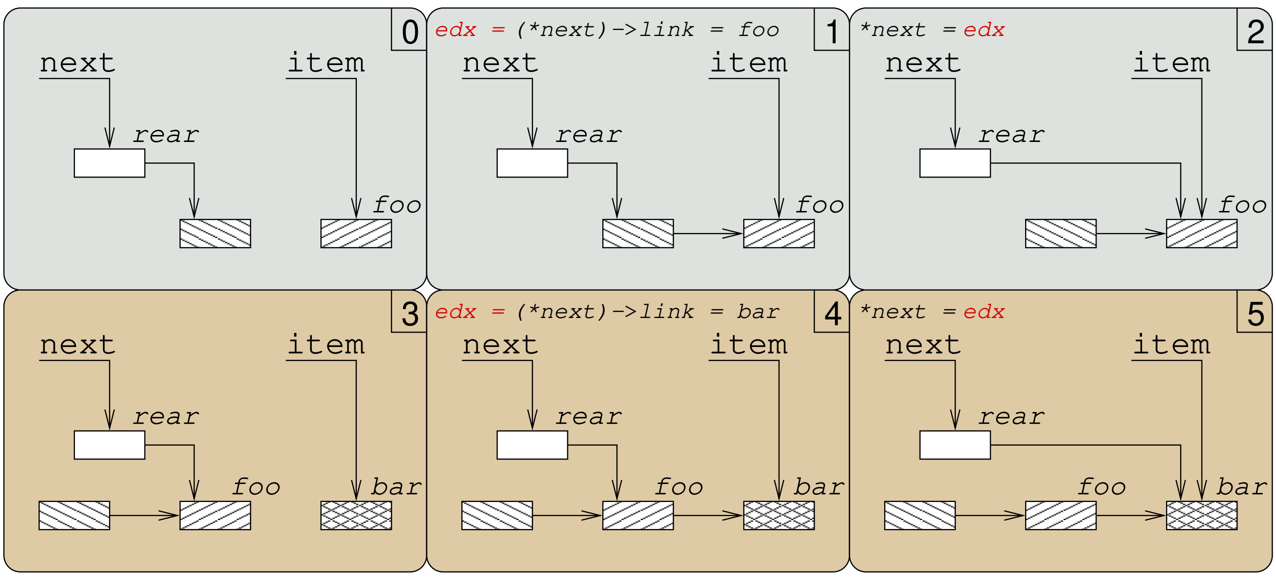
```
_chain:
   lwz r5,0(r3)
   stw r4,0(r5)
   stw r4,0(r3)
   blr
```


Einreihung in eine einfach verkettete Liste (Forts.)

