HOWTO zu UNIX/Linux-Prozess-Schnittstellen

$Id: unix-process-interfaces-HOWTO.txt,v 1.36 2025/02/23 20:14:55 tsbirn Exp $

Dieses Dokument beschreibt die Schnittstellen von UNIX/Linux-Prozessen.

INHALTSVERZEICHNIS

---

1) Einleitung

Ein UNIX/Linux-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) Standard-Schnittstellen (automatisch vorhanden)

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) Programm-Name	IN	Ein Wort
2) Aufruf-Argumente	IN	Liste von Worten
3) Umgebungs-Variable	IN	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) Exit-Status                        |   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) Konfigurations-Dateien             |   IN   | Byte-Strom (Text/Daten)   |
  +---------------------------------------+--------+---------------------------+
```

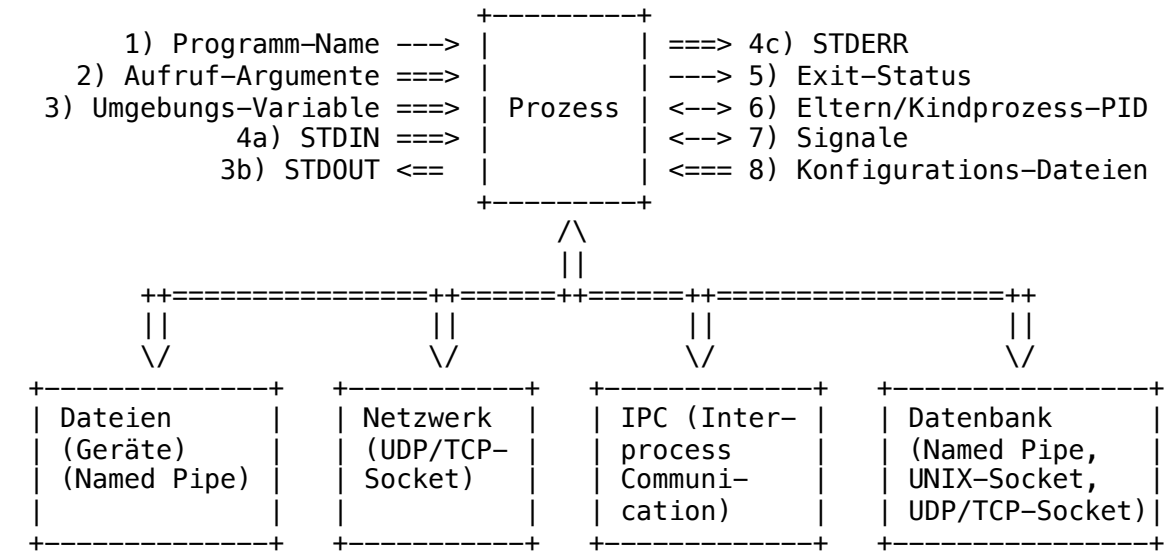
1.2) Manuelle Schnittstellen

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 Eltern-Prozesse und ihre Kind-Prozesse.

```
  +-----------------------+-------------------------------------------------+
  | Schnittstelle         | Beschränkung/Richtung                           |
  +-----------------------+-------------------------------------------------+
  | Programm-Name         | Eltern --> Kind                                 |
```

```
  | Aufruf-Argumente       | Eltern ==> Kind                                  |
  | Umgebungs-Variable     | Eltern ==> Kind                                  |
  | STDIN                  | Eltern ==> Kind                #=> Geschwister   |
  | STDOUT                 | Eltern <== Kind + Geschwister <=#                |
  | STDERR                 | Eltern <== Kind                                  |
  | Exit-Status            | Eltern <-- Kind                                  |
  | Eltern/Kindprozess-PID | Eltern <-> Kind                                  |
  | Signale                | Eltern <-- Kind + Geschwister <--> Geschwister   |
  +------------------------+--------------------------------------------------+
```

In grafischer Form:

```
  +---------+ -->   Programm-Name  --> +---------+  per Pipe |    +---------+
  |         | ==> Aufruf-Argumente ==> |         |   verbunden    |         |
  |         | ==>  Umgebungs-Var.  ==> |         |                |         |
  | Eltern- | ==>      STDIN       ==> |  Kind-  | STDOUT   STDIN |  Kind-  |
  | Prozess | <==      STDOUT      <== | Prozess | ==============> | Prozess |
  |         | <==      STDERR      <== |    1    | <------------> |    2    |
  |         | <--    Exit-Status   <-- |         | Signal SIGPIPE |         |
  |         | <->  Elt/Kind-PID    <-> |         |                |         |
  +---------+ <-- Signal SIGCHLD   <-- +---------+                +---------+
```

2) Die Standard-Schnittstellen

2.1) Programm-Name

Jedem Prozess ist der Programm-Name bekannt, über den er gestartet wurde. Dieser Programm-Name ist Teil der Aufruf-Argumente 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 Programm-Name nicht mehr dem eigentlichen Programm-Namen entspricht, sondern dem Linknamen.

