Chair of Connected Mobility
Department of Informatics
Technische Universität München

Kapitel 05

Grundlagen Betriebssysteme und Systemsoftware

Inter-Prozesskommunikation (IPC)

Prof. Dr.-Ing. Jörg Ott

Lehrstuhl Connected Mobility

Folien frei nach der Vorlage von Prof. C. Eckert (WS16/17, WS17/18) – danke!

Ziele

- Weitere Mittel zur kontrollierten Interaktion zwischen Prozessen

- Synchrone vs. asynchrone Kommunikation
- Nachrichtenbasierte Kommunikation
- Stromorientierte Kommunikation

- Signale
- Shared Memory
- Pipes
- Sockets

Einführung

- Thema: (Inter-)Prozesskommunikation
- Kommunikation ist eigentlich Thema der Vorlesung „Grundlagen Rechnernetze und Verteilte Systeme“ (GRNVS)
- Wir konzentrieren und auf die für Betriebssysteme bzw. Betriebssystementwicklung relevanten Aspekte besprochen
- Einige Konzepte werden auch zur Kommunikation über mehrere Computersysteme hinweg verwendet

Bisher

- Prozesse mit getrennten Adressräumen
  - Wissen über Prozess-Ids nur innerhalb von Prozessgruppen
  - Elternprozess ↔ Kindprozesse
- Threads mit gemeinsamem Adressraum
- Synchronisationsmechanismen
  - zur Koordination beim Zugriff auf gemeinsame Ressourcen
  - Semaphore, Mutexe usw.
- Inter-Prozesskommunikation
  - Für wohldefinierte Interaktionen zwischen Prozessen, Threads, …
  - Dabei ist natürlich Synchronisation erforderlich!

Kommunikationsformen

- Arten der Kommunikation zwischen Prozessen bzw. Threads
  - Signalisierung von Ereignissen
  - Austausch von Nachrichten
    - Auch komplexe Kommunikationsprotokolle möglich (z.B. TCP, UDP)
  - Gemeinsamer Speicher

- Eigenschaften von Interprozesskommunikation (u.a.)
  - Breitbandig vs schmalbandig
  - Implizit vs explizt
  - Nachrichtenorientiert vs speicherorientiert
  - Antwortverhalten: Nachricht vs Auftrag

- Meta-Problem: Rendezvous
  - Woher weiß ein Prozess, mit welchem anderen er kommunizieren soll?
  - Wie findet ein Prozess seine(n) Kommunikationspartner?

Bandbreite des Kommunikationskanals

- Schmalbandige Kanäle
  - Übertragen von wenigen Bits an Information
  - Melden von Ereignissen
  - Synchronisationskonzepte und Unterbrechungskonzepte erforderlich
- Beispiel unter Linux: Signals
  - Prozesse können sich gegensseitig Signale senden (`kill()`)
  - Prozesse können diese Signale explizit abfangen (`signal()`)
    - Durch registrieren eines Signal-Handlers
    - Durch explizites Ignorieren (SIG_IGN)
  - In Posix 31 unterschiedliche Signale möglich
    - z.B. SIGINT, SIGTERM, SIGKILL, SIGSEGV, SIGILL, ... (`man 7 signal`)
  - Wenn der Prozess das Signal nicht abgefängt, wird er vom Betriebssystem beendet
  - Die Signale SIGKILL und SIGSTOP können nicht abgefangen werden
  - Achtung: Signale unter Linux unterbrechen einen Prozess asynchron
    - Der Prozess muss mit solchen Unterbrechungen umgehen können

Beispiel

```
void sighandler (int sig) {
    printf ("Caught signal %d\n", sig);
    signal (SIGINT, sighandler);
    signal (SIGALRM, sighandler);
}

int main (int argc, char *argv [], char *envp []) {
    char    buffer [1024];
    int     len;

    signal (SIGINT, sighandler);
    signal (SIGALRM, sighandler);
    alarm (5);
    for (len = 0; len < 10; len++)
          printf ("Counting %d...\n", len), sleep (1);
    alarm (10);
    while (1) {
          len = read (0, buffer, sizeof (buffer) - 1);
          if (len == -1) {
              perror ("read () failed");
              continue;
          }
          if (len == 0) {
              printf ("Exiting\n");
              exit (0);
          }
          buffer [len] = '\0';
          if (!strncmp (buffer, "exit", 4))
              exit (0);
          write (1, buffer, strlen (buffer));
    }
}
```

```
$ ./signal
Counting 0...
Counting 1...
Counting 2...
Counting 3...
Counting 4...
Caught signal 14
Counting 5...
Counting 6...
Counting 7...
Counting 8...
Counting 9...
hello, world
hello, world
Caught signal 14
echo this
echo this
^CCaught signal 2
exit
$
```

Bandbreite des Kommunikationskanals

- Breitbandige Kanäle
  - Übertragung größerer Datenmengen
  - Unterscheidung nach:
    - impliziter Übertragung (z.B. Gemeinsamer Speicher)
    - expliziter Übertragung (z.B. Pipes, Sockets (AF_INET))
  - Hierbei auch Konzepte aus dem Bereich der verteilten Systeme relevant
    - z.B. Omega-Switching-Netze oder verschiedene Vernetzungsarchitekturen (Ring, Stern)
    - Allgemeine, teilstrukturierte Netze wie das Internet
    - Werden über die gleichen Betriebssystemfunktionen realisiert

- Im folgenden werden einige Beispiele für breitbandige Kanäle betrachtet

Implizite Kommunikation

- Implizite Kommunikation über gemeinsame Betriebsmittel
  - keine direkte Unterstützung durch das Betriebssystems
  - Vorteil: Einfach und schnell, da kein Kopieren zwischen Adressräumen benötigt wird
  - Aber:
    - Gemeinsame Bereiche sind nicht immer vorhanden.
    - Aufwendiges busy-waiting kann auftreten.
    - Weitere (schmalbandige) Synchronisation ist notwendig (siehe Kapitel 3)
  - Beispiele
    - Speicher, Register, Dateien, Ringpuffer
    - Queue-Datenstruktur im Prozess, welche von allen Threads verwendet wird
  - Speicher der Datenstruktur kann auch in mehrere Prozesse gemappt sein
    - Hier ist Unterstützung durch das Betriebssystem erforderlich
    - Eintrag in Pagetables (siehe Kapitel 6)

Beispiel: Gemeinsamer Speicher (1/2)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>
#include <signal.h>

int     message = 0;

void child_handler (int sig) {
    message = 1;
}

int main (int argc, char *argv [], char *envp []) {
    char    *shmem;
    pid_t   pid;

    if ((shmem = mmap (NULL, 1024, PROT_READ|PROT_WRITE, MAP_ANONYMOUS|MAP_SHARED, 0, 0))
                == NULL)
        perror ("Cannot create shared memory");
        exit (-1);
    }
    // set up the signal handler before forking, so it surely is in place
    signal (SIGUSR1, child_handler);

    ...
```

Beispiel: Gemeinsamer Speicher (2/2)

```
    switch (pid = fork ()) {
    case -1:    /* error */
         perror ("fork () failed");
         exit (-1);

    case 0:         /* child process */
         printf ("child %d: waiting for message...\n", getpid ());
         while (!message)
             usleep (1000);
         printf ("child %d reading from shmem: %s\n", getpid (), shmem);
         sprintf (shmem, "Got it [%d] -- exiting", getppid ());
         printf ("child %d writing to shmem: %s\n", getpid (), shmem);
         break;