Sequentielle Ausführung

```
chainlink *rear, *foo, *bar;
```

`chain(&rear, foo);`



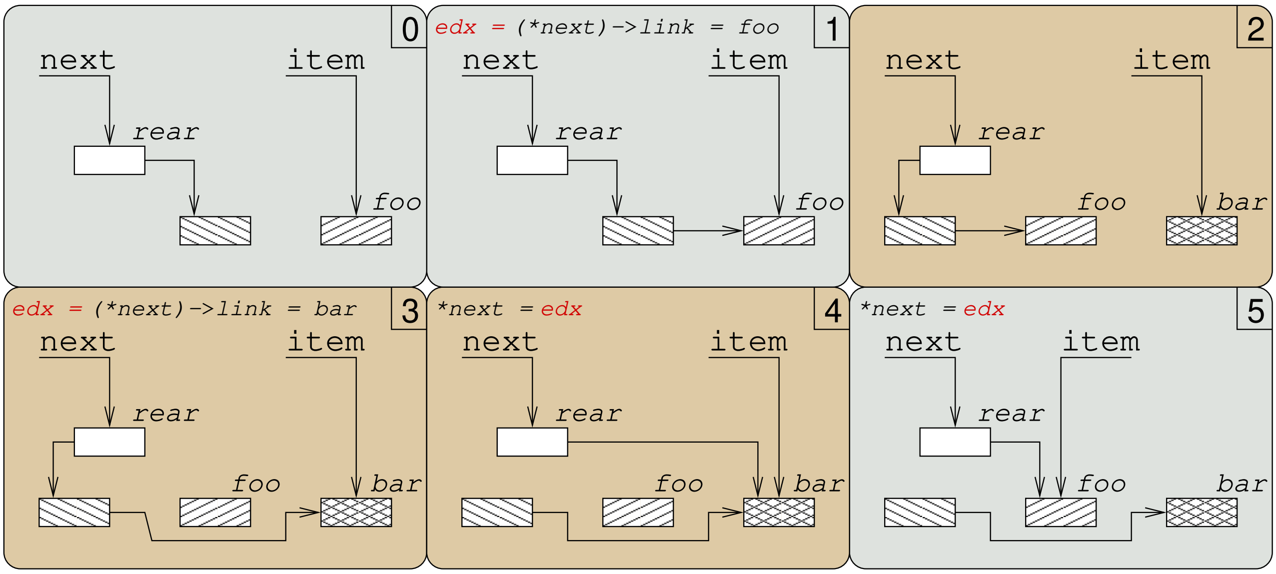
`chain(&rear, bar);`

Einreihung in eine einfach verkettete Liste (Forts.)

Nicht-sequentielle Ausführung

```
chainlink *rear, *foo, *bar;
```

chain(&rear, foo); ······ chain(&rear, bar); ······



······ ▻chain(&rear, foo); ··· ▻

Koordinationsvariable

Semaphor (engl. *semaphore*)

Eine „nicht-negative ganze Zahl“, für die zwei unteilbare Operationen definiert sind [54]:

P (hol. *prolaag*, „erniedrige“; auch *down*, *wait*)
- hat der Semaphor den Wert 0, wird der laufende Prozess blockiert
- ansonsten wird der Semaphor um 1 dekrementiert

V (hol. *verhoog*, erhöhe; auch *up*, *signal*)
- inkrementiert den Semaphor um 1
- auf den Semaphor ggf. blockierte Prozesse werden deblockiert

Ein abstrakter Datentyp zur Signalisierung von Ereignissen zwischen gleichzeitigen Prozessen (deren Ausführung sich zeitlich überschneidet).

Koordination von Kooperation und Konkurrenz

Sequentialisierung nicht-sequentieller Programme

Synchronisation (engl. *synchronization*) bringt die Aktivitäten von verschiedenen Prozessen in eine Reihenfolge [55, S. 26]:

- dadurch wird prozessübergreifend das erreicht, wofür innerhalb eines Prozesses die Sequentialität von Aktivitäten sorgt
- Nebenläufigkeit bzw. Parallelität wird damit gezielt unterbunden

```
void chain (chainlink **next, chainlink *item) {
    P();
    *next = (*next)->link = item;
    V();
}
```

`P()` und `V()` klammern den kritischen Abschnitt: `P()` stellt sicher, das die Anweisungen bis zum `V()` nicht zugleich von mehreren Prozessen ausgeführt werden können.

UNIX Systemfunktionen

Operationen auf Semaphore

Linux, MacOS, SunOS

```
id  = semget(key, nsem, flag)
val = semctl(id, semnum, cmd, ...)
ok  = semop(id, sembuf, nops)
    ⋮
```

Sequenzen von Semaphoroperationen können unteilbar ausgeführt werden

- technisch ist die Sequenz als ein `sembuf`-Feld repräsentiert

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

- jede `sembuf`-Instanz beschreibt eine (ggf. andere) auszuführende Operation
- die `sembuf`-Reihenfolge bestimmt die Ausführungsreihenfolge der Operationen

UNIX Semaphor

Einrichten und initialisieren einer Koordinationsvariablen

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

int mutex;
struct sembuf sema;

int main () {
    mutex = semget(IPC_PRIVATE, 1, IPC_CREAT);
    if (mutex == -1) perror("semget");
    else if (semctl(mutex, 0, SETVAL, 1) == -1)
        perror("semctl");
    ...
}
```


UNIX Semaphor (Forts.)

Nachbildung von P und V

