Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

3 Prozesse

3.1 Was ist ein Prozess?

Prozess:

- Ausführung eines logisch zusammengehörenden Programms
- Verwaltungseinheit des Betriebssystems
    - Prozesse belegen Betriebsmittel
    - Prozesse sind aktive Komponenten des Gesamtsystems

Programm	Prozess
Statisch, wiederholbar	Dynamisch, nicht wiederholbar Einmalige Abläufe (Prozessidentifikation PID) Braucht ein Programm Ein Objekt im Sinne der Informatik (Prozessbeschreibung und Methoden)

- Im Sinne des BT besteht ein Prozess aus:
    - Unveränderlichem Teil: Code, Konstanten
    - Veränderlichem Teil: Globale Daten, Stack, Heap
    - Kontext (Umgebung): Register, Zeit, Verwaltungsdaten

Eigenschaften:

- Das BT ordnet einzelnen Prozessen das Betriebsmittel CPU zu und entzieht es ihnen wieder.
- Bei einem Prozessor: Quasi-simultane Ausführung der Prozesse
- Auf einem n-Prozessorsystem laufen zu einem Zeitpunkt maximal n Prozesse.

Typische Zustände eines Prozesses:

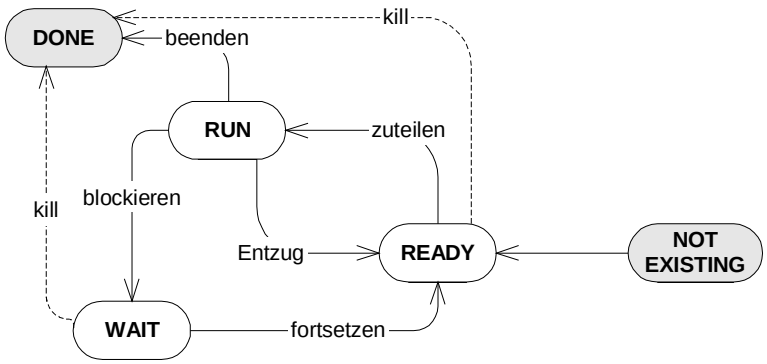

Abbildung 22: Prozess-Zustände

- Prozesserzeugung:
    - Unix: fork()
    - Win32: CreateProcess()

Begrifflichkeiten für lauffähige Objekte des BT:

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

- Üblicherweise wird heutzutage *Prozess* verwendet.
- Früher (Batch-Betrieb) war *Job* gebräuchlich.
- In Embedded Systemen oft auch *Task* (typischerweise kein Speicherschutz).
- *Thread* oder *LWP (Light-Weight Process)* bezeichnet lauffähiges Objekt innerhalb eines Prozesses, die typischerweise Resourcen mit dem Prozess teilen (Dateien, Speicher, etc)

3.2 Prozessverwaltung

3.2.1 Prozesskontrollblöcke

Prozesskontrollblock (PLB, PCB) oder Task Control Block (TCB). Pro Prozess gibt es genau einen PCB, der diesen Prozess beschreibt. Das BT verwaltet die Prozesse anhand der PCBs.

Enthält alle wesentlichen Eigenschaften eines Prozesses:
- Beschreibende Eigenschaften:
    - Name
    - Art
    - Basispriorität
    - Codedatei-Verweis
    - Betriebsmittel-Bedarf: Speicherbedarf, weitere Resourcen, Zeitbedarf
- Prozessorverwaltung:
    - Prozess-Zustand (READY, WAITING, etc.)
    - Aktuelle Priorität
    - Prozessidentifikation PID (bzw. TID)
    - Prozessor-Zustand:
        - Spezialregister-Inhalte: Programmzähler, Flags, Statuswort, Unterbrechungsmasken, Schutzmasken, Adressumsetzungsdaten, Stack-Zeiger
        - Daten- und Adressregister-Inhalte
    - Ggfs. verbrauchte Zeit
- Speicherverwaltung
    - Zeiger auf Programm (CODE)
    - Zeiger auf Daten (DATA)
    - Zeiger auf Stack
    - Ggfs. Shared Memory
- Dateiverwaltung (I/O)
    - Root Directory
    - Aktuelles Directory
    - Geöffnete Dateien
    - Zugriffsrechte
    - User ID, Group ID
- Verwaltungsdaten
    - Rechte
    - Statistikdaten

Implementierung:
- Entweder alle Eigenschaften in einer Struktur
- Oder Verweise auf Unterstrukturen

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

3.2.2 Verwaltung der PCBs

Viele Prozesse möglich, die effektiv verwaltet werden müssen -> Datenstruktur-Problem

- Einzelne Skalare (Variablen)
    - Bei eingebetteten Systemen sind oftmals alle Tasks statisch und zur Compilezeit bekannt
- Konstantes Array (Feld)
    - Maximale Anzahl der Prozesse vorgegeben
- Variable verkettete Liste
- Baum