```
  skript.sh       # Skript-Aufruf
  echo $0         # Programm-Name 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                 # Programm-Name im Skript ausgeben --> .../hl.sh
  sl.sh                   # Skript-Aufruf über Softlink
  echo $0                 # Programm-Name im Skript ausgeben --> .../sl.sh
```

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

```
  case $(basename $0) in   # Reinen Programm-Namen 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) Aufruf-Argumente

Umfassen alle beim Aufruf eines Programms hinter dem Programm-Namen 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 2 Mio Zeichen). Der Programm-Name und die einzelnen Aufruf-Argumente werden durch "WHITESPACE" (Leerzeichen, Tabulator, Newline) voneinander getrennt (--> Quotierung). Beispiel sind die Optionen des Befehls "ls" und der Quell- und Zieldateiname des Befehls "mv".

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:

```
  skript.sh a b c d e f    # Skript-Aufruf mit 6 Argumenten
  echo "1. Argument: $1"   # Im Skript --> a
  echo "2. Argument: $2"   # Im Skript --> b
  echo "3. Argument: $3"   # Im Skript --> c
  echo "4. Argument: $4"   # Im Skript --> d
  echo "5. Argument: $5"   # Im Skript --> e
  echo "6. Argument: $6"   # Im Skript --> f
  echo "Anz. Argum.: $#"   # Im Skript --> 6
  echo "Alle Argum.: $@"   # Im Skript --> a b c d e f
  echo "Alle Argum.: $*"   # Im Skript --> a b c d e f
  shift                    # Ersten Aufrufparameter entfernen, Rest nachrücken
  echo "$# - $*"           # --> 5 - b c d e f
  shift 4                  # Weitere 4 Aufrufparam entfernen, Rest nachrücken
  echo "$# - $*"           # --> 1 - 6
```

2.3) Umgebungs-Variable

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 50-100 Variablen.

Diese Liste von Umgebungs-Variablen 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.

Beispiele sind Variable HOME, die den Befehl "cd" beeinflusst, Variable PATH zur Steuerung der Suche nach eingetippten Kommandos und Variablen LANGUAGE, LANG, LC_ALL, LC_MESSAGE, LC_... mit denen Spracheinstellung vieler Programme festlegt werden.

Umgebungs-Variable in der Shell anlegen:

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

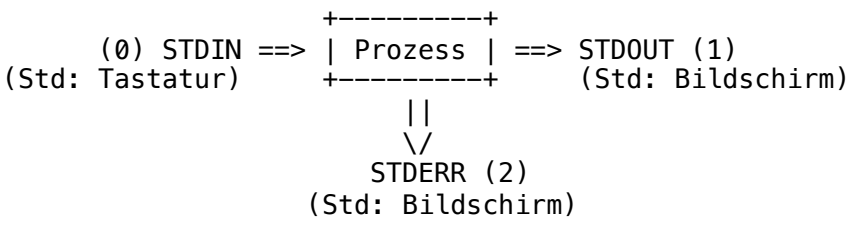
Zugriff auf Umgebungs-Variable in der Shell:

```
  echo $VAR     # Wert einer Umgebungs-Variable ausgeben
  echo ${VAR}   # Wert einer Umgebungs-Variable ausgeben
  env           # Alle aktuell gesetzten Umgebungs-Variablen auflisten
  env | sort    # ... alphabetisch sortiert auflisten
```

ACHTUNG: Umgebungs-Variable nicht mit Shell-Variablen verwechseln! Letztere sind nur für die Shell selbst relevant und steuern deren Verhalten. Sie werden ausschliesslich von der Shell verwendet und 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 effiziente Übergabe sehr großer Datenmengen. Die 3 Kanäle haben feste Namen (STDIN, STDOUT, STDERR) und feste Nummern (0, 1, 2):

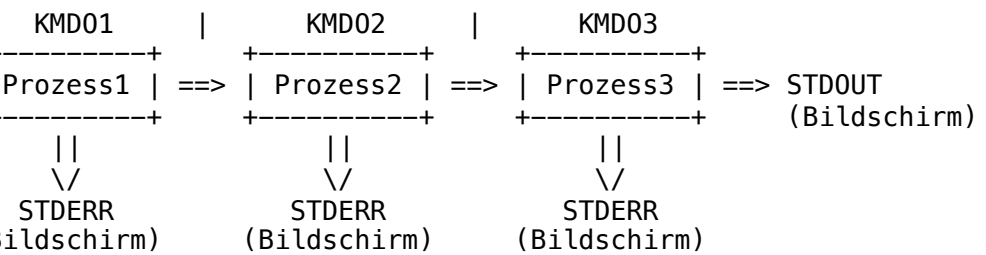


Ein Prozess behandelt die 3 Kanäle wie bereits geöffnete sequentielle Dateien, aus denen er ausschließlich lesen oder in die er ausschließlich schreiben kann. D.h. darin neu positionieren ist nicht möglich, bereits gelesene Daten können nicht erneut gelesen und bereits geschriebene Daten können nachträglich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



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)
  cat | ...            # 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) Exit-Status

Wird von jedem Prozess am Ende an seinen Elternprozess zurückgegeben:

  * Wert "0" bedeutet, dass der Prozess ERFOLGREICH ausgeführt wurde.
  * Wert ungleich "0" bedeutet, dass ein FEHLER auftrat.