```
void P () {
    int ok;

    sema.sem_op = -1;
    ok = semop(mutex, &sema, 1);

    assert(ok != -1);
}
```

```
void V () {
    int ok;

    sema.sem_op = 1;
    ok = semop(mutex, &sema, 1);

    assert(ok != -1);
}
```

Scheitern von P/V sieht die klassische Definition [54] nicht vor:

- durch Zusicherung (engl. *assertion*) wird Gelingen „garantiert"
  - gelingt `semop()`, kehrt die aufrufende Funktion zurück
  - gelingt `semop()` nicht, wird das Programm abgebrochen
- Achtung: `-DNDEBUG` bzw. `#define NDEBUG` stellen Zusicherungen ab

Kommunikation durch Nachrichten

Motivation zum Botschaftenaustausch (engl. *message passing*)

Konsequenz der physikalischen Adressraumtrennung durch eine MMU:

- in Ausführung befindliche Programme sind abgeschottet
  - Prozesse sind in (log./virt.) Adressräumen eingesperrte „Gefangene“
  - sie können nicht ohne weiteres mit der „Außenwelt“ kommunizieren
- Kooperation muss Adressraumgrenzen überwinden können

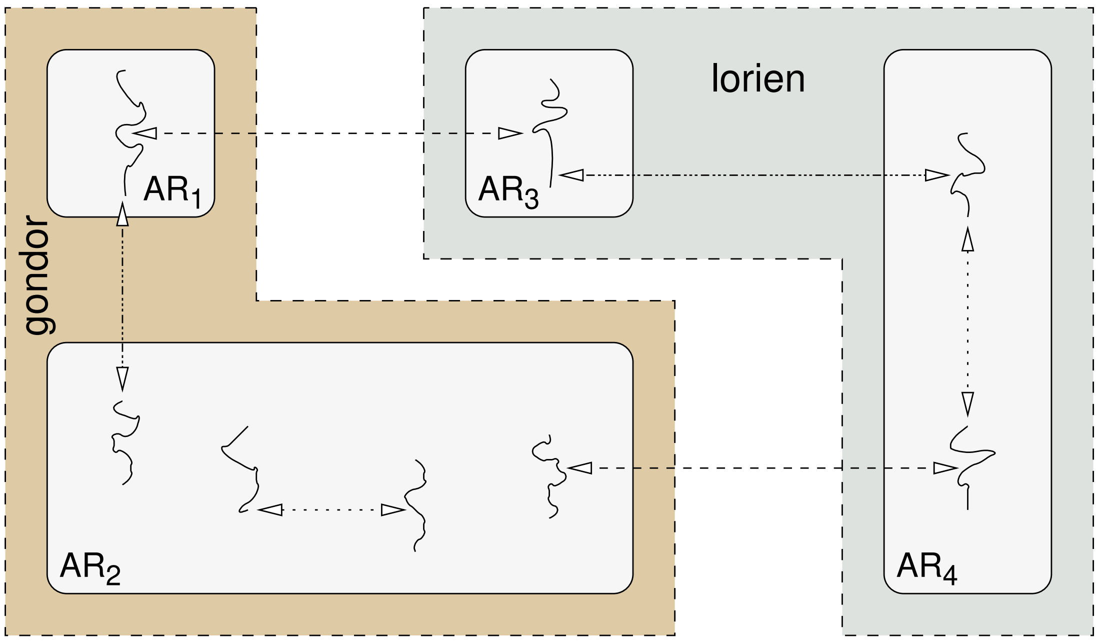
Konsequenz mehrerer Ausführungskontexte innerhalb eines Adressraums:

- Programme werden ggf. mehrfädig (engl. *multi-threaded*) ausgeführt
  - Fäden (engl. *threads*) sind eigene Kontrollflüsse im Programm
  - sie können nicht ohne weiteres mit anderen Fäden kommunizieren
- Kooperation muss Kontrollflussgrenzen überwinden können

Ein Semaphor eignet sich zur Anzeige des Ereignisses, dass Daten den einen Prozess verlassen haben und bei einem anderen Prozess eingetroffen sind. Den Datenaustausch selbst bewerkstelligt ein Semaphor nicht.

Problemdomänen der Kommunikation

Notwendigkeit domänenspezifischer Kommunikationsmechanismen



Botschaftenaustausch zwischen Prozessen

Prinzipielle Aktionen

Datentransfer vom Sende- zum Empfangsadressraum

- über einen den Prozessen gemeinsamen Kommunikationskanal

Synchronisation von Sende- und Empfangsprozess

- der Fortschritt des Empfangsprozesses hängt ab vom Sendeprozess
  - die Nachricht ist ein konsumierbares Betriebsmittel
  - der Empfangsprozess ist Konsument, der Sendeprozess ist Produzent
  - konsumiert werden kann nur, nachdem produziert worden ist
- der Fortschritt des Sendeprozesses hängt ab vom Empfangsprozess
  - der Nachrichtenpuffer ist ein wiederverwendbares Betriebsmittel
  - der Sendeprozess füllt, der Empfangsprozess leert den Puffer
  - gefüllt werden kann nur, wenn noch Platz ist ($\hookrightarrow$ leeren)