    default:    /* parent process */
         signal (SIGUSR1, SIG_IGN);
         sprintf (shmem, "Parent %d sending message to child %d", getpid (), pid);
         kill (pid, SIGUSR1);  /* notify the child process */
         wait (NULL);          /* wait for the child to terminate */
         printf ("Message from child %d: %s\n", pid, shmem);
         break;
    }
    exit (0);
}
```

Explizite Kommunikation

- Versenden und Empfangen von Nachrichten: message passing
  - Geeignet für Prozesse in disjunkten Adressräumen
  - Dedizierte Interaktion mit dem Betriebssystem (Beispiel Linux)
    - `send (int sockfd, const char* buf, ...)`
    - `recv (int sockfd, const char* buf, ...)`
  - Funktionssignaturen sind abhängig vom Betriebssystem, aber in POSIX standardisiert
  - Typische Signatur: (Sender/Empfänger, Nachricht)
  - Kommunikation sowohl lokal (z.B. PID) als auch mit entfernten Partnern (z.B. IP Adresse, Portnummer) möglich
  - Auch Verwendung von `read()/write()` möglich, da Sockets auch Filedeskriptoren sind (siehe Kapitel 9)

Explizite Kommunikation

- Nachrichtenaufbau
  - Nachrichtenkopf (header): Managementinformationen
    - u.a. Sender-, Empfänger-Identifikation, Größe usw.
  - Nachrichtenkörper enthält die Nutzlast (payload).

- Ablauf
  1. Prozess sendet mit Hilfe von `send()` die Nachricht.
  2. Nachrichtendienst des Betriebssystems übermittelt die Nachricht
  3. Empfänger Prozess empfängt Nachricht mit Hilfe von `recv()`

- Unterscheidung in synchrone und asynchrone Kommunikation

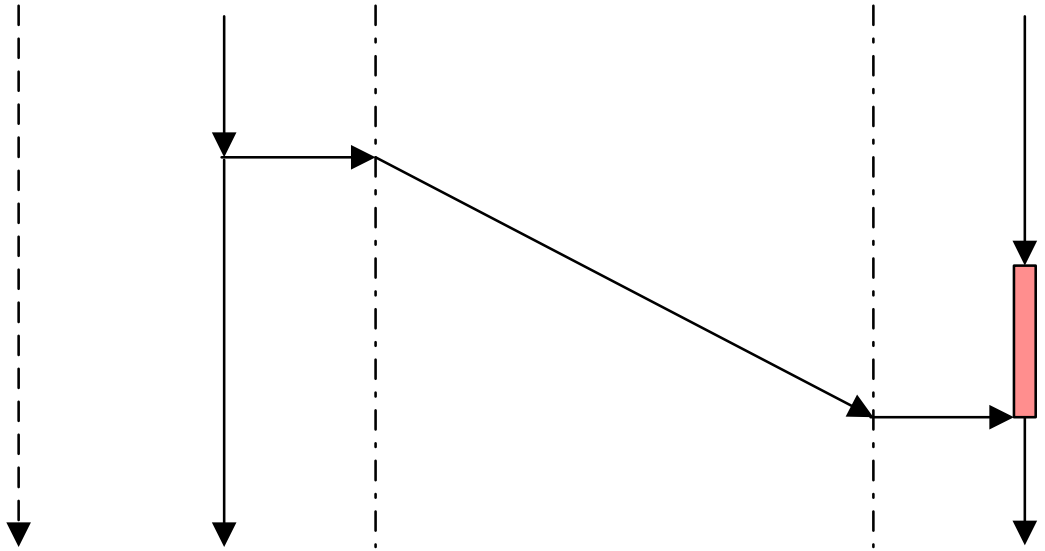
Synchron vs. Asynchron

- Kopplungsgrad zwischen Prozessen
  - synchron: beide Prozesse werden zur Nachrichtenübertragung synchronisiert, blockierend
  - asynchron: Entkopplung von Sender und Empfänger, nicht blockierend

- Muster der Nachrichtenkommunikation
  - Meldung (Signal, nicht mit Linux-Signals verwechseln!)
    - Einzelne Nachricht vom Sender zum Empfänger (unidirektional)
    - Wenige Daten werden übertragen.
    - Beispiel: Mitteilung einer Zustandsänderung (Datei fertig gedruckt)
    - Aber: häufig Empfangsbestätigung erforderlich, Acknowledgment (Ack)
  - Auftrag (Request – Response)
    - Interaktion zwischen Sender und Empfänger (bidirektional)
    - Beispiel: Anfrage nach bestimmten Daten. Antwort enthält geforderte Daten

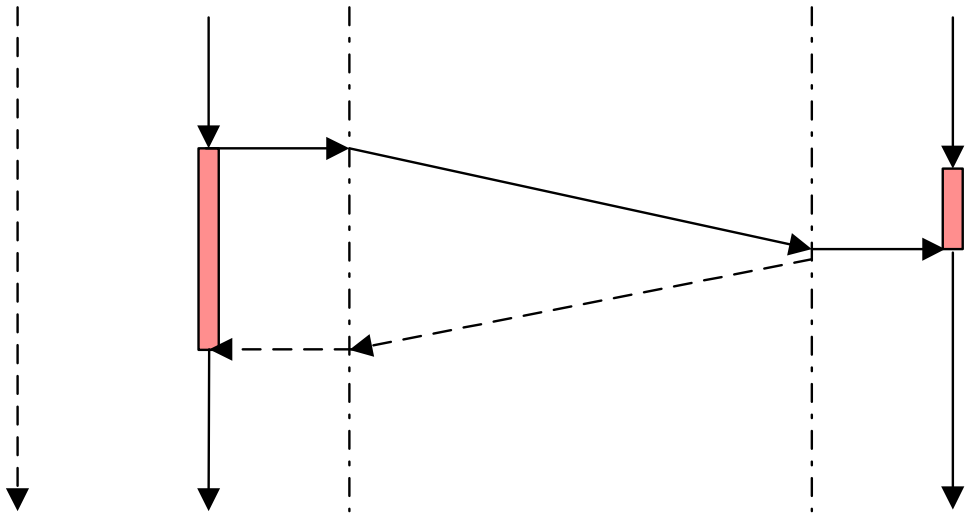
Asynchrone Meldung

- Sender (S) übergibt die Nachricht an den Nachrichtendienst des Betriebssystems (`send()`)
- BS puffert Nachrichten
  - Asynchron: S wartet nicht, bis der Empfänger die Nachricht empfangen hat
- Empfänger (E) kann Nachricht mittels `recv()` empfangen
  - Falls keine Nachricht gepuffert ist wird der Empfänger blockiert, bis eine Nachricht gesendet wird



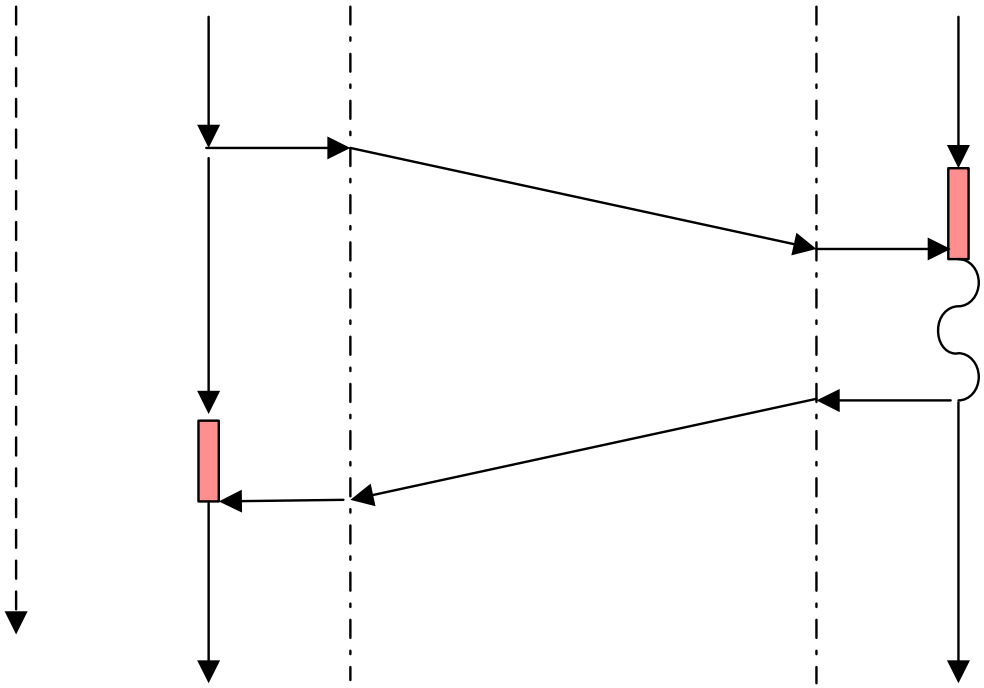
Synchrone Meldung

- Der Empfänger sendet nach Erhalt der Nachricht eine Bestätigung
- Der Sender wartet nach dem Senden auf die Empfangsbestätigung
- Die Empfangsbestätigung enthält hier keine Daten sondern dient nur der Synchronisation
- Alternativ: Rendezvous-Verfahren
  - Sender und Empfänger stellen vor Austausch der Meldung die Sende- und Empfangsbereitschaft her
  - Die Nachricht muss nicht gepuffert werden



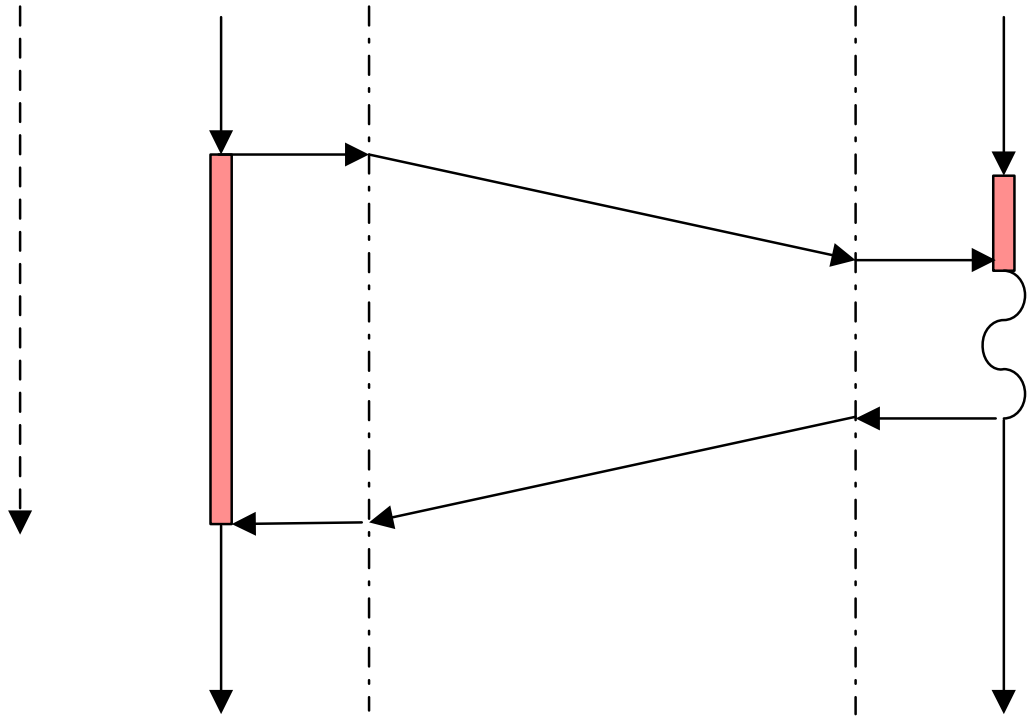
Asynchroner Auftrag

- Auftrag und Resultat werden als unabhängige Meldungen verschickt
- Zwischen `send()` und `recv()` kann der Sender weitere Aufträge versenden
- Die Aufträge können an den gleichen oder andere Empfänger gesendet werden



Synchroner Auftrag

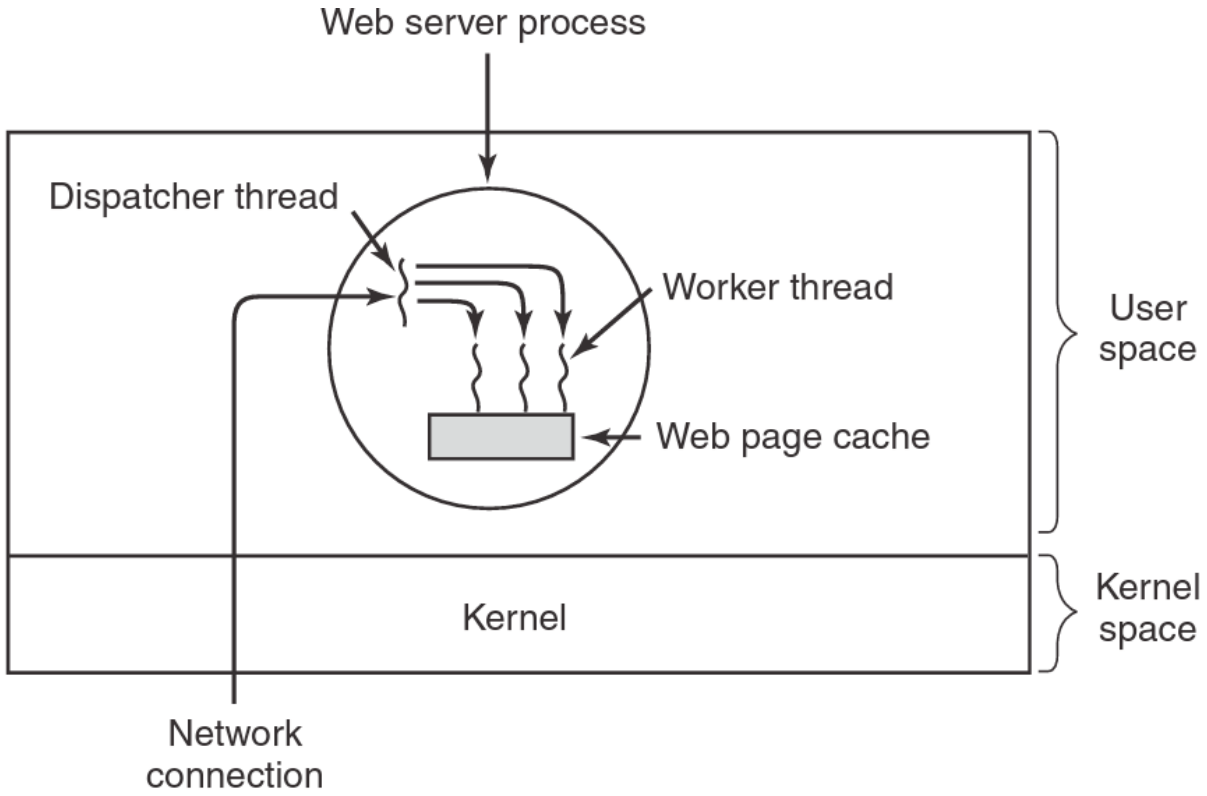
- Nachrichtenbearbeitung und Senden des Resultats sind Teil der Transaktion
- Zum Beispiel Aufruf einer Webseite



Bewertung synchron vs. asynchron

- Vorteile von asynchronem Senden:
  - Nützlich für Echtzeitanwendungen, wenn sendender Prozess nicht blockiert werden darf
  - Ermöglicht parallele Abarbeitung durch Sender und Empfänger
  - Anwendbar zum Signalisieren von Ereignissen
- Nachteile
  - Verwaltungsoverhead im Betriebssystem (Puffer für Nachrichten)
  - Behandlung von Fehlern schwieriger
    - Keine direkte Benachrichtigung des Senders möglich
    - Paketverlust (durch volle Puffer, insbesondere bei Netzkommunikation)
    - Wiederholung von Paketen
- Typischerweise wird asynchrone Kommunikation verwendet
  - Vor allem, wenn unklar ist, ob Empfänger erreichbar oder wie schnell mit einer Antwort zu rechnen ist
  - Option: Threads verwenden, um synchron und asynchron zu mischen

Beispiel



Dispatcher-Thread