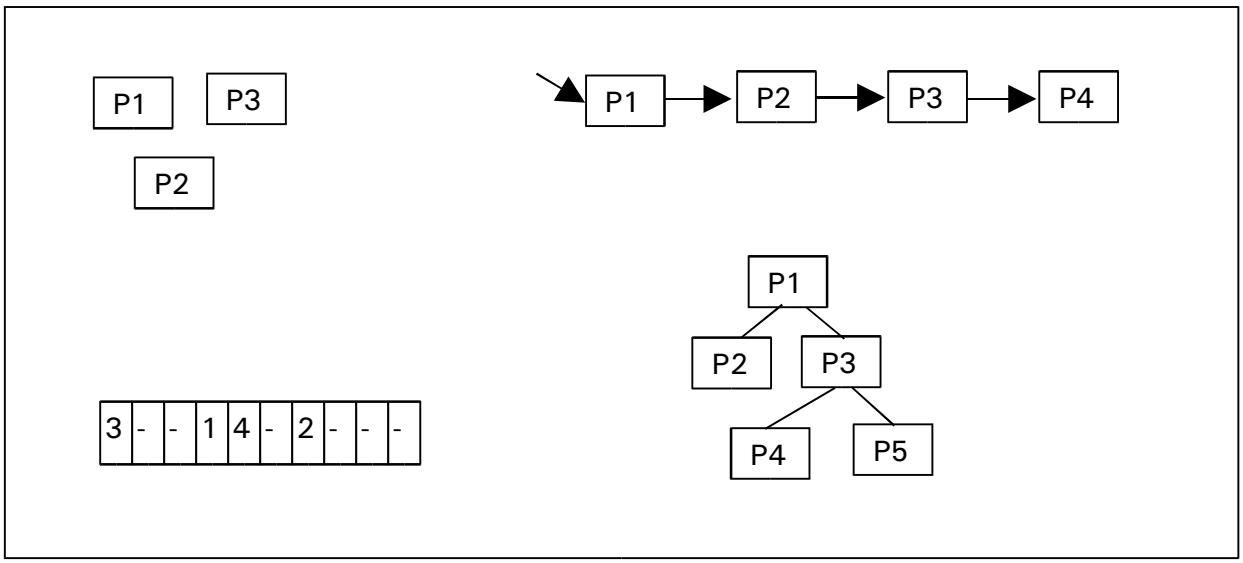

Abbildung 23: Organisation von PCBs

Teilmengenbildung bezüglich wichtiger Attribute, z.B. je eine Queue für Prozesse im Zustand READY, WAITING, RUN, dient zur Effizienzsteigerung

3.2.3 Prozesserzeugung & -vernichtung

3.2.3.1 Statisch vs. Dynamisch

Statische Betriebssysteme:
Alle Prozesse sind vorher bekannt und statisch definiert. Z.B. bei eingebetteten Systemen.

- Alle Prozesse werden „am Schreibtisch" statisch festgelegt
    - PCBs werden als Variable gespeichert, z.B.
      struct task_struct MyTask1;
- Prozesse werden für eine bestimmte Anwendung geschrieben
- Die PCBs werden vom einem Tool einmalig erzeugt
- Keine Vernichtung von Prozessen

Dynamische Betriebssysteme:
Die Prozesse werden mittels Kernel-Aufrufen erzeugt und vernichtet.

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

3.2.3.2 Prozesserzeugung

- Systeminitialisierung
- Aus anderem Prozess heraus
- Durch Benutzeranforderung
- Als Batchjob

3.2.3.3 Prozessvernichtung

- Normales Ende (freiwillig)
- Fehlerzustand (freiwillig)
- Systemfehler (unfreiwillig)
- Killed (durch anderen Prozess)
- **Unix:** *exit()*
- **Win32:** *ExitProcess()*

3.2.3.4 Beispiel Unix: fork()

Unix-Aufruf zum Erzeugen eines Prozesses:

pid_t fork(void);

- Erzeugt eine „Kopie" des aktuellen Prozesses
  - Dasselbe Programm
  - Dieselben geöffneten Dateien
  - Neue PID
  - Der gesamte Speicher wird kopiert und dem neuen Programm zur Verfügung gestellt
- Nach dem fork() Aufruf existieren zwei verschiedene Prozesse für dasselbe Programm.
  - Falls der Rückgabewert ungleich 0 ist: ursprüngliches Programm, Rückgabewert = PID des Kindes
  - Ist der Rückgabewert 0, so handelt es sich um den Kindprozess.