- die Koordination geschieht implizit mit der angewandten Primitive

Kommunikationssemantiken

Primitiven zum Botschaftenaustausch [56]

Sendeprimitiven wirken unterschiedlich auf den ausführenden Prozess, je nach Grad der Synchronisation mit dem Empfangsprozess:

no-wait send Sendeprozess wartet, bis die Nachricht im Transportsystem zum Absenden bereitgestellt worden ist

- Interprozesskommunikation im Vorübergehen (durch Pufferung)

synchronization send Sendeprozess wartet, bis die Nachricht vom Empfangsprozess angenommen worden ist

- Rendezvous zwischen Sender und Empfänger (ohne Pufferung)

remote-invocation send Sendeprozess wartet, bis die Nachricht vom Empfangsprozess verarbeitet und beantwortet worden ist

- Fernaufruf einer vom Empfangsprozess auszuführenden Funktion

Empfangsprimitiven wirken (im Regelfall) gleich auf den ausführenden Prozess: er wartet, bis eine Nachricht von einem Sendeprozess eintrifft

Kommunikationsmodelle

Rollenspiele bei der Interprozesskommunikation

Gleichberechtigte Kommunikation

Die miteinander kommunizierenden Prozesse spielen dieselbe Rolle; zwei Kommunikationspartner, P_1/P_2, sind sowohl Sender als auch Empfänger:

$$P_1 \left\{ \begin{array}{ccc} \textit{send} & \longrightarrow & \textit{receive} \\ \textit{receive} & \longleftarrow & \textit{send} \end{array} \right\} P_2$$

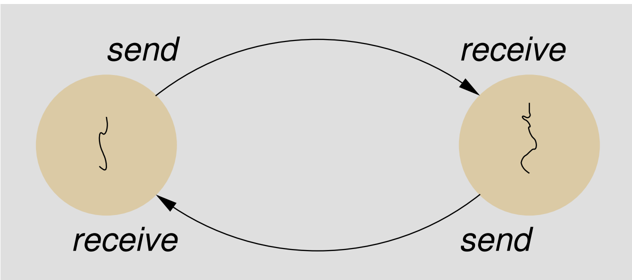
Ungleichberechtigte Kommunikation

Die miteinander kommunizierenden Prozesse spielen verschiedene Rollen; ein Kommunikationspartner, P_2, ist Dienstgeber (engl. *server*), der andere, P_1, ist Dienstnehmer (engl. *client*):

$$(\text{Klient})\ P_1 \left\{ \begin{array}{ccc} \textit{send} & \begin{array}{c}\longrightarrow \\ \longleftarrow\end{array} & \begin{array}{c}\textit{receive} \\ \textit{reply}\end{array} \end{array} \right\} P_2\ (\text{Anbieter})$$

Gleichberechtigte Kommunikation

no-wait send oder *synchronization send*



Die an der Kommunikation beteiligten Prozesse sind in ihrer Rollenfunktion gleichzeitig Produzent und Konsument von Nachrichten.

send Bereitstellung eines konsumierbaren Betriebsmittels
- *in* Identifikation des Empfängers (Konsument)
- *in* Basis/Länge der Nachricht

receive Anforderung eines konsumierbaren Betriebsmittels
- *in* Basis/Länge eines Empfangspuffers
- *out* Identifikation des Senders (Produzent)

Gleichberechtigte Kommunikation (Forts.)

Gegenseitige Abhängigkeit der Botschaften austauschenden Prozesse