```
while (TRUE) {
   get_next_request (&buf);
   handoff_request (&buf);
}
```

Worker-Threads

```
while (TRUE) {
   wait_for_work (&buf);  // block
   look_for_page_in_cache (&buf, &page);
   if (page_not_in_cache (&page));
      read_page_from_disk (&buf, &page);
   return_page (&page);
}
```

Message Passing

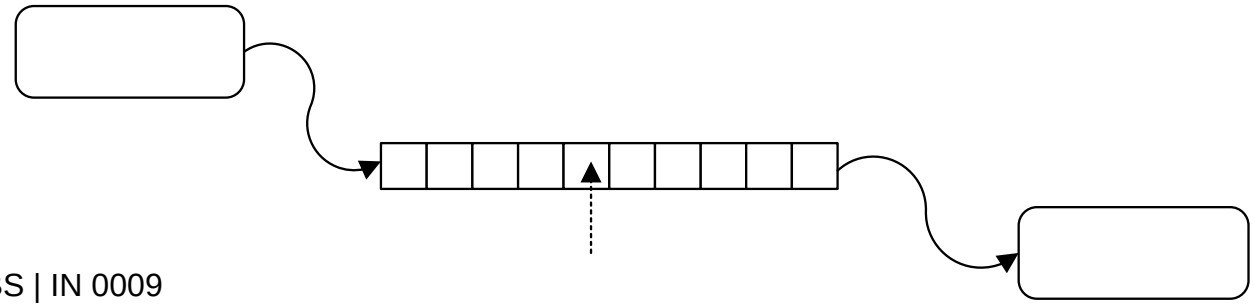
- Beispiel: Lösung des Erzeuger-Verbraucher Problems mit Hilfe von synchronem Message Passing.
  - Verwendung der Funktionen send() und recv().
  - Die Synchronisation von Erzeuger und Verbraucher erfolgt durch das Kommunikationssystem selbst, es sind keine Semaphore notwendig
  - Bemerkung: Kein gemeinsamer Speicherbereich, der bzgl. der Zugriffe von Erzeuger und Verbraucher synchronisiert werden muss

```
// Erzeuger

while (true) {
    data = produce_item ();
    send (V, data);
}
```

```
// Verbraucher

while (true) {
    recv (E, data);
    consume_item (data);
}
```

Streams

- Bisher
  - Verbindungslose Kommunikation
  - Keine Annahme über den Art des Nachrichtenversands
- Ein Strom (engl. Stream) = Abstraktion einer Verbindung
  - Nachrichten werden auf dem Kommunikationsweg gepuffert
  - Die versendeten Nachrichten werden als logischer Bytestrom vereinigt
  - Der Bytestrom kann unabhängig von den Nachrichtengrenzen verarbeiten werden
    - Achtung: wer Nachrichtengrenzen braucht, muss diese selbst hinzufügen
  - Dienste des Betriebssystems: Verbindungsauf- und -abbau, lesen und schreiben in Strom
    - Beispiel: Input/Output Streams in Java oder C++



Pipes

- Das Pipe-Konzept (z. B. in Unix, Windows) dient zur Realisierung von Strömen
  - Eine Pipe (“|”, Röhre) ist ein unidirektionaler Strom zwischen zwei Kommunikationspartnern
    - Es kann auch bidirektionale Kommunikation mit Paaren von Pipes realisiert werden
  - Eine Pipe erlaubt FIFO-artigen Datentransfer: open pipe, read, write
  - Blockieren bei voller Pipe (write) und bei leerer Pipe (read)
  - Pipes können als virtuelle Dateien realisiert sein (named pipes)
  - Unter Linux oft auf der Kommandozeile: Prozess1.stdout → Prozess2.stdin)
    - `cat <datei> | wc -l`
    - `grep “<stdio.h>” /usr/src/linux-source-4.8 -R | wc -l`

Bidirektionale Pipes

- Verwendung von Pipes:
  - Die Funktion `pipe()` erstellt zwei Kommunikationsendpunkte
  - Einer davon kann z.B. beim `fork()` an den Kindprozess übergeben werden
  - Alternativ: Verwendung in unterschiedlichen Threads

- Named Pipes
  - Für die Kommunikation zwischen mehreren Prozessen sind auch Named Pipes möglich
  - Diese tragen einen Namen und sind im Dateisystem verlinkt

```
int main (...) {
  int p [2];
  pipe (p);

  sendData (p [1]) ;
  receiveData (p [0]) ;
}
```

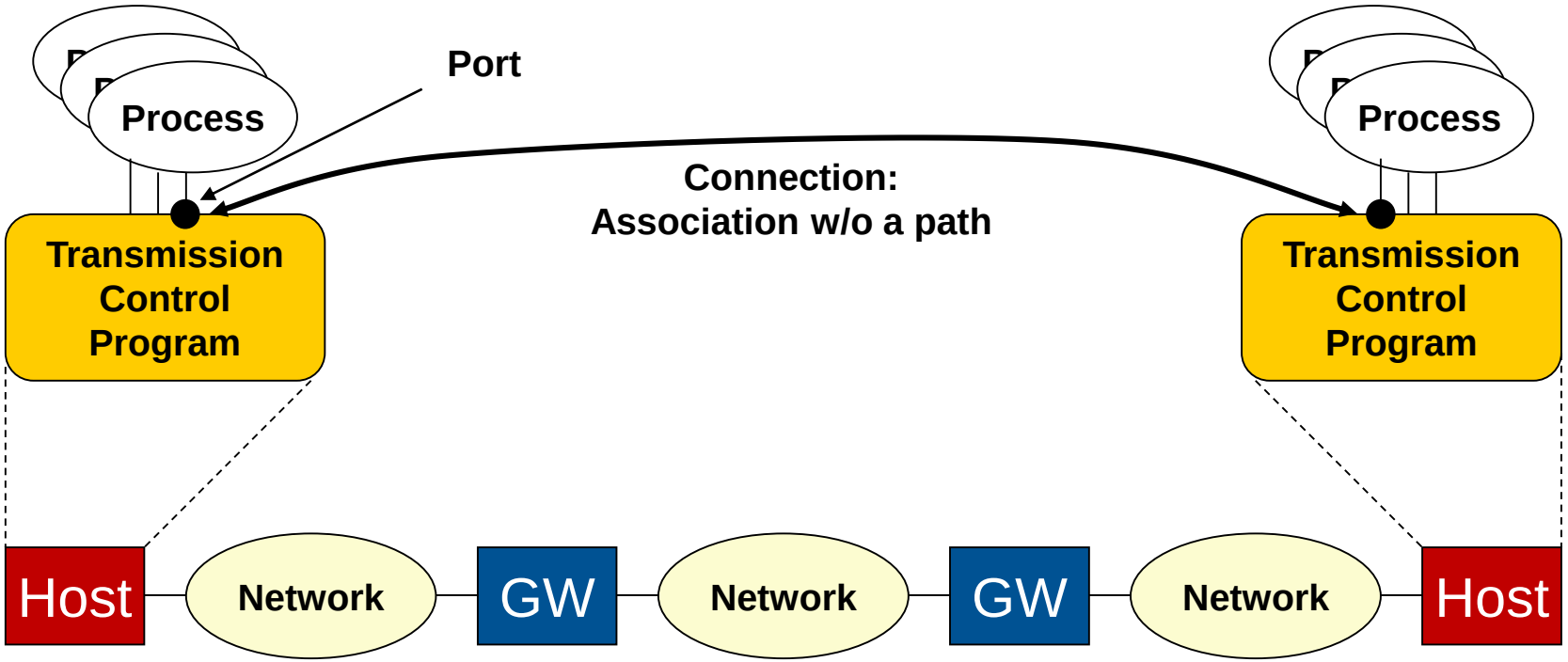
```
void sendData (int fd) {
  const char *msg = "Wello, world";
  uint16_t len = strlen (msg);
  uint16_t nLen = htons (len);

  send (fd, &nLen, sizeof (nLen));
  send (fd, msg, len);
}
```

```
void receiveData (int fd) {
  char msg [1024];
  uint16_t nLen;
  uint16_t len;

