HOWTO zu UNIX-Prozess-Schnittstellen

(C) 2007-2017 T.Birnthaler/H.Gottschalk <howtos(at)ostc.de>
OSTC Open Source Training and Consulting GmbH
http://www.ostc.de

$Id: unix-process-interfaces-HOWTO.txt,v 1.32 2018/02/17 11:44:07 tsbirn Exp $

Dieses Dokument beschreibt die Schnittstellen von UNIX-Prozessen.

INHALTSVERZEICHNIS

---

1) Einleitung

Ein UNIX-Prozess kennt viele Schnittstellen, über die er Daten mit seiner Umgebung (das ist sein Elternprozess, seine Kindprozesse sowie alle anderen Prozesse) austauschen kann.

Die Schnittstellen unterscheiden sich in der RICHTUNG (uni/bidirektional) und in der DATENMENGE, die über sie transferiert werden kann:

  * "Dünn" (nur EIN Wert übergebbar) ----
  * "Dick" (mehr als ein Wert übergebbar) ====
  * "Unidirektional" (nur EINE Richtung) ---> ===>
  * "Bidirektional" (ZWEI Richtungen) <--> <==>

1.1) Automatische Schnittstellen

Der 1. Teil umfasst die AUTOMATISCH vorhandenen Standard-Schnittstellen, die von den Prozessen sofort (ohne sie zu "öffnen") benutzt werden können. Oft werden dies Schnittstellen sogar "stillschweigend" genutzt, ohne dass dies großartig auffällt (IN=Eingabe, OUT=Ausgabe, "/"=entweder-oder, "+"=gleichzeitig):

Name	Art	Datentyp
1) Programmname	IN	Ein Wort
2) Aufrufargumente	IN/OUT	Liste von Worten
3) Umgebungsvariablen	IN/OUT	Paare der Form VAR=TEXT
4a) Standard-Eingabekanal (STDIN)	IN	Byte-Strom (Text/Daten)
4b) Standard-Ausgabekanal (STDOUT)	OUT	Byte-Strom (Text/Daten)
4c) Standard-Fehlerkanal (STDERR)	OUT	Byte-Strom (Text/Daten)
5) Exitstatus	IN/OUT	Eine Zahl (0-255)
6) Eltern/Kindprozess-PID	IN	Zahl(en) (0-65Tsd/2Mrd)
7) Signale	IN+OUT	Eine Zahl (0-31/63)
8) Konfigurationsdateien	IN	Byte-Strom (Text/Daten)

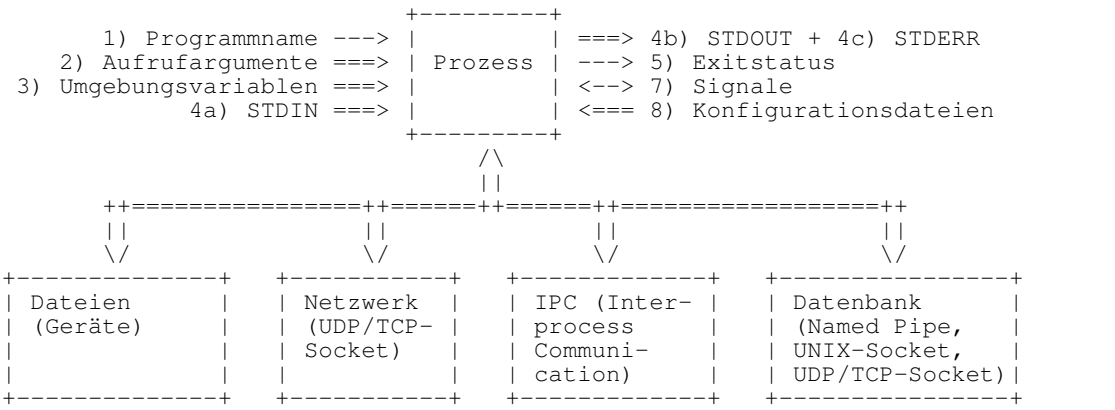
1.2) Manuelle Schnittstellen

Der 2. Teil der Schnittstellen muss im Programm explizit GEÖFFNET werden, d.h. im Prozess muss durch einen "open"- oder "connect"-Aufruf explizit eine Verbindung dafür hergestellt werden (IN=Eingabe, OUT=Ausgabe, "+"=gleichzeitig):