```
void ibm () {
    char buf[SBUFSZ];
    pid_t from, peer = lookup("hal");

    ipc_send(peer, "?", 1);
    from = ipc_receive(buf, sizeof(buf));
    if (from == peer) printf("%s\n", buf);
}
```

```
void hal () {
    char buf[RBUFSZ];
    pid_t peer;

    peer = ipc_receive(buf, sizeof(buf));
    if (peer && (buf[0] == '?'))
        ipc_send(peer, "42", 3);
}
```

Sender „`ibm`“ muss die Adresse des Empfängers ermitteln, um ihm eine Nachricht zustellen zu können:

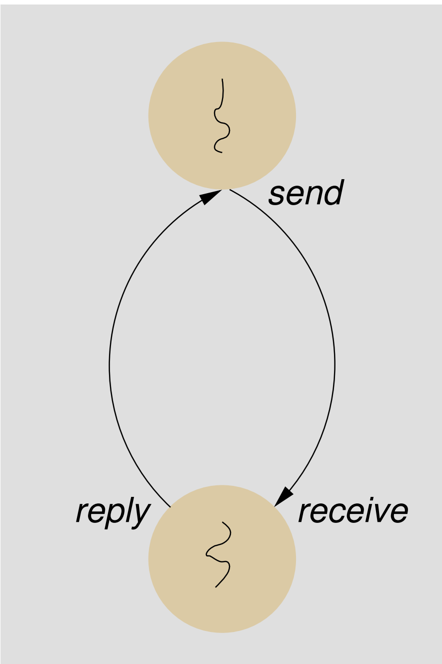
- Namensdienst (engl. *name service*)
- `lookup()` liefert die PID von „`hal`“

Empfänger „`hal`“ erhält die Adresse des Senders implizit beim Empfang der Nachricht

Ungleichberechtigte Kommunikation

remote-invocation send

Die an der Kommunikation beteiligten Prozesse besitzen unterschiedliche Rollenfunktionen, Klient einerseits und Anbieter andererseits:



send einer Anforderungsnachricht
- *in* Identifikation des Anbieters
- *in* Basis/Länge der Nachricht
- *in* Basis/Länge eines Empfangspuffers
- *out* Identifikation eines Anbieters

receive einer Anforderungsnachricht
- *in* Basis/Länge eines Empfangspuffers
- *out* Identifikation des Klienten

reply einer Antwortnachricht
- *in* Identifikation des Klienten
- *in* Basis/Länge der Nachricht

Ungleichberechtigte Kommunikation (Forts.)

Anbieter sind unabhängig von der Empfangsbereitschaft der Klienten

```
void ibm () {
    char buf[SBUFSZ];
    pid_t peer = lookup("hal");

    if (send(peer, "?", 1, buf, sizeof(buf)) == peer)
        printf("%s\n", buf);
}
```