  recv (fd, &nLen, sizeof (nLen));
  len = ntohs (nLen);
  recv (fd, msg, sizeof (msg) - 1);
  msg [len] = '\0';
  printf ("\%u \%s\n", len, msg);
}
```

Adressierung: Ports

- Bisherige Annahmen
  - Feste Beziehung zwischen Sender und Empfänger über PID / Namen
  - Prozesse sollen Nachrichten im eigenen Adressraum entgegennehmen können
  - Prozesse benötigen eigene Nachrichtenpuffer für neue, nicht angenommene Nachrichten
- Problem
  - Prozessnummern ändern sich und Prozessnamen sind nicht eindeutig
  - Prozesse wollen unter Umständen mit mehreren Partnern gleichzeitig kommunizieren.
- Lösung
  - Einführung von Ports als logische Abstraktion von Kommunikationsendpunkten
  - Wir wählen Ports der Internet-Protokolle als Beispiel

Historisches: Ursprüngliche Terminologie und Architektur im Internet (1974)



Nach: Robert E. Kahn, Vinton G. Cerf: A Protocol for Packet Network Intercommunication, IEEE Trans on Comm, May 1974

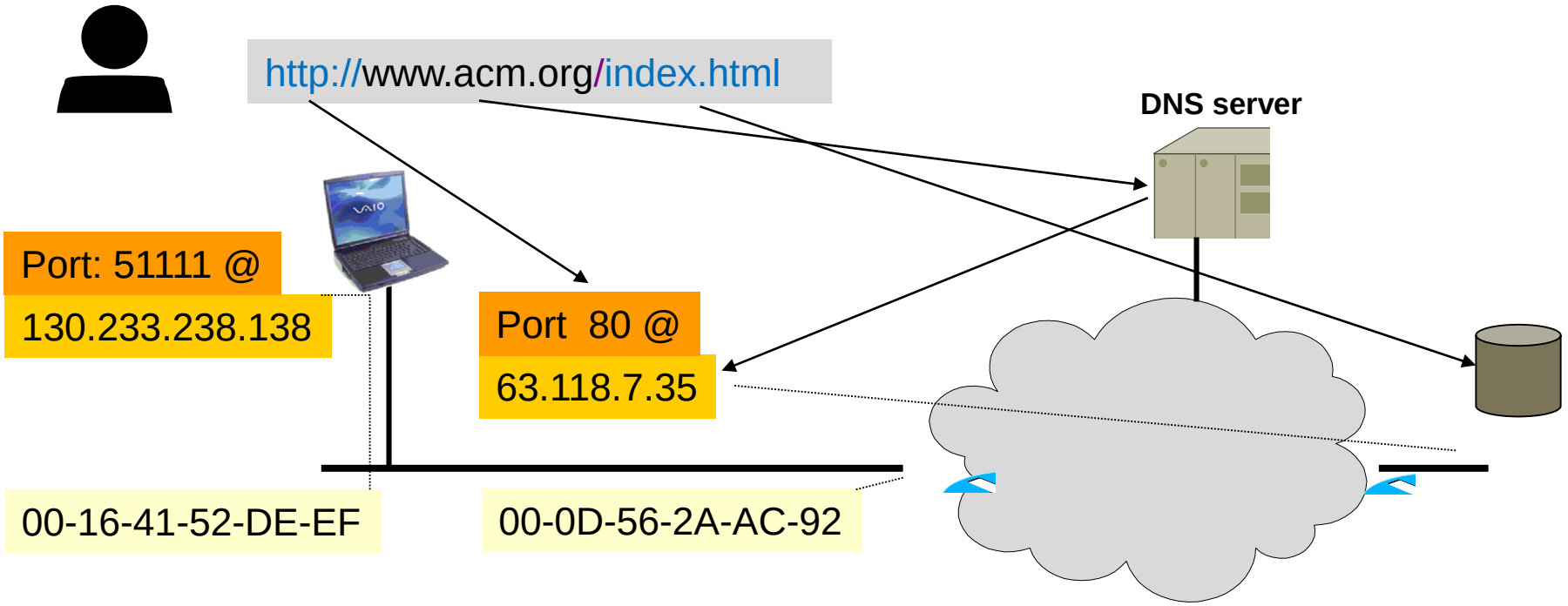
Ports

- Ein Port gehört eindeutig zu einem Prozesses
  - zu einer gegebenen Zeit
- Empfängerprozess kann senderspezifische Ports einrichten
  - Einem Port kann ein eigener Thread zugeordnet werden
- Mehrere Ports pro Prozess möglich
  - z.B. 65.535 TCP-Ports pro IP-Adresse
  - Portnummern 0 – 1023 sind fest reserviert für bestimmte Protokolle (Well-known Ports)
  - Portnummern 1024 – 49151 für registrierte Anwendungen (IANA)
  - Portnummern 49152 – 65535 dynamisch vom BS zugewiesen
    - etwa für abgehende Verbindungen

Ports (2)

- Passive Prozesse (z.B. Server)
  - Registrieren sich auf einer (oder mehrerer) Portnummer(n)
  - Warten auf eigehende Verbindungen (TCP) oder Nachrichten (UDP)
  - Ein (TCP-)Port entspricht einer FIFO-Warteschlange
    - Die Warteschlange sammelt die Verbindungswünsche
    - Die maximale Länge ist abhängig vom Betriebssystem, (z. B. 50)
    - Es werden keine weiteren Verbindungswünsche akzeptiert, wenn die Warteschlange voll ist
- Aktive Prozesse (z.B. Client)
  - Initiieren die Interaktion mit einem passiven Prozess
  - Erhalten hierfür dynamisch vom BS eine freie Portnummer
  - Auf dieser empfangen Sie dann auch Daten
- Identifikation einer Kommunikationsbeziehung durch 5-Tupel
  - Quell-IP-Adresse, Ziel-IP-Adresse, Quell-Port, Ziel-Port, Protokoll
  - Vom BS verwendet, um eingehende Nachrichten zuzuordnen

Adressen im Internet



Ports

- Beispiele

Protokoll	Port	Beschreibung
echo	7	Sendet empfangene Daten zurück
SSH	22	Secure Shell (interaktive Terminal-Sitzung)
SMTP	25	Versenden von E-Mail (zwischen Servern)
DNS	53	Auflösung von Domain-Namen
HTTP	80	Unsicheres HTTP
HTTPS	443	Sicheres HTTP
SMTPS	465	Sicheres SMTP
IMAPS	993	Protokoll zum sicheren E-Mail-Abruf

- Viele weitere well-known Ports
  - siehe /etc/services (unter Linux)
  - https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml
- Zustände von Ports: offen, geschlossen, blockiert.

Kurze Erinnerung

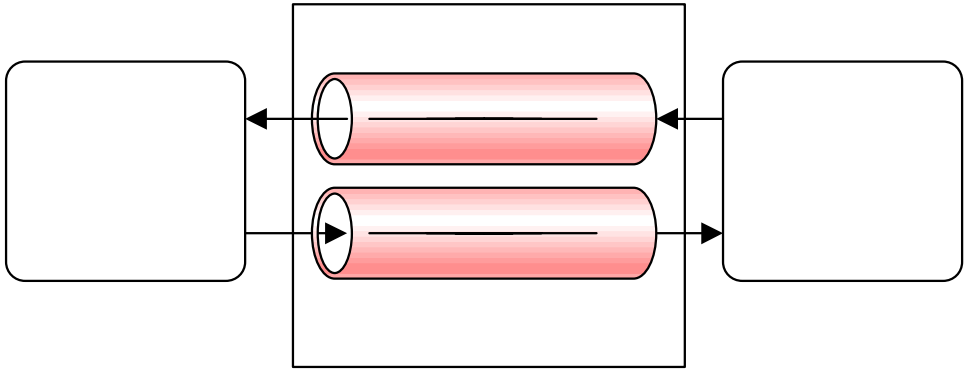
- Interprozesskommunikation
- Verschiedene Kommunikationsformen
  - Synchron vs. asynchron
  - Schmalbandig vs. breitbandig
  - Nachrichtenbasiert vs. Datenströme
  - Meldungen vs. Request-Response-Interaktionen
  - Rendezvous-Problem
- Spezifische Mechanismen
  - Signale
  - Shared Memory
  - Pipes
  - Ports und Adressierung (im Internet)
  - Sockets

Sockets

- Bisherige Annahmen
  - Verbindungslose Kommunikation mit Nachrichtenversand (UDP)
  - Bei verbindungsorientierter Kommunikation (TCP) muss zwischen den Endpunkten ein logischer Kanal aufgebaut werden
- Abstraktion im BS für beide Interaktionsformen: socket („Steckdose“)
  - logische Verbindung zwischen Kommunikationspartnern
  - typischerweise bidirektional, kann aber einseitig geschlossen werden
  - typischerweise eine Zuordnung von zwei Endpunkten (Ports) (und ggf. IP-Adressen)
  - aber auch als Broadcast (alle) oder Multicast (selektiv) möglich
  - ermöglicht verbindungsorientierte, strombasierte Kommunikation
- BSD 4.2 Unix Socket API für Netzprogrammierung
  - Details zu Socket-Programmierung später

Sockets (2)

- Abstraktion für bidirektionalen Datenaustausch



- Lokale Integration durch das Betriebssystem
  - Socket-Deskriptoren == File-Deskriptoren
  - Unterstützung gleicher System-Calls
    - `read()`, `write()`, `close()`
  - Besonderheiten für das Erzeugen und Verbinden von sockets