Pseudo-Code:

```
int pid = fork();
if( pid==0 ) {
      // dieser Code wird im Kindprozess ausgeführt
} else {
      // dieser Code wird im Vaterprozess ausgeführt
}
```

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

In Unix strenge, unabänderliche Prozesshierarchie: einmal child, immer child:

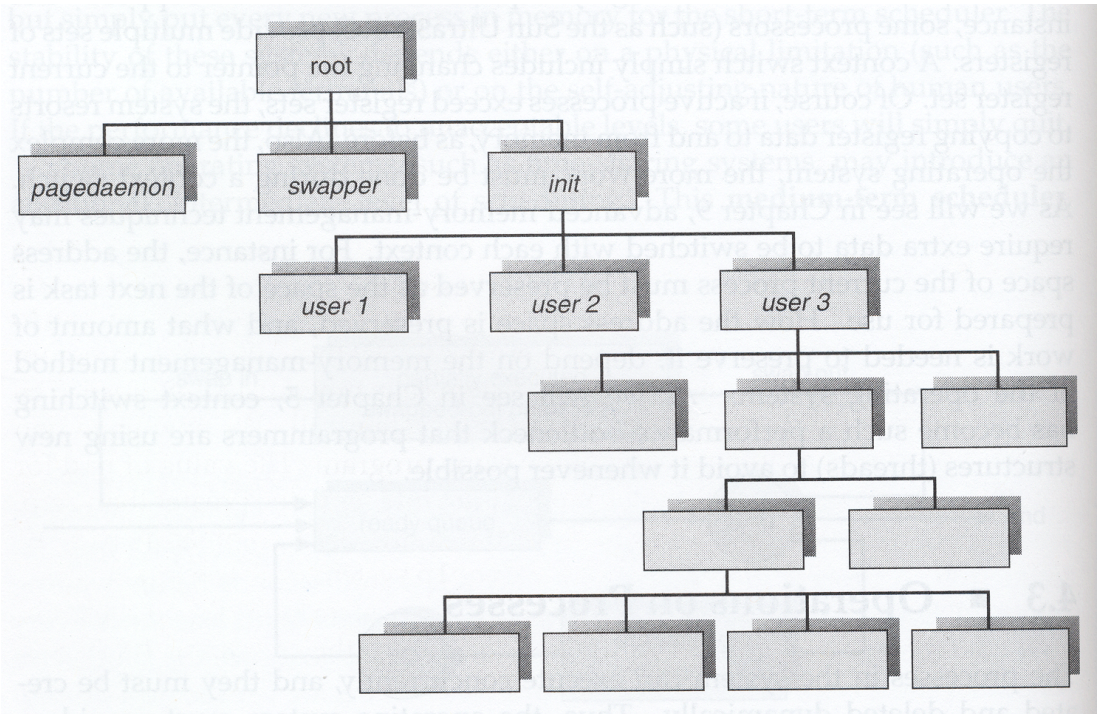

Abbildung 24: Prozesshierarchie

3.2.3.5 Beispiel Unix: exec()

Lädt ein neues Prozess-Image im aktuellen Prozess:

- Ersetzt das aktuelle Programm vollständig
- Das neue Programm startet von Null
- exec() kehrt nicht zurück; Ausnahme: im Fehler (neues Programm nicht gefunden, keine ausreichenden Rechte, zu wenig Speicher) gibt exec() einen Fehlercode zurück.

Ausschnitt aus POSIX-Standard 1003.1-2001:

NAME

environ, execl, execv, execle, execve, execlp, execvp - execute a file

SYNOPSIS

```
#include <unistd.h>

extern char **environ;
int execl(const char *path, const char *arg0, ... /*, (char *)0 */);
int execv(const char *path, char *const argv[]);
int execle(const char *path, const char *arg0, ... /*,
       (char *)0, char *const envp[]*/);
int execve(const char *path, char *const argv[], char *const envp[]);
int execlp(const char *file, const char *arg0, ... /*, (char *)0 */);
int execvp(const char *file, char *const argv[]);
```

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

DESCRIPTION
The *exec* family of functions shall replace the current process image with a new process image. The new image shall be constructed from a regular, executable file called the *new process image file*. There shall be no return from a successful *exec*, because the calling process image is overlaid by the new process image.