```
void hal () {
    char buf[RBUFSZ];
    pid_t peer;

    peer = receive(buf, sizeof(buf));
    if (peer && (buf[0] == '?'))
        reply(peer, "42", 3);
}
```

Klient „`ibm`“ ruft Dienst ab und erwartet Antwort (`send()`)

Anbieter „`hal`“ erbringt Dienst (`receive()`) und gibt Antwort (`reply()`)

Senke der Interprozesskommunikation

Adressierung des Kommunikationspartners — direkt vs. indirekt

Faden (engl. *thread*) Konsument der Nachricht
- direkte Adresse der die Nachricht verarbeitenden Instanz
- die Prozessidentifikation (PID)

Tor (engl. *port*) Anschluss zur Weiterleitung/Zustellung von Nachrichten, der einem bestimmten Prozess zugeordnet ist
- Prozesse können mehrere solcher Anschlüsse besitzen
  - Ein- und/oder Ausgangstore für Nachrichten
- die Zuordnung ist statisch oder dynamisch

Briefkasten (engl. *mailbox*) Zwischenspeicher für Nachrichten, der durch Senden gefüllt und Empfangen geleert wird
- der Pufferbereich ist keinem Prozess zugeordnet
- *N* Prozesse können dahin senden und daraus empfangen

Kommunikation und Betriebsmittel

Synchrone vs. asynchrone Interprozesskommunikation

Prozesse synchronisieren sich zur Kommunikation, indem sie Betriebsmittel anfordern und bereitstellen (S. V-89):

Sender benötigt das wiederverwendbare Betriebsmittel „Puffer“

- synchrone IPC $\Rightarrow$ der Zielpuffer (des Empfängers)
- asynchrone IPC $\Rightarrow$ ein Zwischenpuffer

Empfänger benötigt das konsumierbare Betriebsmittel „Nachricht“

- asynchrone IPC $\Rightarrow$ ein Zwischenpuffer
- synchrone IPC $\Rightarrow$ der Quellpuffer (des Senders)

Betriebsmittelmangel ist die Ursache dafür, dass Prozesse bei der Kommunikation ggf. blockieren werden:

- Empfänger erwartet Nachricht, Sender erwartet freien Puffer
- „asynchron“ bedeutet nicht „nicht-blockierend“ oder „wartefrei“

Verbindungen zwischen kommunizierenden Prozessen

Gütemerkmale (engl. *quality of service*) garantieren

IPC verwendet (in dem Fall) Torverbindungen und verläuft in drei Phasen:

Aufbauphase plant die zur Durchsetzung der jeweils angeforderten Gütemerkmale notwendigen Betriebsmittel ein

- Puffer, Fäden, Bandbreite, . . . , Protokoll

Nutzungsphase Botschaftenaustausch gemäß Gütemerkmale

Abbauphase gibt reservierte Betriebsmittel frei, löst die Verbindung auf

Betriebsarten der durch Torverbindungen entstehenden Verbindungskanäle bestimmen u.a. die möglichen Richtungen des Datentransfers:

unidirektional	Tor_s	$\longrightarrow$	Tor_r	halbduplex
bidirektional	Tor_{sr}	$\longleftrightarrow$	Tor_{rs}	vollduplex

UNIX Systemfunktionen

Linux, MacOS, SunOS

```
s   = socket(domain, type, protocol)
ok  = bind(s, name, namelen)
num = recvfrom(s, buf, buflen, flags, from, fromlen)
num = sendto(s, msg, msglen, flags, to, tolen)
ok  = connect(s, name, namelen)
ok  = listen(s, backlog)
d   = accept(s, addr, addrlen)
num = recv(d, buf, buflen, flags)
num = send(s, msg, msglen, flags)
ptr = gethostbyname(name)
    ⋮
```


Botschaftenaustausch mit UNIX Systemfunktionen

Sendeprimitive (passend zu S. V-93) — mit Bedacht zu verwenden

- jede Aktivierung fordert *Socket* und Puffer an: Betriebsmittel sind nicht garantiert

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include "./message.h"
#include "./pid2rid.h"

int ipc_send (pid_t peer, char *data, unsigned size) {
    int ok, s, nbytes;
    struct sockaddr_un addr;
    message *msg;
    char name[RIDSZ];

    if ((ok = s = socket(AF_UNIX, SOCK_DGRAM, 0)) != -1) {
        if ((ok = (msg = (message*)malloc(nbytes = sizeof(message) + size)) != 0)) {
            msg->ipc_from = getpid(); bcopy(data, msg->ipc_data, size);

            addr.sun_len = 0; addr.sun_family = AF_UNIX;
            bcopy(pid2rid(peer, name), addr.sun_path, RIDSZ);

            ok = sendto(s, msg, nbytes, 0, (struct sockaddr *)&addr, sizeof(addr));
            free(msg);
        }
        close(s);
    }
    return ok;
}
```

Botschaftenaustausch mit UNIX Systemfunktionen (Forts.)