    - `Socket()`, `connect()`, `bind()`, `listen()`, … (analog zu `pipe()`)
  - Steuerung
    - `setsockopt()`, `getsockopt()`, `ioctl()`

Pipes, Sockets und Prozesse

- Pipes und Sockets liefern Ein-/Ausgabe über Prozessgrenzen hinweg

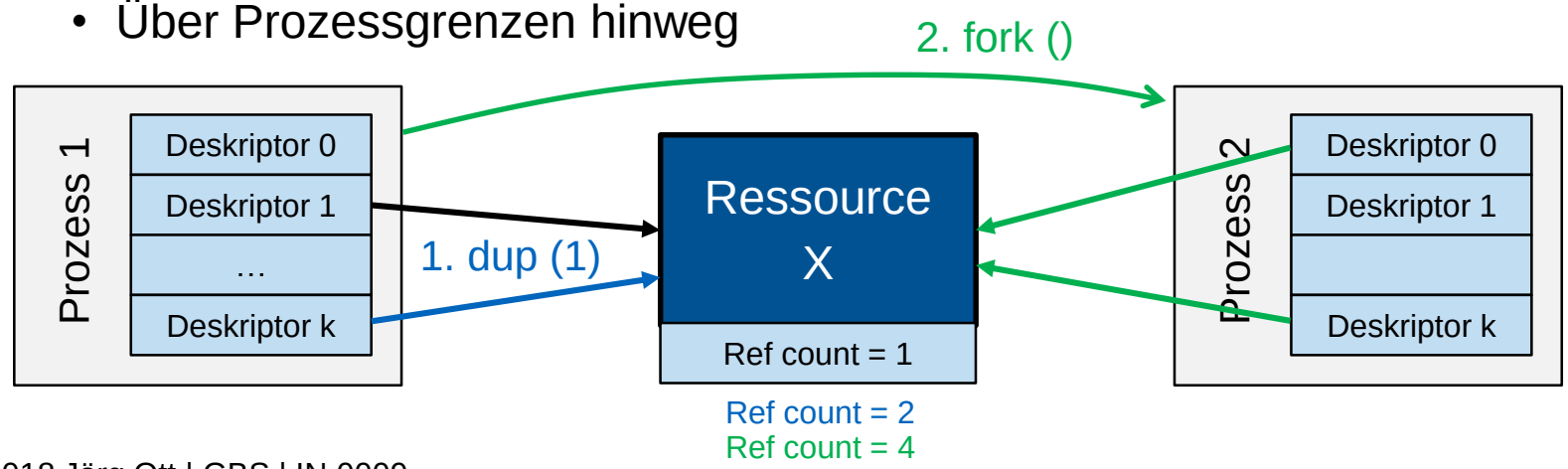
- Was passiert, wenn ein Prozess terminiert, der mit einem anderen verbunden ist?
  - Die Pipe bzw. Socket-Verbindung wird geschlossen
  - Signalisierung kann verschiedene Formen annehmen
    - EOF für Eingabe
    - Linux-Signal: SIGPIPE
    - Fehlermeldung, dass der entsprechende Deskriptor nicht mehr funktioniert
- Erfordert entsprechende Fehlerbehandlung

- Signal wird aber nur generiert, wenn die entsprechende Verbindung wirklich keinen Gegenüber mehr hat
  - Wichtig beim Erzeugen von Kind-Prozessen

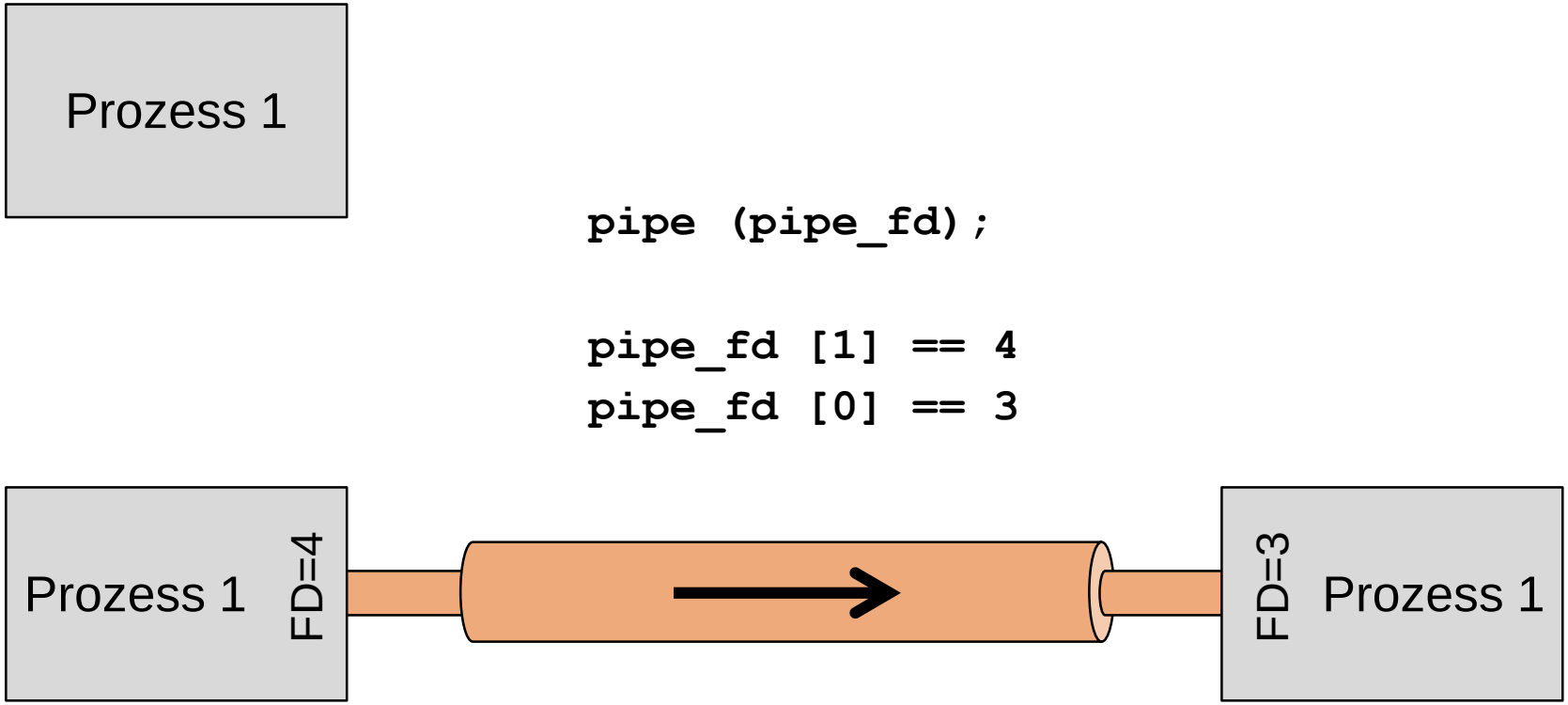
Pipes, Sockets und Prozesse (2)

- Exkurs zu Kernel-Ressourcen
  - Gilt für alle Formen von I/O-Deskriptoren
    - Standardein-/-ausgabe, Pipes, Sockets, geöffnete Dateien

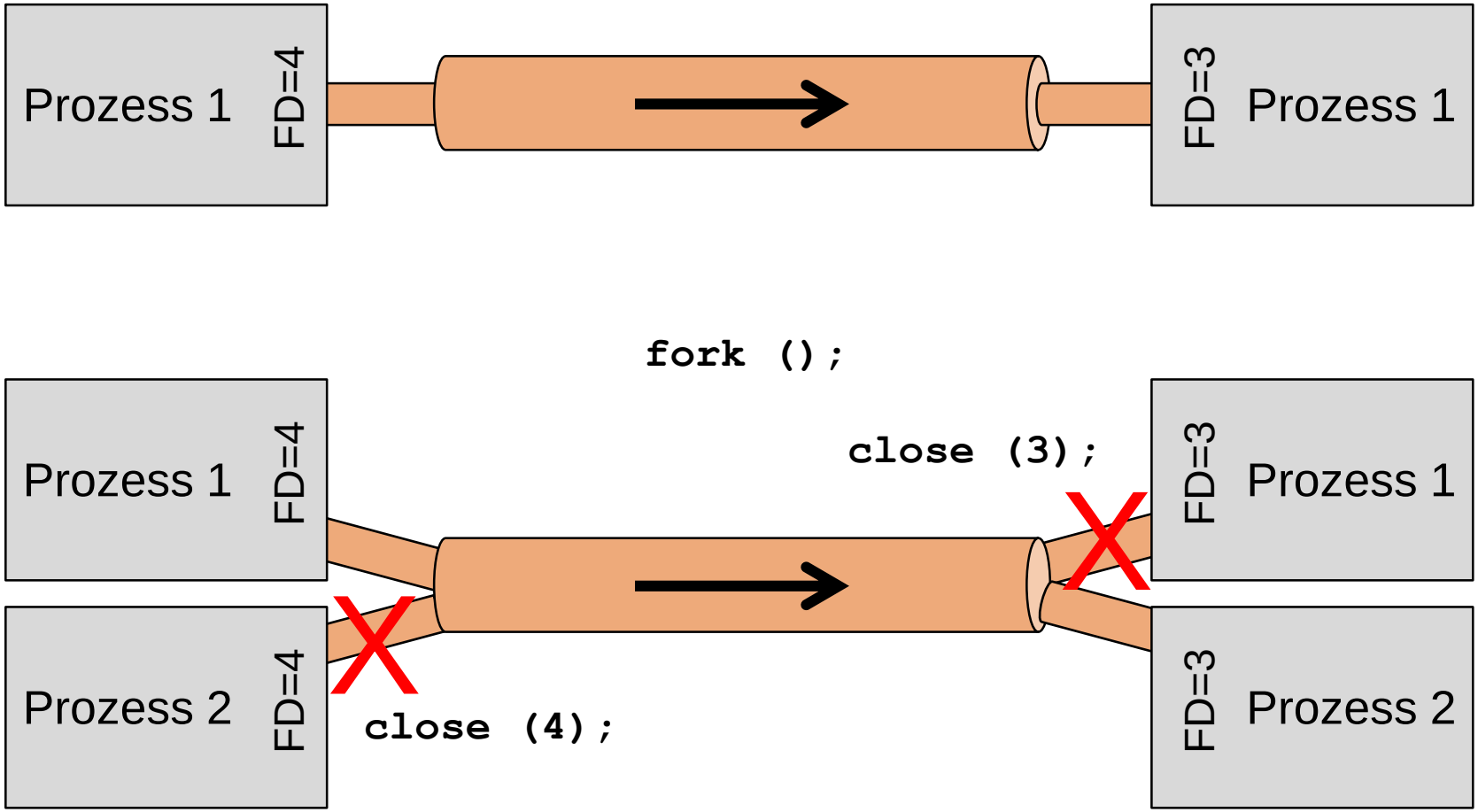
- Wenn eine Ressource geöffnet/genutzt wird, erhält sie
  - eine entsprechende Datenstruktur
    - Informationen über die Ressource
  - einen Reference-Counter
    - Wie oft diese Ressource gerade genutzt wird
    - Innerhalb eines Prozess
    - Über Prozessgrenzen hinweg



Pipes, Sockets und Prozesse (3)



Pipes, Sockets und Prozesse (4)



Pipes, Sockets und Prozesse (5)

- Durch ein fork() nutzen plötzlich zwei Prozesse eine gemeinsame Ressource im Kern
  - Trotz sonst getrennter Adressräume
- Implikationen
  - Wenn beide Prozesse darauf schreiben, ist die Reihenfolge der einzelnen Ausgaben nicht definiert
    - Siehe die Synchronisation der Ausgabe mehrerer Threads
  - Wenn beide Prozesse davon lesen, ist nicht definiert, welcher Prozess welchen Teil der Eingaben erhält
  - Wenn ein Prozess die Pipe/Verbindung schließt oder terminiert, so wird diese durch den anderen Prozess immer noch offen gehalten
    - Es wird also kein EOF, SIGPIPE oder Verbindungsende signalisiert
    - Denn der andere Prozess kann ja noch Daten senden.
- Wichtig: Deskriptoren schließen, die in einem Prozess nicht mehr gebraucht werden (z.B. in einer Shell mit Pipes)

Client-Server-Modell

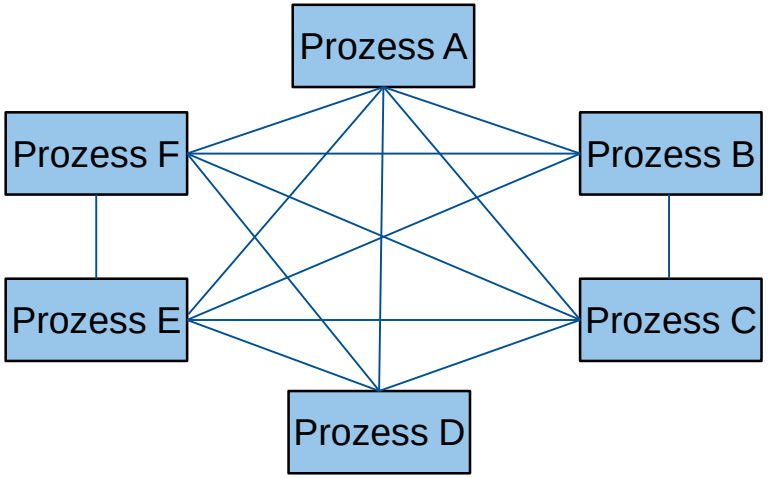
- Das Client-Server Modell basierend auf der verteilten Verarbeitung synchroner Aufträge
- Annahme: Endpunkte (Clients) sind ressourcenarm, Server sind ressourcenstark
- Server: ein dedizierter Prozess, der einen Dienst zur Verfügung stellt
  - Beispiele: Dateidienst (File-Server, Web-Server), Namensdienst (DNS-Server)
- Client initiiert Aufträge an einen von einem Server angebotenen Dienst
  - Zuvor findet er den (bzw. einen geeigneten) Server z.B. über einen Namensdienst
- Clients sind kurzlebigere Prozesse und einem Server a-priori nicht bekannt
- Server sind langlebige Prozesse, Aufträge werden in einzelnen Threads (Worker-Threads) abgearbeitet
- Clients und Server kommunizieren über Nachrichten und werden i.d.R. auf verschiedenen Rechnern ausgeführt
  - Diese Nachrichten können dennoch über einen Datenstrom übertragen werden

Alternative: Peer-to-Peer-Modell (P2P)

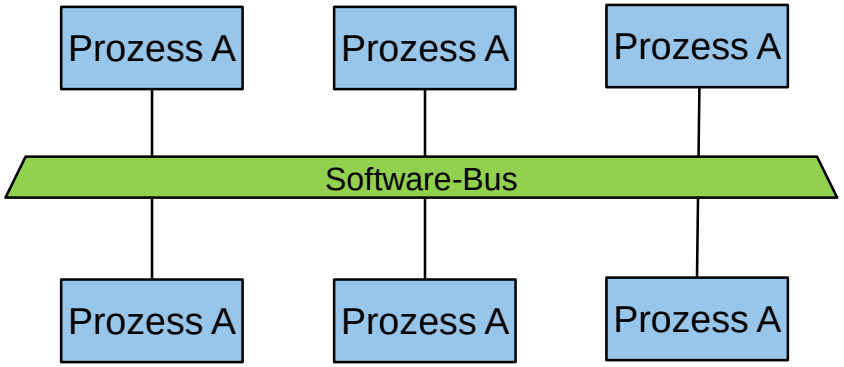
- Keine Unterscheidung zwischen Client und Server
  - Die beteiligten Systeme übernehmen beide Rollen
- Prinzipieller Ansatz
  - Eine Gruppe von Systemen organisiert sich selbst
    - Systeme treten der Gruppe explizit bei
      - Auffinden der Gruppe z.B. über DNS
    - Systeme bauen Verbindungen zueinander auf
      - z.B. Ringstruktur, Grid, Mesh, ...
  - Jedes System steuert Ressourcen zur Gemeinschaft bei
    - Speicherplatz, Rechenkapazität, Dienste
    - Verteilter Ansatz statt zentral bereitgestellter Ressourcen
    - Systeme wachen übereinander und ersetzen ausfallende Peers
  - Angebotene Dienste werden in gruppenspezifischen Verzeichnis (Directory Service) registriert
    - Diese können mit einem Discovery-Protokoll gefunden werden
- Strukturierung möglich (z.B. reguläre Peers vs. Super-Peers)
- In der Vorlesung nicht weiter betrachtet

Software-Bussysteme

- Bisherige Annahmen
  - (Verbindungsorientierte) Kommunikation zwischen jeweils zwei Endpunkten
  - Prozesse können / wollen Aktivitäten miteinander abstimmen
  - Sie sind daher an gegenseitigen Benachrichtigungen interessiert