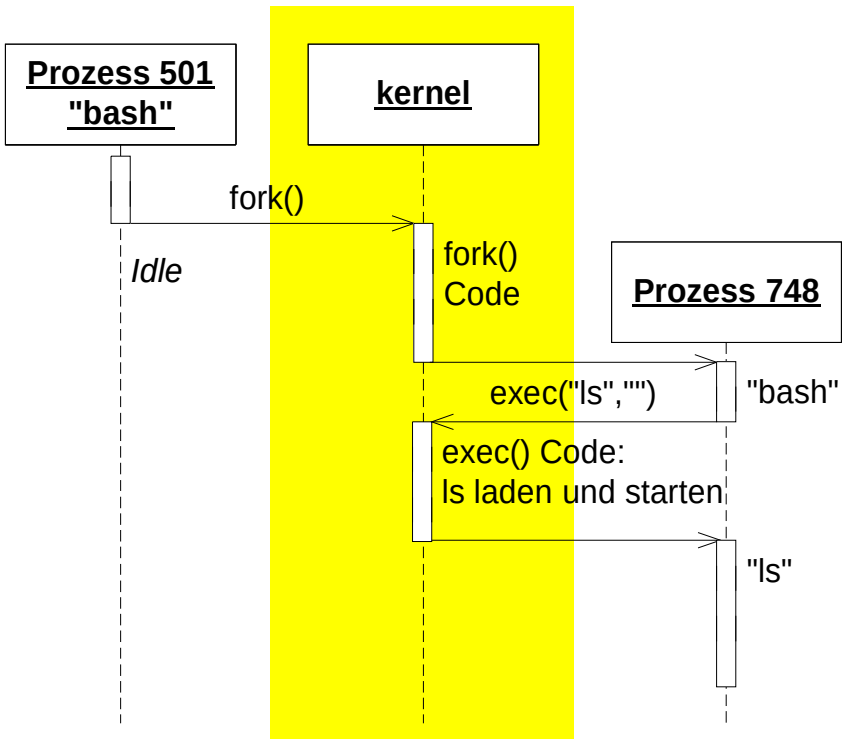

Abbildung 25: Ablauf fork() - exec()

3.2.3.6 Beispiel Win32: CreateProcess()

Es wird immer ein neues Programm ausgeführt
Parameter von CreateProcess():

- Name des Programms
- Kommandozeilenparameter
- Prozess- und Thread-Security Beschreibungen
- Vererbungsflag: sollen alle Handles vererbt werden? Ja -> wie fork
- Spezialeinstellungen
- Neues Environment,
- Aktuelles Verzeichnis
- Startup-Informationen für das Fenster-System

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

CreateProcess

The **CreateProcess** function creates a new process and its primary thread. The new process runs the specified executable file.

To create a process that runs in a different security context, use the CreateProcessAsUser or CreateProcessWithLogonW function.