Empfangsprimitive (passend zu S. V-93) — mit Bedacht zu verwenden

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include "./message.h"
#include "./pid2rid.h"

pid_t ipc_receive (char *data, unsigned size) {
    int ok, s, nbytes, addrlen;
    message *msg;
    char name[RIDSZ];
    struct sockaddr_un addr;

    if ((ok = s = socket(AF_UNIX, SOCK_DGRAM, 0)) != -1) {
        if ((ok = (msg = (message*)malloc(nbytes = (sizeof(message) + size))) != 0)) {
            addr.sun_len = 0; addr.sun_family = AF_UNIX;
            bcopy(pid2rid(getpid(), name), addr.sun_path, RIDSZ);

            if ((ok = bind(s, (struct sockaddr *)&addr, sizeof(addr))) != -1) {
                addrlen = sizeof(addr);
                if ((ok = recvfrom(s, msg, nbytes, 0, (struct sockaddr *)&addr, &addrlen)) != -1) {
                    ok = msg->ipc_from; bcopy(msg->ipc_data, data, size);
                }
                unlink(name);
            }
            free(msg);
        }
        close(s);
    }
    return ok;
}
```

- jede Aktivierung fordert *Socket* und Puffer an: Betriebsmittel sind nicht garantiert
- nur für die Dauer von `recvfrom(2)` ist die (lokale) PID des Empfängers an einen (globalen) Namen gebunden: Sendeversuche können scheitern

Botschaftenaustausch mit UNIX Systemfunktionen (Forts.)

Protokolldateneinheit (engl. *protocol data unit*, PDU) und Namensfunktion

Nachricht als Dienstdateneinheit (engl. *service data unit*, SDU), die in einer PDU vom Sender zum Empfänger transportiert wird:

```
typedef struct ipc_message {
    pid_t ipc_from;
    char  ipc_data[0];
} message;
```

Schablone mit der PID des Senders und einem generischen Feld zur Aufnahme der SDU.

Kommunikationsendpunkt ist eine Prozessinstanz, aus deren PID eine symbolische Adresse (*receiver ID*, RID) generiert wird:

```
#define RIDSZ (int)((sizeof(pid_t) * 2.5) + 3)

inline char* pid2rid (pid_t pid, char rid[]) {
    sprintf(rid, "rid%u", pid);
    return rid;
}
```

Die RID wird im UNIX Namensraum abgelegt (`bind(2)`).

UNIX Kommunikationsfunktionen „*Considered Harmful*“

Typisch „Allgemeinzweck“ — und dennoch problematisch für IPC

Pros

- Datenstromkonzept
  - *no-wait send* Semantik, kanalorientiert
- in E/A-Schnittstelle integriert
  - `read(2)` empfängt, `write(2)` sendet
  - `close(2)` gibt *Socket* frei, `unlink(2)` entfernt *Socket*-Verknüpfung
- verschiedenste Protokolle und Betriebsarten

Cons

- rein asynchrones Modell
  - *synchronization/remote-invocation send* fehlt
- sehr komplexe Schnittstelle
  - knifflige, kostspielige problemorientierte Lösungen auf Bibliotheksebene
  - synchrone IPC nur sehr aufwändig nachbildbar
- man muss mit Kanonen auf Spatzen schießen. . .

☞ ganz gut, um schwergewichtige Klient/Anbieter-Systeme zu bauen. . .

UNIX Dienstgeber

Anbieter (engl. *server*)

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

void fail (char* msg) { perror(msg); exit(0); }

int main (int argc, char *argv[]) {
    struct sockaddr_in addr;
    int sockfd, fd, addrlen = sizeof(addr);
    char buffer[256];

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) fail("socket");

    bzero((char *)&addr, sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = INADDR_ANY;
    addr.sin_port = htons(atoi(argv[1]));
    if (bind(sockfd, (struct sockaddr *)&addr, sizeof(addr)) < 0) fail("bind");

    if (listen(sockfd, 5) < 0) fail("listen");
    for (;;) {
        bzero(buffer, 256);
        if ((fd = accept(sockfd, (struct sockaddr *)&addr, &addrlen)) < 0) fail("accept");
        if (read(fd, buffer, 256) < 0) fail("read");
        close(fd);
        printf("%s\n", buffer);
    }
}
```