  - Bei gleichzeitiger Kommunikation mit mehreren Kommunikationspartnern müssen (quadratisch) viele Verbindungen aufgebaut werden
- Lösung: Kommunikation über einen gemeinsamen Nachrichten-Bus



N x (N – 1) Verbindungen: $O(N^2)$



N Verbindungen: O(N)

Beispiel: Dbus (Linux)

- In Analogie zu Hardware-Bussystemen
  - PCI Express Bus (in PCs), frühes Ethernet
- Ein oder mehrere Busse für alle Kommunikationspartner (System Bus, Session Bus)
- Adressierung über Namen (well-known names)
  - Jeder Teilnehmer bekommt eine Adresse und einen oder mehrere Namen
  - Konventionen über Namensvergabe erforderlich (wie bei Ports)
- Möglichkeit, Signale und Funktionen zu registrieren
- Verschiedene Interaktionsformen
  - Remote Method Invocation (RMI) / Remote Procedure Call (RPC)
    - Request-Response
  - Publish/Subscribe (Beispiel: MQTT)
    - Prozesse registrieren Interesse an bestimmten Nachrichten/Ereignissen (über Namen)
    - Ereignisse werden dann unter diesen Namen „publiziert“ und erreichen genau die Interessenten
- Geschwindigkeit: Früher rein im Userspace, mittlerweile auch im Kernel

Beispiel: Dbus (Linux)

- Vorteile
  - Kanal ist breitbandiger als reine Linux-Signale
  - Applikationen können sich für Nachrichten selektiv registrieren

- Beispiele
  - Neue Hardware (z.B. USB-Stick) verbunden. Dateimanager kann direkt reagieren.
  - Network Manager setzt neue IP-Adresse. Applikationen können reagieren.
  - Mit folgendem Befehl können alle auf dem Bus verbundenen Endpunkte gelistet werden
    - `dbus-send -system -dest=org.freedesktop.DBus -print-reply /org/freedesktop/DBus org.freedesktop.DBus.ListNames`
  - Die Einträge mit Namen bieten registrierte Dienste an

Socket-Programmierung

- Sockets eingeführt in BSD 4.2 Unix
- BSD-Unix-Mechanismus für Interprozesskommunikation
- Transparenz lokaler und entfernter Kommunikation
- Unterstützen unterschiedliche Interaktionsformen
- Essentiell für die Verbreitung von TCP/IP in den 1980ern!

Übersicht

- Sockets abstrahieren von den technischen Details der Kommunikation
  - Übertragungsmedium, Paketgröße, Paketwiederholung bei Fehlern

- Ein Socket unterstützt eine Reihe von Basisoperationen:
  - socket(): Aufbau eines neuen Sockets (anfangs als einzelner Endpunkt)
  - bind(): Server: assoziiere Socket mit einem Port
  - listen(): Server: warte auf eintreffende Daten
  - accept(): Server: akzeptiere Verbindungswünsche von Clients
  - connect(): Client: Initiierung der Verbindung zum Server
  - send(): Senden von Daten. Alternativ: read()
  - recv(): Empfangen von Daten. Alternativ: write()
  - close(): einseitiges Beenden der Verbindung

Ablauf: Server

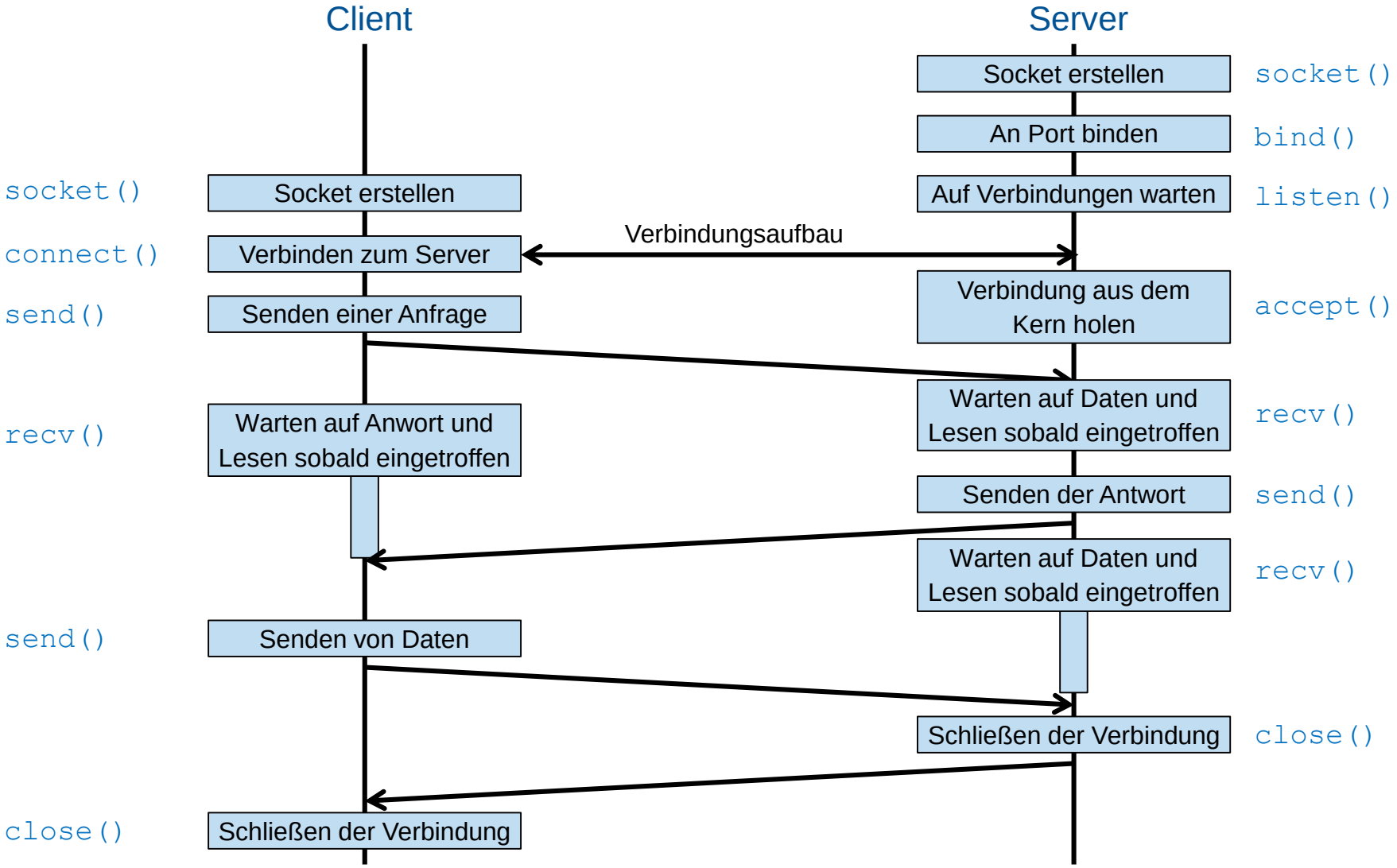
- Randbedingungen: Der Server...
  - kommuniziert mit N Clients, die a priori nicht bekannt sind
  - muss auf eintreffende Verbindungswünsche reagieren können
  - nimmt die Assoziierung mit einem Port vor und akzeptiert Verbindungswünsche
- Ablauf aus Serversicht
  - socket(): Es wird zuerst ein ungebundener Socket erzeugt
  - bind(): Der Socket wird mit einem bestimmten Port verbunden (localhost:4444)
  - listen(): Anschließend wartet der Server auf Verbindungswünsche von Clients
  - accept(): Eingehende Verbindungswünsche werden einzeln akzeptiert und liefern jeweils einen neuen Socket
  - read/write(): Austausch von Daten zwischen Client und Server
  - close(): Schießen der Verbindung auf Serverseite

Anlauf: Client

- Randbedingungen: Der Client...
  - initiiert eine Verbindung durch Senden eines Verbindungswunsches an den Port des Servers

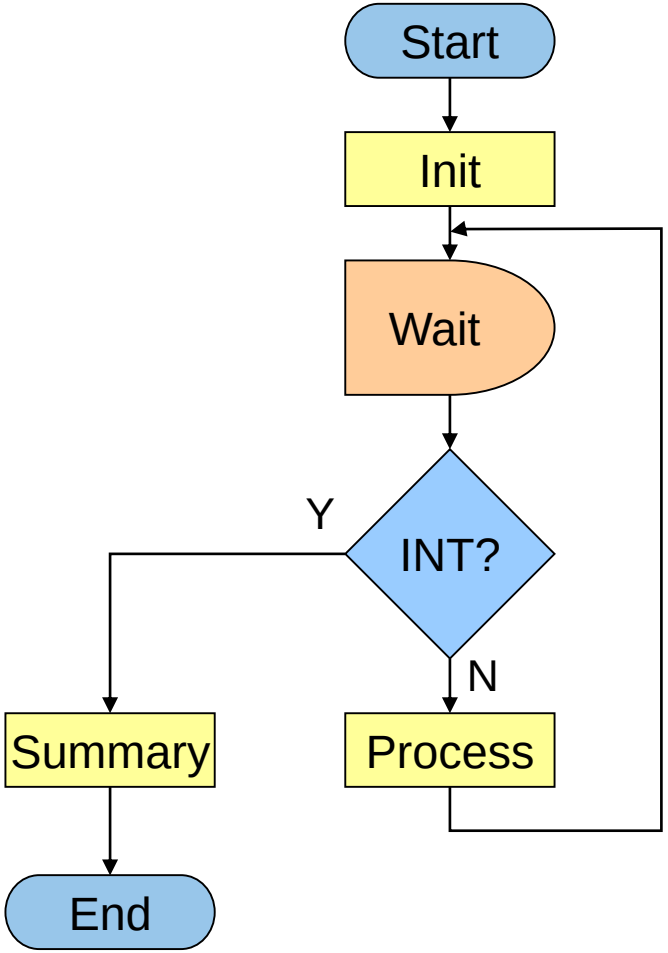
- Ablauf aus Clientsicht:
  - socket(): Es wird zuerst ein ungebundener Socket erzeugt
  - connect(): Der Socket wird mit dem Port des Servers verbunden (z.B. localhost:4444)
  - read/write(): Austausch von Daten zwischen Client und Server
  - close(): Schießen der Verbindung auf Clientseite

Beispiel: Einfache Client-Server-Interaktion



Socket-Programmierung: Details

Typische Programmstruktur



Initialisierung

- Kommandozeile parsen, Argumente interpretieren
- Ggf. Rechnernamen auflösen
- Sockets erzeugen, ggf. Port registrieren
- Signal-Behandlung initialisieren

Hauptschleife

- Socket-Deskriptoren verwalten
- Daten lesen, interpretieren
- Ausgaben/Antworten erzeugen
- Signal-, Timeout und Fehlerbehandlung

Aufräumen

- Alle Deskriptoren schließen
- Speicher freigeben
- Geordnet terminieren

Socket erzeugen