Name	Art	Datentyp
9) Dateien (Geräte)	IN+OUT	Byte-Strom (Text/Daten)
10) Netzwerk (UDP/TCP-Socket)	IN+OUT	Byte-Strom (Text/Daten)
11) IPC (Interprocess Communication)	IN+OUT	Byte-Strom (Text/Daten)
Named Pipe (FIFO)	IN+OUT	Byte-Strom (Text/Daten)
Locking	IN+OUT	Boolean (Text/Daten)
Semaphor	IN+OUT	Zahlen (Text/Daten)
Message Queue	IN+OUT	Byte-Strom (Text/Daten)
Shared Memory	IN+OUT	Byte-Strom (Text/Daten)
Memory Mapped Files	IN+OUT	Byte-Strom (Text/Daten)
UNIX-Socket	IN+OUT	Byte-Strom (Text/Daten)
12) Datenbank (UDP/TCP-Socket, Named	IN+OUT	Byte-Strom (Text/Daten)

1.3) Übersicht

Alle prinzipiell möglichen Schnittstellen sind in der folgenden Grafik dargestellt (die automatischen im oberen Bereich, die manuellen im unteren):

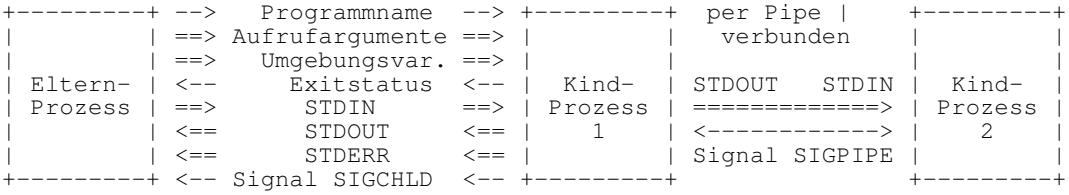


1.4) Unidirektionale Schnittstellen

Folgende Schnittstellen bieten eine Datenübergabe nur in EINER Richtung und sind beschränkt auf bestimmte Prozess-Zusammenhänge:

Schnittstelle	Beschränkung/Richtung
Programmname	Eltern --> Kind
Aufrufargumente	Eltern ==> Kind
Umgebungsvariablen	Eltern ==> Kind
Exitstatus	Eltern <-- Kind
STDIN	Eltern ==> Kind #=> Geschwister
STDOUT	Eltern <== Kind + Geschwister ==#
STDERR	Eltern <== Kind
Signale	Eltern <-- Kind + Geschwister <--> Geschwister

In grafischer Form:



2) Die Standard-Schnittstellen

2.1) Programmname

Jedem Prozess ist der Programmname bekannt, über den er gestartet wurde. Dieser Programmname ist Teil der Aufrufargumente und zwar das 0-te Argument. Normalerweise würde man erwarten, dass der Name eines Programms nur durch Umbenennen veränderbar ist (was wenig sinnvoll erscheint). Unter UNIX können allerdings in Form von Hardlinks und Symbolischen Links beliebig viele weitere Namen für ein Programm vergeben werden (per "ln" bzw. "ln -s"). Der Programmstart über einen Link führt dazu, dass der Programmname nicht mehr dem

```
eigentlichen Programmnamen entspricht, sondern dem Linknamen.

  skript.sh       # Skript-Aufruf
  echo $0         # Programmname im Skript ausgeben --> .../skript.sh

Über diese "Argument" wird häufig das Grundverhalten des Kommandos gesteuert.
Beispielsweise hat das Programm "bzip2" die zusätzlichen Namen "bunzip2" und
"bzcat". Ein weiteres typisches Beispiel ist das LVM-System (Logical Volume
Management), das ein einziges Hauptprogramm names "lvm" und 40 Unterkommandos
(namens "pv/vg/lv...") kennt, die alle als symbolische Links auf dieses
Hauptprogramm realisiert sind und das grundsätzliche Verhalten des
Hauptprogramms steuern.

  ln skript.sh hl.sh      # Hardlink auf Skriptdatei erstellen
  ln -s skript.sh sl.sh   # Symbolischen Link auf Skriptdatei erstellen
  hl.sh                   # Skript-Aufruf über Hardlink
  echo $0                 # Programmname im Skript ausgeben --> .../hl.sh
  sl.sh                   # Skript-Aufruf über Softlink
  echo $0                 # Programmname im Skript ausgeben --> .../sl.sh

Beispiel für die Abfrage des verwendeten Skriptnamens und unterschiedliche
Reaktionen darauf im Skript:

  case $(basename $0) in      # Reinen Programmnamen extrahieren (ohne Pfad)
      *skript.sh) echo "Per Name 'skript.sh' aufgerufen" ;;
      *hl.sh)     echo "Per Name 'hl.sh' aufgerufen" ;;
      *sl.sh)     echo "Per Name 'sl.sh' aufgerufen" ;;
      *)          echo "Per Name '$0' aufgerufen" ;;
  esac


2.2) Aufrufargumente
--------------------
Umfassen alle beim Aufruf eines Programms hinter dem Programmnamen auf der
Kommandozeile angegebenen Optionen (kurze "-x" und lange "--xxx") und Argumente
(Datei/Verz.namen, sonstige Parameter). Die erlaubte Maximallänge dieser Liste
ist sehr groß, aber nicht unbeschränkt (etwa 100.000-2 Mio Zeichen). Der
Programmname und die einzelnen Aufrufargumente werden durch "Whitespace"
(Leerzeichen, Tabulator, Newline) voneinander getrennt (--> Quotierung).
Beispiel sind die vielen Optionen des Befehls "ls" und der Quell- und
Zieldateiname des Befehls "mv".

  skript.sh a bb ccc 1234   # Skript-Aufruf mit 4 Argumenten
  echo "1. Argument: $1"    # Im Skript --> a
  echo "2. Argument: $2"    # Im Skript --> bb
  echo "3. Argument: $3"    # Im Skript --> ccc
  echo "4. Argument: $4"    # Im Skript --> dddd
  echo "Anz. Argum.: $#"    # Im Skript --> 4
  echo "Alle Argum.: $@"    # Im Skript --> a bb ccc dddd
  echo "Alle Argum.: $*"    # Im Skript --> a bb ccc dddd

HINWEIS: Leerzeichen trennen Argumente, das macht die Verwendung von Dateinamen
mit Leerzeichen umständlich (müssen mit "...", '...' oder \. quotiert werden).

Angabe von Aufrufparametern beim Programmaufruf:

  KMDO PARAM1 PARAM2 ...   # Aufruf eines Kommandos mit Parametern

Zugriff auf Aufrufparameter in der Shell:

  shift                   # 1. Aufrufparameter entfernen, restliche nachrücken
  shift 4                 # Die ersten 4 Aufrufparameter entfernen, restliche n.


2.3) Umgebungsvariablen
-----------------------
Jeder Prozess hat einen Umgebungsbereich (Environment), in dem er Variablen +
ihre Werte hinzufügen, abfragen, ändern und löschen kann. Diese Variablen
werden meistens GROSS geschrieben. Die Größe dieses Bereiches ist unbeschränkt,
typischerweise stehen darin etwa 100-200 Variablen.

Diese Liste von Umgebungsvariablen mit ihren Werten wird von jedem Prozess beim
Start eines Kindprozesses an diesen vererbt, indem beim Start eine KOPIE der
Liste des Elternprozesses erstellt und dem Kindprozess zugeordnet wird.

Jeder Prozess hat nur auf seinen EIGENEN Umgebungsbereich Zugriff, er kann
keine Variablen+Werte im Umgebungsbereich anderer Prozesse ändern. Am Ende
eines Prozesses verschwindet sein Umgebungsbereich mit den eigenen Variablen.

Beispiel ist die Variable HOME, die den Befehl "cd" beeinflusst, die Variable
PATH zur Steuerung der Suche nach eingetippten Kommandos und die Variablen
LANGUAGE, LANG, LC_ALL, LC_MESSAGE, LC_... mit denen Spracheinstellung vieler