Betriebsmittel für die Lebendsauer des Dienstgebers, um ein Scheitern des Sendens bei hoher Anfragefrequenz zu vermeiden:

- Anbieter-*Socket* (`socket(2)`),
- *Socket*-Verknüpfung (`bind(2)`)

(vergl. S. V-101).

UNIX Dienstnehmer

Klient (engl. *client*)

Klient/Anbieter-Systeme sind mit einem mehr auf IPC und weniger auf Datenstromkommunikation ausgerichteten Konzept besser bedient:

- *remote-invocation send* ist ein Muss [57],
- ein Mikrokern [58, 59] ist vorteilhaft

(vergl. S. V-95).

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

void fail (char *msg) { perror(msg); exit(0); }

int main(int argc, char *argv[]) {
    int sockfd;
    struct sockaddr_in addr;
    struct hostent *server;

    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) fail("socket");

    bzero((char *)&addr, sizeof(addr));
    addr.sin_family = AF_INET;
    if ((server = gethostbyname(argv[2])))
        bcopy((char *)server->h_addr, (char *)&addr.sin_addr.s_addr, server->h_length);
    addr.sin_port = htons(atoi(argv[1]));
    if (connect(sockfd, (struct sockaddr*)&addr, sizeof(addr)) < 0) fail("connect");

    if (write(sockfd, argv[3], strlen(argv[3])) < 0) fail("write");
}
```


UNIX Dienstgeber/-nehmer

Int{ra,er}rechner-Interprozesskommunikation

Anbieter: „printf()-Server“

```
wosch@gondor 85$ gcc -O6 server.c -o server
wosch@gondor 86$ ./server 4711
Die Anwort auf alle Fragen ist
42
```

Klient 1

```
wosch@gondor 11$ gcc -O6 client.c -o client
wosch@gondor 12$ ./client 4711 gondor 42
wosch@gondor 13$
```

Klient 2

```
wosch@lorien 62$ gcc -O6 client.c -o client
wosch@lorien 63$ ./client 4711 gondor “Die Antwort auf alle Fragen ist“
wosch@lorien 64$
```

Resümee eines Betriebssystemüberblicks

Grundlegende Funktionen im Schnelldurchlauf

- Betriebssysteme bieten eine „Hand voll" nützlicher Abstraktionen
  - Adressräume, Speicher, Dateien, Namensräume
  - Prozesse, Koordinationsmittel
- von zentraler Bedeutung ist die Abbildung von Namen auf Adressen
  - symbolische $\mapsto$ numerische $\mapsto$ logische $\mapsto$ virtuelle $\mapsto$ physikalische
  - das Konzept findet im Kontext von Rechnernetzen seine Fortführung
- nicht weniger bedeutend ist die Einplanung von Prozessen
  - d.h., die Planung des zeitlichen Ablaufs der Prozessorzuteilung
  - allgemein: die Planung der Zuteilung von Betriebsmitteln an Prozesse
- jeder Prozess gehört einer bestimmten Gewichtsklasse an
  - schwer-, leicht- oder federgewichtiger Prozess
  - u.a. eine Frage der Verzahnung von Prozessinstanz und Adressraum
- Mehrprozessbetrieb erfordert die Koordination von Prozessen