```
int socket(domain, type, proto)
```

- Erzeugt einen neuen Kommunikationsendpunkt
- Domain: verschiedene Adressarten (~30 in socket.h)
  - (Named) Pipes (e.g., AF_UNIX), ...
  - Internet Protocols (AF_INET, AF_INET6)
- Type: unterschiedliche Interaktionsformen
  - SOCK_STREAM: TCP = stromorientierte Kommunikation
  - SOCK_DGRAM: UDP = nachrichtenbasierte Kommunikation
    - Achtung: UDP ist unzuverlässig!
- Proto: Protokollfamillie
  - Muss zur Domain passen, meistens auf 0 gesetzt

Datenstrukturen für Adressen

- Adressierung eines Kommunikationspartners über dessen IP-Adresse
- Spezialfall: lokaler Rechner: 127.0.0.1 bzw. ::1 (localhost)

```
struct sockaddr_in {
    sa_family_t    sin_family;
    in_port_t      sin_port;
    struct in_addr sin_addr;
};
```

IPv4-Adresse (historisch motiviert, unhandlich)

```
struct in_addr {
    in_addr_t      s_addr;
};
```

```
struct sockaddr_in6 {
    sa_family_t     sin6_family;
    in_port_t         sin6_port;
    uint32_t          sin6_flowinfo;
    struct in6_addr sin6_addr;
};
```

IPv6-Adresse (in Kurzform)

```
struct in6_addr {
          uint8_t     u6_addr8[16];
#define s6_addr in6_u.u6_addr8
};
```

Passives Warten (z.B. Server)

- Registieren eines Ports
- Warten auf den Empfang von Nachrichten (UDP)
- Warten auf eingehende Verbindungen (TCP)

```
int bind (int sd, struct sockaddr *, socklen_t len);
```

- UDP: fertig

- TCP: Einrichten einer Queue für eingehende Verbindungen
  - `int listen (int sd, int backlog);`
  - Gestattet <backlog> ausstehende Verbindungen im Kern

- `setsockopt()` und `ioctl()`
  - Zum Setzen weiterer Socket-Parameter
  - Siehe Man-Pages für Details, hier nur sekundär von Bedeutung

Verbindungen (TCP)

- `int connect (int sd, struct sockaddr *target, socklen_t len);`
  - Baut (synchron) eine Verbindung zu einem Server auf
  - Synchron
    - Systemcall kehrt erst zurück, wenn die Verbindung aufgebaut wurde oder
    - Wenn ein Fehler gemeldet wird (z.B. kein Serverprozess gestartet) oder
    - Wenn eine im System vorgegebene Wartezeit überschritten wurde (Timeout) (z.B. weil der Server nicht erreichbar war)
  - Asynchron: Option TCP_NODELAY für den Socket einschalten
- `new_sd = accept (int sd, struct sockaddr *peer, socklen_t *peerlen);`
  - Annehmen einer neuen Verbindung
    - Explizites Ablehnen nicht vorgesehen → sofortiges `close()`
  - Liefert einen neuen Socket-Deskriptor für diese Verbindung
  - Zeigt an, wer der Kommunikationspartner der Verbindung ist
  - Der ursprüngliche (sd) bleibt zum Annehmen weiterer Verbindungen
- `close (sd)`
  - schließt eine Verbindung; Achtung: Datenverlustrisiko!
- `shutdown (int sd, int mode)`
  - Lokal: nicht mehr senden/empfangen

Senden von Daten

- Datenstrom (TCP)

```
write (int sd, char *buffer, size_t length);
writev (int sd, struct iovec *vector, int count);
```

  - Liste von Puffern mit Speicherbereichen und Länge

```
send (int sd, char *buffer, size_t length, int flags)
```

  - Steuerung des Versendens durch ergänzende Flags

- Nachrichtenbasiert (UDP)

```
sendto (int sd, char *buffer, size_t length, int flags,
        struct sockaddr *target, socklen_t addrlen)
sendmsg (int sd, struct msghdr *msg, int flags)
```

  - Zieladresse (muss pro Nachricht angegeben werden)
  - Zeiger auf Speicherbereich mit Daten und Länge
  - Steuerung des Versendens durch ergänzende Flags

Empfangen von Daten

- Datenstrom (TCP)

```
read (int sd, char *buffer, size_t length);
readv (int sd, struct iovec *vector, int count);
```

  - (Liste von) Puffer(n) mit Speicherbereich(en) und Länge

```
recv (int sd, char *buffer, size_t length, int flags)
```

  - Steuerung des Empfangvorgangs über Flags
  - z.B. auch Fehlermeldungen, asynchron empfangen

- Nachrichtenbasiert (UDP)

```
recvfrom (int sd, char *buffer, size_t length, int flags,
            struct sockaddr *target, socklen_t addrlen)
recvmsg (int sd, struct msghdr *msg, int flags)
```

  - Liefert die Absenderadresse
  - Puffer zum Ablegen der Daten
  - Steuerinformationen

Weitere Funktionen

- `getpeername (int sd, struct sockaddr *peer, size_t *len)`
  - Auslesen der Adresse des Kommunikationsparters

- `getsockname (int sd, struct sockaddr *local, size_t *len)`
  - Auslesen der eigenen Socket-Adresse
    - Nützlich für dynamisch vom Betriebssystem zugewiesene Ports

- Modifizieren von Socket-Einstellungen

```
getsockopt (int sd, int level, int option_id, char *value, size_t length)
setsockopt (int sd, int level, int option_id, char *value, size_t length)
```

  - Beispiel
    - Puffergröße im Kern, Paket-Lebensdauer (TTL), Protokollparameter

```
ioctl (int sd, int request, ...);
fcntl (int sd, int cmd [, long arg] [, ...]);
```

  - Beispiel
    - Asynchrones Senden/Empfangen

Namensauflösung

- Umwandeln einen symbolischen Namens in eine protokollspezifische Adresse
- Achtung: unterschiedliche Adressformate und Längen

- APIs
  - **`gethost*(), inet_aton(), inet_ntoa()`**
  - **`getaddrinfo(), inet_pton(), inet_ntop()`**

⇨ alt, funktionieren aber noch

Alte Funktionen

```
int gethostname (char *name_buffer, int buffer_length)
struct hostent *gethostbyname (char *namestr)
struct hostent *gethostbyaddr (struct sockaddr *, size_t, int);

struct hostent {
        char            *h_name;
        char            **h_aliases;
        int             h_addrtype;
        int             h_length;
        char            **h_addr_list;
#define h_addr          h_addrlist [0]
};

struct hostent *gethostent ();
endhostent ();
```

getaddrinfo

int getaddrinfo(host,server,hints,result)