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,                // name of executable module
  LPTSTR lpCommandLine,                     // command line string
  LPSECURITY_ATTRIBUTES lpProcessAttributes, // SD
  LPSECURITY_ATTRIBUTES lpThreadAttributes,  // SD
  BOOL bInheritHandles,                     // handle inheritance option
  DWORD dwCreationFlags,                    // creation flags
  LPVOID lpEnvironment,                     // new environment block
  LPCTSTR lpCurrentDirectory,               // current directory name
  LPSTARTUPINFO lpStartupInfo,              // startup information
  LPPROCESS_INFORMATION lpProcessInformation // process information
);
```

3.3 Kontextwechsel

Funktionsaufruf, Interrupt (Systemaufruf), Kontextwechsel

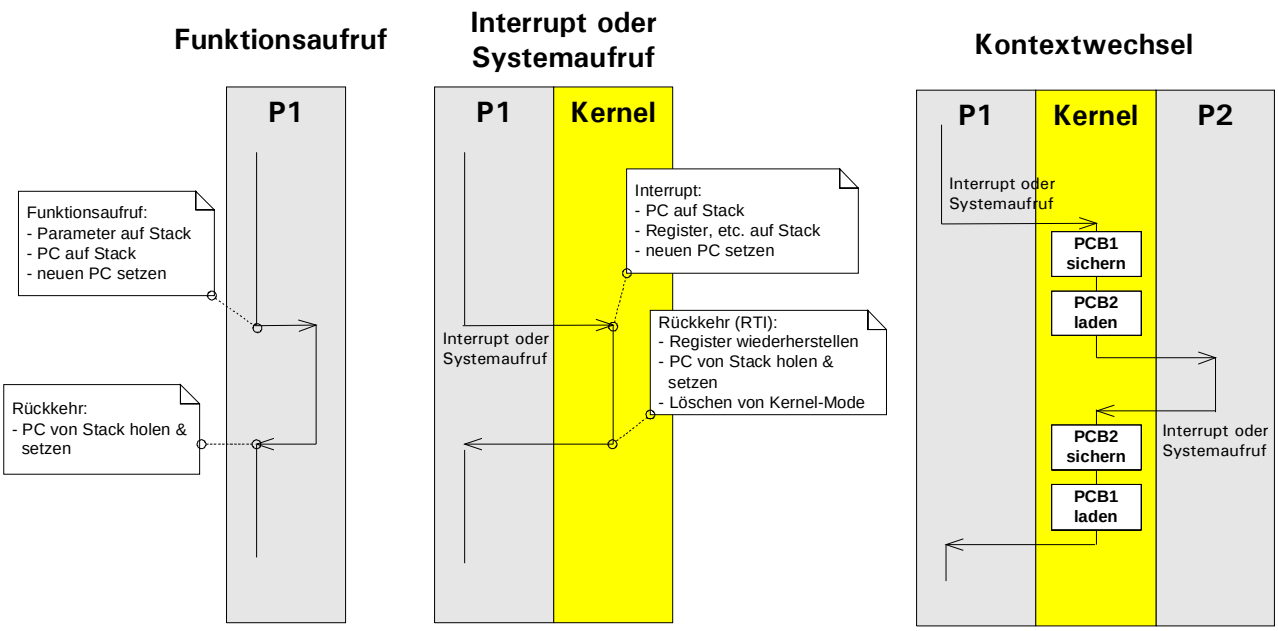

Abbildung 26: Funktionsaufruf, Interrupt, Prozesswechsel

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

3.3.1 Funktionsaufruf

- Auslöser ist der Prozess (das Programm) selbst
- Aufruf:
    - Parameter auf Stack legen
    - Befehlszähler auf Stack legen
    - Neuen Befehlszähler laden
- Rückkehr:

3.3.2 Alten Befehlszähler vom Stack ladenInterrupt

- Auslöser von I/O Gerät (z.B. Festplattenkontroller, Timer) oder Prozess selbst (System-Trap für Kernel-Aufruf)
- Aufruf:
    - Befehlszähler auf Stack legen
    - Neuen Befehlszähler laden
    - Kernel-Mode setzen
- Rückkehr (Return from Interrupt, RTI):
    - Interrupt-Flag, Kernel-Mode zurücksetzen
    - Alten Befehlszähler vom Stack laden

3.3.3 Prozesswechsel

- Interrupt in den Kernel
    - Auslöser von I/O Gerät (z.B. Festplattenkontroller, Timer) oder Prozess selbst (System-Trap für Kernel-Aufruf)
- Scheduler entscheidet, jetzt Prozess 2 laufen zu lassen
    - Prozess 1 geht von RUN -> READY oder WAITING über
- Zustand (PCB) von Prozess 1 sichern
- PCB P2 laden:
    - Prozessor-Spezialregister
    - Daten- und Adressregister
    - Speicherverwaltung
    - I/O Verwaltung
- Rücksprungadresse auf Stack manipulieren
- Rücksprung (RTI) aus Interrupt in den neuen Prozess

➔ Kontextwechsel sehr zeitaufwändig
➔ HW Unterstützung möglich, Wechselregistersätze, etc.

3.3.4 Unterbrechungsarten

(übersprungen)
Externe Unterbrechungen

- *Geräte-Interrupts:*
    - *Fertig-Meldungen von I/O Steuerungen / Geräten*
    - *Alarme von Spezialperipherie*
    - *Konsol-Eingriffe (z.B. Ctrl-Alt-Del)*

Interne Unterbrechungen

- *Fehlersignale*

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

- *Verknüpfungsfehler, z.B. Division durch Null*
- *Privileg-Verstöße*
- *Unbekannter Befehl*
- *Adressierungsfehler*

- *Durch Spezialbefehle erzeugt*
  - *BT-Aufrufe (Trap)*
  - *Befehlssimulation*
- *Debug-Interrupts*
  - *Einzelschrittmodus: jeder Befehl löst einen Interrupt aus*
  - *Sprungmodus: bei bestimmten Sprüngen (z.B. Rücksprung) wird ein IR ausgelöst*
  - *Zugriffsmodus: bei Zugriff auf bestimmten Speicher wird ein IR ausgelöst*

Interprozessor-Signale

- *Im Mehrprozessorsystem von fremdem Prozessor per Spezialbefehl ausgelöst*

***Interrupt-Verarbeitung**:*
Je nach Prozessor verschiedene Bearbeitungen möglich:

- *Nur ein Interrupt möglich: alle weiteren Interrupts müssen warten bis der aktuelle fertig ist*
  - *Nachteil: hochpriore, „eilige" Interrupts können verzögert werden*
- *Mehrere, geschachtelte Interrupts möglich: Rettung der Daten auf dem Stack*
  - *Vorteil: hochpriore IR werden bevorzugt behandelt*
  - *Nachteil: erhöhter Stackbedarf*
- *Spezialregister teilweise mehrfach vorhanden für verschiedene Interrupts*
- *Prioritisierung von Interrupts*
- *Maskierung von Interrupts*

3.4 Interprozess-Kommunikation

Prozesse:
• eigenständige, abgekapselte Einheiten im Betriebssystem
-> keine Kommunikation zwischen den Prozessen möglich!
Man möchte jedoch eine Zusammenarbeit zwischen verschiedenen Prozessen haben!

3.4.1 Message Passing

Mindestens zwei Funktionen:

- `send(message)`
- `receive(message)`

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

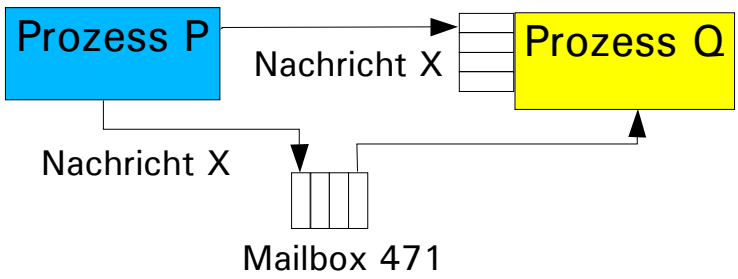

Abbildung 28: Message Passing
Abbildung 27: Schema Message Passing

Verschiedene Möglichkeiten:
- direkte oder indirekte Kommunikation
- synmmetrisch oder asymmetrisch
- automatische oder explizite Pufferung
- Send-by-Copy oder Send-By-Reference
- Feste Länge der Messages oder variabel

3.4.1.1 Direkte Kommunikation

- Eins-zu-eins Kommunikation: Für jedes Sender / Empfänger Paar gibt es genau einen Kommunikationskanal.
- Ein sendender Prozess muss explizit den Empfänger-Prozess angeben.
- Symmetrisch: Sender und Empfänger kennen sich:
  - `send(P, message) – schicke Nachricht an P`
  - `receive(Q, message) – hole Nachricht`
- Asymmetrisch: Sender kennt Empfänger, Empfänger jedoch nicht unbedingt den Sender:
  - `send(P, message) – schicke Nachricht an P`
  - `receive(sender_id, message) – hole Nachricht (asymmetrisch)`

Nachteil:
- fehlende Modularität, da der Sender den Empfänger kennen muss

3.4.1.2 Indirekte Kommunikation

- Nachrichten werden an Mailboxen (oder Ports) geschickt:
  - `send(A, message) – A ist eine Mailbox`
  - `receive(A, message)`
- Mehr als 1 Sender oder 1 Empfänger möglich
- Mehrere verschiedene Kommunikationskanäle zwischen zwei Prozessen möglich

3.4.1.3 Synchronisierung

Sender:
- blockierend (synchron): Sender wartet, bis die Nachricht verarbeitet wurde
- nicht-blockierend (asynchron): Sender schickt die Nachricht und rechnet sofort weiter; die Nachricht muss vom OS gepuffert werden

Empfänger:
- blockierend: falls keine Nachricht da ist, wartet der Empfänger solange bis eine kommt

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

- nicht-blockierend: Empfänger schaut nach, ob eine da ist; falls ja, wird diese geholt, falls nicht, so kommt eine entsprechende Fehlermeldung zurück

3.4.1.4 Pufferung

- Keine Pufferung (nur synchrone Zustellung möglich)
- Feste Puffergrösse: einfach, Nachteil: Puffer kann überlaufen
- Variable Puffergrösse: aufwändig zu implementieren

3.4.1.5 Beispiel: Mach

Verwendet MP zur Interprozess-Kommunikation und zur Kommunikation von Prozessen mit dem Kernel: alle System-Calls werden über MP realisiert

- Jede Task (Prozess) hat zwei spezielle Mailboxen: *Kernel* und *Notify*
- Über *Kernel* kann die Task messages an den Kernel schicken
- Sie wird über *Notify* vom Kernel über die Ergebnisse benachrichtigt

3.4.2 Pipes

Unix-Manual zu Pipes:

PIPE
Section: Linux Programmer's Manual (2)
NAME
pipe - create pipe
SYNOPSIS
#include <unistd.h>

int pipe(int filedes[2]);
DESCRIPTION
pipe creates a pair of file descriptors, pointing to a pipe inode, and places them in the array pointed to by filedes. filedes[0] is for reading, filedes[1] is for writing.
RETURN VALUE
On success, zero is returned. On error, -1 is returned, and errno is set appropriately.
SEE ALSO
read(2), write(2), fork(2), socketpair(2)

Dient zum Datenaustausch mit einem anderen Prozess

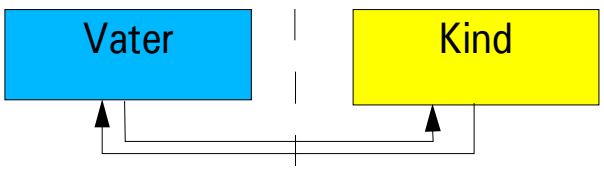

Abbildung 29: Schema Pipes

- Mittels read() und write() werden Daten an den anderen Prozess übergeben

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

  - read() liest eine Anzahl von Daten von der Pipe
  - write() schreibt eine Anzahl von Daten in die Pipe
- Kommunikation typischerweise zwischen Vater und Kind
  - Vater legt pipes an
  - Nach fork() stehen diese auch dem Kind zur Verfügung