```

```
Programme festlegt werden.

Umgebungsvariablen in der Shell anlegen:

  export VAR          # Umgebungsvariable definieren (exportieren)
  export VAR=TEXT     # Umgebungsvariable mit Wert belegen (exportieren)
  KMDO                # Kommando starten, Umgebungsvariablen werden kopiert
  VAR=TEXT ... KMDO   # Temporäre Umgeb.var. mit Wert belegen vor Kmdostart
  VAR=                # Wert einer Umgebungsvariablen löschen --> Var. leer
  unset VAR           # Umgebungsvariable entfernen --> Var. undefiniert

Zugriff auf Umgebungsvariablen in der Shell:

  echo $VAR    # Wert einer Umgebungsvariablen ausgeben
  env          # Alle aktuelle gesetzten Umgebungsvariablen auflisten
  env | sort   # ... alphabetisch sortiert auflisten

ACHTUNG: Umgebungsvariablen nicht mit Shellvariablen verwechseln! Diese sind
nur für die Shell selbst relevant und steuern deren Verhalten oder werden von
ausschliesslich von ihr verwendet, sie werden nicht an andere Prozesse
weitergereicht.


2.4) Standard-Ein/Ausgabekanäle (STDIN/STDOUT/STDERR)
-----------------------------------------------------
Über die Standard-Ein/Ausgabekanäle kann jeder Prozess Daten mit anderen
Prozessen austauschen (meist mit der Shell). In der Regel handelt es sich dabei
um zeilenorientierte ASCII-Texte, aber auch Binärdaten können übertragen werden.
Sie eignen sich auch für die effizente Übergabe sehr großer Datenmengen. Die
3 Kanäle haben feste Namen (STDIN, STDOUT, STDERR) und feste Nummern (0, 1, 2):

                           +---------+
              (0) STDIN ==> | Prozess | ==> STDOUT (1)
        (Std: Tastatur)     +---------+     (Std: Bildschirm)
                                 ||
                                 \/
                              STDERR (2)
                           (Std: Bildschirm)

Ein Prozess behandelt die 3 Kanäle wie bereits geöffnete sequentielle Dateien,
aus denen er ausschließlich lesen oder in die er ausschließlich schreiben kann.
D.h. darin neu positionieren ist nicht möglich, bereits gelesene Daten können
nicht erneut gelesen und bereits geschriebene Daten können nachträglich nicht
mehr verändert werden. Es kann auch nicht gleichzeitig per Umlenkung aus einer
Datei gelesen und in sie geschrieben werden.

Diese Kanäle bilden die Grundlage der Dateiumlenkung und des Pipemechanismus
unter UNIX. Standardmäßig ist STDIN mit der Tastatur verbunden und STDOUT sowie
STDERR mit dem Bildschirm. Die aktuelle Belegung der drei Kanäle wird von einem
Elternprozess (meist die Shell) an seine Kindprozesse vererbt. EINMAL beim
Start eines Prozesses sind die Kanäle auch auf andere Dateien/Geräte umlenkbar.
In der Shell geschieht diese Umlenkung durch folgende Anweisungen:

  KMDO < DATEI      # Eingaben auf STDIN von DATEI lesen
  KMDO > DATEI      # Ausgaben auf STDOUT auf DATEI schreiben
  KMDO 2> DATEI     # Ausgaben auf STDERR auf DATEI schreiben
  KMDO1 | KMDO2     # STDOUT von KMDO1 mit STDIN von KMDO2 verbinden

Mehrere Kommandos lassen sich über ihre Standard-Ein/Ausgabekanäle leicht zu
einer "Filterkette" oder "Pipeline" zusammensetzen, indem per Pipe-Symbol "|"
die STDOUT- und die STDIN-Kanäle der Kommandos miteinander verbunden werden.

                  KMDO1      |      KMDO2      |      KMDO3
                +----------+     +----------+     +----------+
     STDIN ==>  | Prozess1 | ==> | Prozess2 | ==> | Prozess3 | ==> STDOUT
  (Tastatur)    +----------+     +----------+     +----------+     (Bildschirm)
                     ||               ||               ||
                     \/               \/               \/
                  STDERR           STDERR           STDERR
                (Bildschirm)     (Bildschirm)     (Bildschirm)

Die Fehlerkanäle STDERR der verbundenen Prozesse werden nicht mit in den
Datenstrom eingeschleust, sondern bleiben getrennt pro Prozess --- sie werden
also normalerweise auf dem Bildschirm ausgegeben. Sollen die Fehlermeldungen an
eine Prozess weitergereicht oder auf Datei geschrieben werden, so ist pro
Kommando eine zusätzliche Dateiumlenkung für STDERR anzugeben (2> lenkt
Fehlerkanal auf Datei um, 2>&1 fügt ihn zu Kanal 1 hinzu).

  KMDO1 < INPUT 2> ERR1 | KMDO2 2> ERR2 | KMDO3 > OUTPUT 2> ERR3
  KMDO1 < INPUT 2>&1    | KMDO2 2>&1    | KMDO3 > OUTPUT 2>&1

Zugriff auf Standard-Ein/Augabekanäle in der Shell:
```

```
  read TEXT            # Text von Standard-Eingabe einlesen (stdin, 0)
  echo "Ausgabe"       # Text auf Standard-Ausgabe ausgeben (stdout, 1)
  echo "Fehler" 1>&2   # Text auf Fehler-Ausgabe ausgeben (stderr, 2)
```