```
struct addrinfo {
     int              ai_flags;      /* AI_PASSIVE, AI_CANONNAME,
                                        AI_NUMERICHOST */
     int              ai_family;     /* PF_UNSPEC */
     int              ai_socktype;   /* SOCK_xxx */
     int              ai_protocol;   /* 0 or IPPROTO_xxx for IPv4 and IPv6 */
     size_t           ai_addrlen;    /* length of ai_addr */
     char             *ai_canonname; /* canonical name for nodename */
     struct sockaddr  *ai_addr;      /* binary address */
     struct addrinfo  *ai_next;      /* next structure in linked list */
};

void freeaddrinfo(struct addrinfo *res);
const char *gai_strerror(int errcode);
```

Konvertierungsfunktionen (1)

- IP-Adressen

Dotted decimal notation: aaa.bbb.ccc.ddd (nur IPv4)

```
in_addr_t inet_addr (char *buffer)
in_addr_t inet_aton (char *buffer)
char *inet_ntoa (in_addr_t ipaddr)
```

aaa.bbb.ccc.ddd (IPv4), aaaa:bbbb:cccc:dddd:eeee:ffff:gggg:hhhh (IPv6)

```
int inet_pton (int af, const char *src, void *dst)
```

dst: in_addr bzw. in6_addr

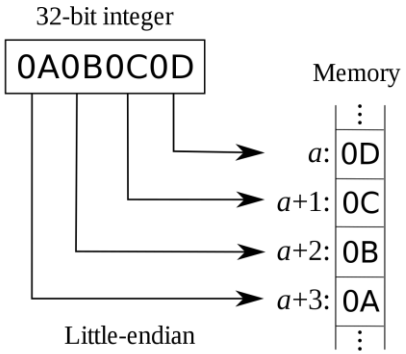
```
const char *inet_ntop (int af, const void *src, char *dst, size_t)
```

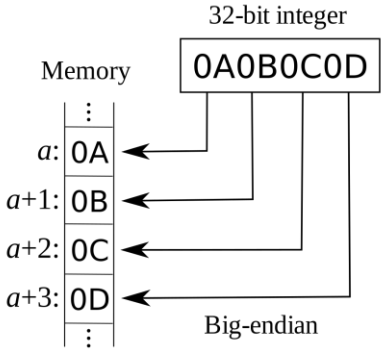
src: in_addr bzw. in6_addr
char dst [INET_ADDRSTRLEN] bzw. char dst [INET6_ADDRSTRLEN]

Konvertierungsfunktionen (2)

- Network vs. Host Byte Order
  - Daten im Netz werden immer “Big Endian” übertragen
  - Lokale CPU verwendet eine andere Repräsentation
  - Beispiel: Intel x86 nutzt “Little Endian”

  Konvertierung in lokale Repräsentation potentiell notwendig
  - Erforderlich für “Little-Endian” (LSB-first) Architekturen wie Intel
  - No-op für MSB-first-Architekturen
  - Sollte **immer** gemacht werden für Portabilität





Quelle: Wikipedia-Artikel zu „Endianness“

```
netshort  = htons (hostshort)     16-bit
netlong   = htonl (hostlong)      32-bit
hostshort = ntohs (netshort)      16-bit
hostlong  = ntohl (netlong)       32-bit
```

- Siehe auch
  - `bswap_16(), bswap_32(), bswap_64()`
  - `htobe{16|32|64}(), htole{16|32|64}()`
  - `betoh{16|32|64}(), letoh{16|32|64}()`

Beispiel: Client

```
#include <arpa/inet.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>

int main (int argc, char *argv []) {
    int                fd = 0, n = 0;
    char               rb [1024] = { 0 };
    struct sockaddr_in sa;

    if ((fd = socket (AF_INET, SOCK_STREAM, 0)) < 0) {
        return -1; // No socket available
    }
    sa.sin_family = AF_INET;
    sa.sin_port   = htons (5000);
    sa.sin_addr.s_addr=inet_addr ("127.0.0.1");
    if (connect (fd, (struct sockaddr *) &sa, sizeof (sa))) {
        return -1; // Connect failed
    }
    while ((n = recv (fd, rb, sizeof (rb) -1, 0)) > 0) {
        rb [n] = '\0 ';
        printf ("%s\n", rb);
    }
    if (n < 0) return -1; // Error during recv
    close (fd);
    return 0;
}
```

Beispiel: Server

```
#include <arpa/inet.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <unistd.h>

int main (int argc, char *argv []) {
    int                 s = 0, fd = 0;
    struct sockaddr_in sa;
    char                sendbuf [1025] = "Message from Server\n";

    if ((s = socket (AF_INET, SOCK_STREAM, 0)) == -1)
        return -1;
    memset (&sa, 0, sizeof (sa));
    sa.sin_family      = AF_INET ;
    sa.sin_addr.s_addr = htonl (INADDR_ANY);  // do not pick a specific local IP address
    sa.sin_port        = htons (5000);
    if (bind (s, (struct sockaddr *) &sa, sizeof (sa)) == -1)
        return -1;
    if (listen (s, 10) == -1)
        return -1;
    while (1) {
        fd = accept (s, NULL, NULL);
        send (fd, sendbuf ,strlen (sendbuf), 0);
        close (fd);
    }
    close (s);
    return 0;
}
```

I/O-Multiplexing

- Was tun, wenn ein Prozess Daten auf mehreren Verbindungen empfangen möchte
  - Round-Robin asynchron abfragen (busy waiting) ist unpraktisch
- Betriebssystem liefert Funktionen, die das Überwachen mehreren Eingabequellen gleichzeitig gestattet
  - Blockieren, bis mindestens eine Eingabequelle Daten zu lesen hat
  - Optional: bis ein Timeout abgelaufen ist
- Liefert dann eine Liste der Deskriptoren, von denen gelesen werden kann

I/O-Multiplexing: select

```
int select (maxfdset, readset, writeset, extset, timeout)
```

- Auflisten aller zu beobachtenden Deskriptoren (fdset)
  - Ggf. Timeout bestimmen
    - Datenstruktur `struct timeval { … }` zur Angabe des Zeitinternvalls als Paar (Sekunden, Mikrosekungen)
    - (0, 0) == Busy Waiting
    - NULL-Pointer == Blockieren (kein Timeout)
- Aufrufen von `select()`
- Rückgabewerte: Fehler untersuchen
  - -1: Fehler untersuchen
    - Fatal → Terminate
    - Reparierbar (z.B. interrupted system call) → nochmal
  - 0: Timeout-Behandlung
  - > 0 = Anzahl der Deskriptoren mit z.B. verfügbaren Eingaben
    - fdset zeigt an, welche Deskriptoren z.B. lesbar sind
    - Durchgehen, Daten lesen, bearbeiten

fd_set Makros

```
#include <sys/select.h>

fd_set rfdset;          // Deskriptoren zum Warten auf Eingabe
FD_ZERO (&rfdset);      // initial mit Nullen füllen
FD_SET (fd, &rfdset); // jeden Deskriptor einzeln setzen
   ...
select (..., &rfdset, NULL, NULL, NULL);
   ...
if (FD_ISSET(fd, &rfdset)) { // jeden Deskriptor einzeln prüfen
   ...
}
```

I/O Multiplexing (poll)

```
int poll (struct pollfd [], int n_fd, int timeout)
```

```
struct pollfd {
    int     fd;         // file descriptor
    int     events;     // events to watch for
    int     revents;    // occurred events
};
```

- Heißt zwar poll(), macht aber kein Polling (= Busy waiting)
- Poll events
  - POLLIN Warten auf Eingabe / Eingabe verfügbar
  - POLLOUT (Prüfen ob) Socket schreibbar (asynchrone E/A)
  - POLLHUP, POLLERR
- Timeout in Millisekunden
  - -1 == no timeout, 0 == return immediately (perform real polling)
- Nutzung vergleichbar mit `select()`

Timeouts (1)

- Interprozesskommunikation nutzt oft Timeouts
  - Häufiges Setzen, Zurücksetzen, Abschalten
  - Muss deshalb effizient implementiert werden
- select () und poll () erlauben genau einen Timeout
  - poll () in Millisekunden
  - select () in Mikrosekunden mittels struct timeval
- Verwalten einer sortierten Liste aller Timeouts
  - Jeweils immer die absolute Zeit merken, wenn Timeout gesetzt wird
  - Außerdem Kontext merken, zu dem der Timeout gehört
    - Beispiel: maximale Wartezeit auf Antwort, dann erneut versuchen
- Vor dem Aufrufen von select/poll
  - Aktuelle – ebenfalls absolute – Zeit bestimmen (gettimeofday ())
  - Den frühesten Timeout aus der Timeout-Liste nehmen
    - Prüfen, ob dieser Zeitpunkt bereits verstrichen ist → ggf. sofort behandeln
    - Schließlich kann viel Zeit seitdem Setzen des Timeout vergangen sein
  - Abstand von frühestem Timeout zur aktuellen Zeit bestimmen
  - Dieses Delta an poll/select() als Timeout übergeben
- Hierzu gibt es Event-Loop-Bibliotheken (etwa libevent oder libev)

Timeouts (2)

Beispiel: Timeout 200ms

```
struct timeval    tv, delta, now;

/* some event -> calculate absolute time till timeout in tv */
gettimeofday (&tv, NULL);
tv.tv_usec += 200*1000;
if (tv.tv_usec >= 1000000) {
          tv.tv_usec -= 1000000;
          tv.tv_sec++;
}

/* ... many other activities -> back in mainloop */
gettimeofday (&now, NULL);
delta.tv_usec = tv.tv_usec - now.tv_usec;
delta.tv_sec  = tv.tv_sec  - now.tv_sec;
if (delta.tv_usec < 0) {
    delta.tv_usec += 1000000;
    delta.tv_sec--;
}
if (delta.tv_sec < 0) {
    /* timeout has also passed -> handle now */
}
switch (n = select (..., ..., ..., ..., &delta) {
    ...
}
```

Fazit

- Verschiedene Konzepte zur Interprozesskommunikation
  - Schmalbandig durch minimale Mitteilungen (z.B. Signale)
  - Breitbandig zur Übermittlung komplexer bzw. großer Datenmengen
  - Implizit durch gemeinsame Ressourcen
  - Explizit durch Signale, Nachrichtenaustausch, Ströme
  - Synchron vs. asynchron

- Wesentliche Mechanismen
  - Gegeneinander abgegrenzte Nachrichten, z.B. UDP, DBus
  - Ströme: Abstraktion nachrichtenbasierter Verbindungen, verdeckte Nachrichtengrenzen
  - Pipe: Unidirektionaler Strom zwischen zwei Kommunikationspartnern
  - Port: Endpunkt einer Kommunikation, kann bei Bedarf dynamisch eingerichtet und gelöscht werden
  - Socket: Gemeinsame Abstraktion verschiedener Kommunikationsformen zwischen zwei Endpunkten (Ports)