```
int a_pipe[2];
int res=pipe(a_pipe);
if( fork()==0 ) {
      // child: read from pipe 0, write to pipe 1
      ...
} else {
      // parent: read from pipe 1, write to pipe 0
      ...
}
```

3.4.3 Sockets

Ein Socket ist definiert als Endpunkt für eine Kommunikation, basiert auf TCP/IP Netzwerkprotokoll.

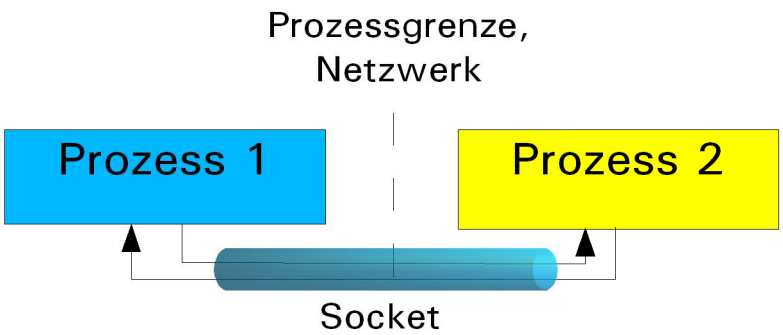

Abbildung 30: Schema Sockets

- Ähnlich wie Pipes, ein Prozess kann Daten in ein Socket schreiben, ein zweiter kann Daten aus dem Socket lesen
  - 1 Socket ist jedoch bi-direktional: beide Seiten können der jeweils anderen etwas schicken
- Client-Server-Verhalten
  - Der Server öffnet ein Socket als Zuhörer (Listener) und wartet auf Verbindungswünsche von den Clients
  - Der Client baut eine Verbindung zum Server auf und schickt diesem die Daten zu
  - Der Server verarbeitet die Daten und schickt eine Antwort zurück
- Im Unterschied zu Pipes auch Netzwerk-fähig
  - Ein Socket wird dynamisch aufgebaut
  - Der Client spezifiziert den gewünschten Server anhand der IP-Adresse und der Port-Nummer, z.B. 192.168.100.11:23 (telnet-Port)
  - Auf einem Rechner können mehrere Server laufen; diese unterscheiden sich anhand der Port-Nummer

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

- Reservierte (bekannte) Portnummer <1024, z.B.
  - Port 23: telnet
  - Port 21: ftp
  - Port 80: http
- Über 1024: frei wählbare Portnummern
- auch zwischen Prozessen auf demselben Rechner möglich; IP-Nummer 127.0.0.1 (localhost)

Beispiel:
- Web-Server

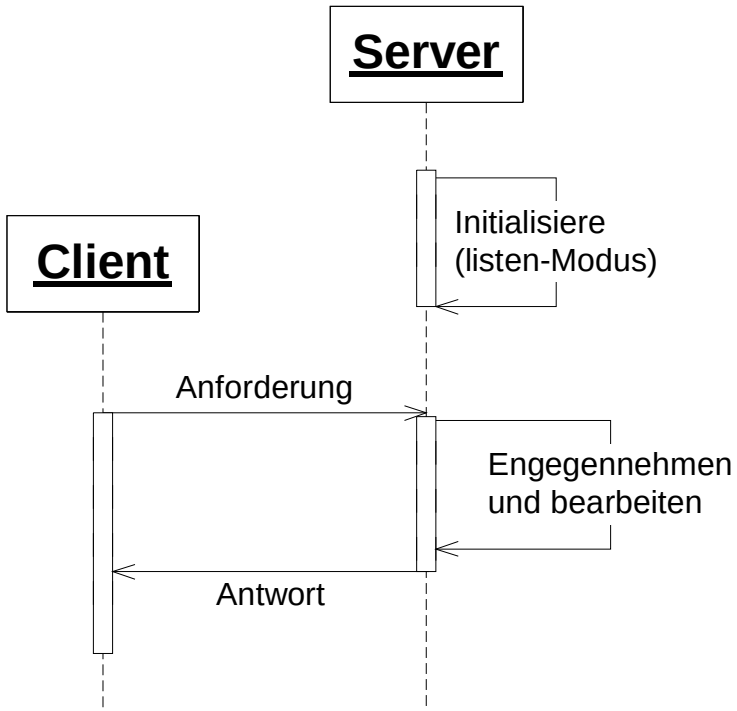

Abbildung 31: Client-Server mit Sockets

3.4.4 Client-Server: RPC

Remote Procedure Call: Funktionsaufrufe in andere Prozesse oder auf andere Rechner

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

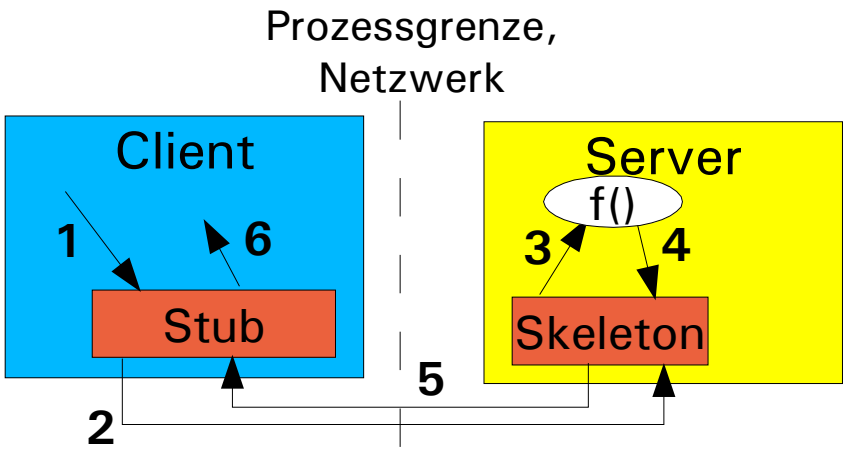

Abbildung 32: Schema RPC Aufruf

- Notation gemäß Java-RMI; Microsoft nennt den Stub Proxy, und das Skeleton Stub...
- Adressierung: wie Sockets über IP-Nummer / Portnummer; basiert auf Sockets
- Ablauf:
    1. Client ruft Funktion auf; wird von Stub entgegen genommen
    2. Stub verpackt Funktionsname und Parameter in einen Datenstrom und schickt diesen an den Server
        - Adresse des Servers (IP + Port)
        - Funktionsname
        - Parametern
    3. Skeleton nimmt Datenstrom entgegen, packt Daten aus und ruft die Funktion auf
    4. Skeleton nimmt das Ergebnis entgegen, erzeugt davon einen Datenstrom
    5. und schickt diesen an den Stub zurück