2.5) Exitstatus

Wird von jedem Prozess am Ende an seinen Elternprozess zurückgegeben. Wert "0" bedeutet, dass der Prozess ERFOLGREICH ausgeführt wurde. Ein Wert ungleich "0" bedeutet, dass ein FEHLER auftrat. Falls von einem Programm verschiedene Exitstati für verschiedene Fehlerarten zurückgegeben werden, dann werden sie in der zugehörigen man-Page beschrieben.

Der Exitstatus kann vom aufrufenden Programm (Elternprozess, meist die Shell) abgefragt und abhängig davon unterschiedlich reagiert werden. Existiert sein Elternprozess nicht mehr, dann bleibt ein Kindprozess so lange als "Zombie" am Leben, bis er vom Betriebssystem mit dem "init//systemdsystemd"-Prozess verbunden wird und dort seinen Exitstatus abliefern kann.

Erzeugen eines Exitstatus in der Shell:

```
  exit      # Exitstatus des zuletzt ausgeführten Kommandos zurückgeben
  exit 0    # Exitstatus 0 zurückgeben
  exit 1    # Exitstatus 1 zurückgeben
```

Zugriff auf Exitstatus in der Shell:

```
  KMDO      # Kommando aufrufen --> liefert Exitstatus zurück
  echo $?   # Exitstatus des vorherigen Kommandos abfragen (geht nur 1x!)
  if KMDO   # Exitstatus von Kommando abfragen und für Verzweigung nutzen
  then
      print "Exitstatus $? == 0 --> alles OK!"
  else
      print "Exitstatus $? != 0 --> irgendein Fehler!"
  fi
```

2.6) Eltern/Kindprozess-PID

Jeder Kindprozess kennt die PID seines Elternprozesses (PPID = parent process id) und der Elternprozess kennt die PIDs seiner Kindprozesse (child processes). Die Kenntnis der PID ist zum gezielten Verschicken von Signalen notwendig.

```
  echo $$      # Eigene Prozess-ID
  echo $PPID   # Elternprozess-ID
  CMD &        # Kommando im Hintergrund starten (Kind-Prozess)
  echo $!      # Prozess-ID des vorher gestarteten Hintergrundkommandos
```

2.7) Signale

Ein Prozess kann einen festen Satz von Signalen (Nachricht bestehend aus einer Nummer von 0-31/63) von anderen Prozessen empfangen bzw. an andere Prozesse senden. Über die (effektive) UID/GID von Sende- und Empfangsprozess und über den "Signal-Handler" des Empfangsprozesses wird geregelt, ob ein Prozess darauf reagiert oder nicht. Ein Signal kann im empfangenden Prozess abhängig von der Signal-Nummer bestimmte Reaktionen auslösen (z.B. liest SIGHUP oft die Konfigurationsdatei neu ein), ihn abbrechen (z.B. per SIGKILL), es kann aber auch ignoriert werden. Es gibt 32 bzw. 64 Signale, die wichtigsten sind:

Name	Nr	Bedeutung	
SIGEXIT	0	Bash: Am Skriptende (exit) ausgelöst	
SIGDEBUG	?	Bash: Nach jedem "einfachen Kmdo" ausgelöst	
SIGRETURN	?	Bash: Nach jedem Shell-Funktionsaufruf + ./source-Kmdo	
SIGERR	?	Bash: Nach jedem "einfachen Kmdo" mit Exit-Status != 0 a.	
SIGHUP	1	Konfigurationsdatei erneut einlesen (Daemon)	[hangup]
SIGINT	2	Abbrechen durch <Strg-C>	[interrupt]
SIGKILL	9	Bedingungsloser Prozessabbruch	[kill]
SIGTERM	15	Prozess beenden (Standard-Signal von "kill")	[terminate]
SIGUSR1	10	Benutzerspezifisch Nr 1	[user]
SIGUSR2	12	Benutzerspezifisch Nr 2	[user]
SIGCONT	18	Ausgabe weiterlaufen lassen durch <Strg-Q>	[continue]
SIGSTOP	19	Ausgabe anhalten durch <Strg-S>	[stop]
SIGTSTP	20	In Hintergrund stellen durch <Strg-Z>	[terminate stop]

```
  | SIGQUIT   |  3 | Prozessende erreicht                                 [quit] |
  | SIGILL    |  4 | Nicht erlaubte Anweisung gefunden                [illegal] |
  | SIGBUS    |  7 | Bus-Zugriffsverletzung (Alignment)                   [bus] |
  | SIGFPE    |  8 | Mathematik-Fehler              [floating point exception] |
  | SIGSEGV   | 11 | Fehlerhafter Speicherzugriff          [segment violation] |
  | SIGPIPE   | 13 | Prozess in Pipeline nicht mehr da           [broken pipe] |
  | SIGALRM   | 14 | Timeout hat stattgefunden                          [alarm] |
  | SIGCHLD   | 17 | Ein Kindprozess hat sich beendet                   [child] |
  +-----------+----+-----------------------------------------------------+

HINWEIS: Einige Signale werden automatisch zwischen Eltern- und Kindprozessen
ausgetauscht (z.B. SIGCHLD, SIGPIPE) bzw. bei bestimmten Ereignissen
automatisch vom Betriebssystem an Prozesse verschickt (z.B. SIGILL, SIGBUS,
SIGFPE, SIGSEGV).

Endet ein Kindprozess, dann wird automatisch an den Elternprozess das Signal
SIGCHLD (17) geschickt. Endet ein per Pipe mit einem anderen Prozess verbundener
Prozess, dann wird an seinen Partnerprozess das Signal SIGPIPE (13) geschickt.

Versenden von Signalen in der Shell:

  kill PID          # Signal 15 (TERM) an Prozess verschicken (kontrolliert abbr.)
  kill -9 PID       # Signal 9 (KILL) an Prozess verschicken (unbedingt abbrechen)
  kill -HUP PID     # Signal 1 (HUP) an Prozess verschicken (Konfig neu lesen)
  KMDO              # Kommando starten
  <Strg-C>          # Signal 2 (INT) an laufendes Kommando schicken
  KMDO              # Kommando starten
  <Strg-Z>          # Signal 20 (TSTP) an laufendes Kommando schicken
                    # --> Angehalten und in den Hintergrund gestellt
  KMDO              # Kommando starten
  <Strg-S>          # Signal 19 (STOP) an laufendes Kommando schicken --> anhalten
  <Strg-Q>          # Signal 18 (CONT) an stehendes Kommando schicken --> weiterlaufen
  killall NAME      # Signale 15 (TERM) an alle Prozesse mit Kmdo NAME verschicken
  kill -l           # Alle Signale mit Name + Nummer auflisten
  man 3 signal      # Liste + Beschreibung aller Signale

Abfangen von Signalen in der Shell:

  trap "KMDO" 1       # Signal 1 (HUP) abfangen und KMDO ausführen
  trap "KMDO" 2 15    # Signal 2+15 (INT+TERM) abfangen und KMDO ausführen
  trap                # Liste aller abgefangenen Signale ausgeben


2.8) Konfigurationsdateien
--------------------------
Die meisten Programme lesen beim Start (und evtl. auch beim Empfang des Signals
SIGHUP=1) einen fest definierten Satz von Konfigurationsdateien mit festen
Namen ein. Diese Konfigurationsdateien heißen häufig so wie das Programm selbst
und liegen zentral im "/etc"-Verz. bzw. in den Heimatverz. der Benutzer in Form
von "versteckten" Dateien und/oder Verz. (der Datei/Verz.name beginnt mit ".").

Zuerst wird für alle Benutzer die zentrale/globale Konfigurationsdatei, dann
wird pro Benutzer dessen benutzerspezifische/lokale Konfigurationsdatei aus
seinem Heimatverz. eingelesen. Letztere überschreibg die zentralen Einstellungen
Hier als Beispiel die Konfigurationsdateien der Bourne-Shell "sh":

  /etc/profile    # Zentral
  ~/.profile      # Benutzerspezifisch

Oder für den Editor "emacs":

  /etc/emacs/*    # Zentral
  ~/.emacs.d/*    # Benutzerspezifisch

Oder für den SSH-Server "sshd" und den SSH-Client "ssh":

  /etc/ssh/sshd_config    # Zentral (Server, ACHTUNG: "d"=Daemon im Namen)
  /etc/ssh/ssh_config     # Zentral (Client, ACHTUNG: KEIN "d" im Namen)
  ~/.ssh/ssh_config       # Benutzerspezifisch (Client)

Fehlt eine Konfigurationsdatei, so wird dies in der Regel stillschweigend
übergangen und das im Programm eingebaute Standardverhalten aktiviert.

HINWEIS: Werden Einstellungen in einer Konfigurationsdateien geändert, so ist
dies einem bereits laufenden Prozess per Signal SIGHUP mitzuteilen oder er ist
anzuhalten und neu zu starten, damit er sie erneut einliest.


3) Weitere Schnittstellen
-------------------------
Die weiteren Schnittstellen von UNIX-Programmen müssen explizit GEÖFFNET
werden, d.h. es muss im Programm explizit eine Verbindung dazu hergestellt
```

werden. Im Gegensatz dazu sind alle vorher beschriebenen Schnittstellen
AUTOMATISCH vorhanden und können sofort benutzt werden, ohne sich um den Aufbau
einer Verbindung zu kümmern.

  * Dateien (Gerätedatei)
  * Interprocess Communication (IPC)
    + Named Pipe (FIFO)
    + Locking
    + Semaphor
    + Message Queue
    + Shared Memory
    + Memory Mapped Files
    + UNIX-Socket
  * Netzwerk
    + UDP/TCP-Socket
  * Datenbank (UNIX-Socket, UDP/TCP-Socket)
  * Transaktionsmonitor
    + MOM = Message Oriented Middleware
    + ESB = Enterprise Service Bus
  * usw...

3.1) Dateien

Ein Prozess kann nahezu beliebig viele Dateien öffnen, auf die er je nach
Dateityp, Zugriffsmodus und Zugriffsrechten (auf Basis der effektiven UID/GID
des Prozesses und der UID/GID der Datei) sequentiellen/wahlfreien lesenden
und/oder schreibenden Zugriff hat. Als besondere Form von "Dateien" sind auch
"Gerätedateien" möglich, die den kontrollierten Zugriff auf Hardware erlauben.

Das UNIX-Betriebssystem führt kein "mandatory" Locking durch, sondern kennt nur
ein "advisory" Locking. D.h. mehrfacher gleichzeitiger Zugriff auf die selbe
Datei wird vom Betriebssystem nicht automatisch überwacht oder unterbunden.
Soll eine Datei dagegen geschützt werden, dann ist das Locking in die
Anwendung(en) einzubauen ("advisory" Locking). Das Betriebssystem stellt dafür
Mechanismen zur Verfügung (z.B. "flock").

3.2) Interprocess Communication (IPC)

Zur Kommunikation zwischen Prozessen stellt das UNIX-Betriebssystem mehrere
Mechanismen zur Verfügung, die sich im Laufe der Zeit entwickelt haben.

  * Named Pipe (FIFO): Gemeinsam genutzte Datei im Dateisystem
    (synchronisiert Zugriffe automatisch)
  * Locking: Flag, das gemeinsam genutzte Resourc gegen
    gleichzeitigen Zugriff schützt
  * Semaphor: Zähler, der gemeinsam genutzte Resource gegen
    gleichzeitigen Zugriff schützt
  * Message Queue: Nachrichten versenden/puffern/empfangen
  * Shared Memory: Gemeinsam genutzter Speicher für mehrere Prozesse
  * Memory Mapped File: Dateiinhalt in Prozessspeicher einblenden
    (lesen/schreiben)
  * UNIX-Socket: Analog UDP/TCP-Socket anzusteuern,
    aber rein lokale Verbindungen ohne Netzwerk

3.3) Netzwerk

* UDP/TCP-Socket

3.4) Datenbank

* Noch zu tun!