    6. Dieser packt das Ergebnis aus und gibt es an das aufrufende Programm zurück (es war solange blockiert)
- Einheitliche Datenrepresentation nötig
    - **external data representation** (XDR) ist Standard
    - muss unterschiedliche Prozessorarchitekturen berücksichtigen (Bigendian vs. Littleendian)
    - das Verpacken der Parameter nennt man **marshalling**
- Client-Server Binding:
    - feste Bindung (zur Compile-Zeit) oder
    - dynamische Bindung: ein OS-Programm vermittelt zwischen Client und Server
- Probleme mit RPC:
    - sehr lange im Vergleich zu "einfachem" Funktionsaufruf:
        - direkter Funktionsaufruf einige ns
        - Funktionsaufruf von einem Prozess zu einem anderen (auf demselben Rechner): mindestens einige µs (Faktor 1000 größer!)
        - Netzwerk-Funktionsaufruf: mindestens einige ms (Faktor 1.000.000 größer!)
    - Funktionsaufrufe können fehlschlagen
        - Netzwerkfehler, Server nicht bereit oder ausgelastet, etc.

Dr. Ulrich Margull
Kirchweg 6 * 93055 Regensburg
Tel. (+49) 941 78956-8 Fax –9
www.margull.de

    - Mehrfache Ausführung möglich
- Ähnliches Protokoll auf XML/HTTP basierend: Web-Services

3.4.5 Verteilte Objekte

-> wird in "Verteilte Systeme" nochmals behandelt
RPC: der Server stellt Funktionen zur Verfügung (strukturierte Programmierung)
Objektorientierte Ansätze: der Server stellt *Objekte* zur Verfügung:

- Zustandsbehaftet: der Server erzeugt ein Objekt, das ausschließlich einem Client gehört; damit ist zum Beispiel möglich:
    - Client "bestellt" Objekt *o* bei Server
    - Client: *o.init(p1, p2);*
    - Client: *o.tu_etwas(p3);*
    - Client gibt Objekt wieder frei: *o.bin_fertig();*
    - Das Objekt wird an einen Client gebunden
- Zustandslos: das Objekt hat keinen Zustand, d.h. es ist nicht an einen Client gebunden; bessere Performance am Server (Beispiel: HTTP Protokoll ist zustandslos)
- Stub/Skeleton für das ganze Objekt
- Objekte finden & erzeugen: Object Request Brokers (ORB)
    - Client stellt Anfrage an ORB, z.B. mit Namen des gewünschten Server-Objektes
    - ORB liefert Referenz auf das Objekt zurück
    - Ggfs. Erzeugt der ORB das Objekt

Techniken:

- Microsoft COM/DCOM
- Java RMI
- CORBA

3.4.5.1 Komponenten-Transaktions-Server (CTS)

Komponenten-Transaktions-Server stellen eine Laufzeitumgebung für Komponenten zur Verfügung

- Erzeugung / Vernichtung von Komponenten
- Transaktionen
- Sicherheit
- Persistenz
- Konfigurierbarkeit
- u.v.m.

Beispiele:

- Microsoft COM+ Services (früher MTS)
- Enterprise Java Beans (EJB), von SUN, IBM, Bea, u.v.a.