Im Prinzip genügt diese Unterscheidung zum Erkennen, ob ein Prozess erfolgreich durchlief oder im Rahmen seiner Tätigkeit ein Fehler auftrat. Falls von einem Programm verschiedene Exit-Stati für verschiedene Fehlerarten zurückgegeben werden, dann werden diese in der zugehörigen man-Page beschrieben.

Der Exit-Status 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 (rein verwaltungstechnisch), bis er vom Betriebssystem mit dem Prozess "init/systemd" verbunden wird und dort seinen Exit-Status abliefern kann.

Exit-Status in der Shell erzeugen (d.h. Abbruch der Shell/des Shell-Skripts):

```
  exit     # Exit-Status des zuletzt ausgeführten Kommandos zurückgeben
  exit 0   # Exit-Status 0 zurückgeben
  exit 1   # Exit-Status 1 zurückgeben
```

Zugriff auf Exit-Status eines Kommandos in der Shell:

```
  KMDO       # Kommando aufrufen --> liefert Exit-Status zurück
  echo $?    # Exit-Status des vorherigen Kommandos abfragen (geht nur 1x!)
             #
  if KMDO    # Exit-Status von Kommando abfragen und für Verzweigung nutzen
  then       # (geht nur 1x!)
      echo "Exit-Status == 0 --> alles OK!"
  else
      echo "Exit-Status != 0 --> irgendein Fehler!"
  fi
             #
  KMDO       # Kommando aufrufen --> liefert Exit-Status zurück
  ES="$?"    # Exit-Status von Kommando in Variable ES merken
  echo "$ES" # --> mehrfach nutzbar
  echo "$ES" # --> mehrfach nutzbar
  echo "$ES" # --> mehrfach nutzbar
```

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 Versenden 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 Konfigurations-Datei neu ein), ihn abbrechen (z.B. SIGKILL), es kann aber auch ignoriert werden. Es gibt 32 (bzw. 64 Signale inkl. der Realtime = RT-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	Konfigurations-Datei erneut einlesen (Daemon) [hangup]
SIGINT	2	Abbrechen per <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 per <Strg-Q> [continue]
SIGSTOP	19	Ausgabe anhalten per <Strg-S> [stop]
SIGTSTP	20	In Hintergrund stellen per <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]

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.

Alle Signale außer SIGKILL (9) und SIGTSTP (20) können von einem Prozess abgefangen und so z.B. ignoriert werden.

HINWEIS: Die mit "Bash" gekennzeichneten Signale sind nur innerhalb eines Bash-Skriptes relevant, es sind KEINE ECHTEN SIGNALE des UNIX-Systems.

Versenden von Signalen in der Shell:

```
kill PID        # Signal 15 (TERM) an Prozess versenden (kontrolliert abbr.)
kill -9 PID     # Signal 9 (KILL) an Prozess versenden (unbedingt abbrechen)
kill -HUP PID   # Signal 1 (HUP) an Prozess versenden (Konfigdatei einlesen)
KMDO            # Kommando starten
<Strg-C>        # Signal 2 (INT) an laufendes Kommando senden
```

```
  KMDO            # Kommando starten
  <Strg-Z>        # Signal 20 (TSTP) an laufendes Kommando senden
                  # --> Angehalten und in den Hintergrund gestellt
  bg              # --> Im Hintergrund weiterlaufen lassen (background)
  fg              # --> Aus Hintergrund in Vordergrund holen (foreground)
  KMDO            # Kommando starten
  <Strg-S>        # Signal 19 (STOP) an laufendes Kmdo senden --> anhalten
  <Strg-Q>        # Signal 18 (CONT) an angehaltenes Kmdo senden --> weiterlaufen
  killall NAME    # Signale 15 (TERM) an alle Prozesse mit Kmdo NAME versenden
  kill -l         # Alle Signale mit Name + Nummer auflisten (list)
  man 3 signal    # Liste + Beschreibung aller Signale
```

Abfangen von Signalen in der Shell:

```
  trap "KMDO" EXIT   # Skriptende abfangen und KMDO ausführen (d.h. bei "exit")
  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) Konfigurations-Dateien

Die meisten Programme lesen beim Start (und evtl. auch beim Empfang des Signals SIGHUP=1) einen fest definierten Satz von Konfigurations-Dateien mit festen Namen ein. Diese Konfigurations-Dateien 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. (Datei/Verz.name beginnt mit ".").

Beim Programm-Start wird zuerst die zentrale/globale Konfigurations-Datei aus dem /etc-Verz. eingelesen, anschlie0end wird die benutzerspezifische/lokale Konfigurations-Datei aus seinem Heimatverz. eingelesen. Letztere kann die zentralen Einstellungen verändern/ergänzen/überschreiben. Hier als Beispiel die Konfigurations-Dateien der Bourne-Shell "sh":

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

Oder für die Editoren "emacs", "vim" und "nano/pico":

```
  /etc/emacs/*   # Emacs zentral
  ~/.emacs.d/*   # Emacs benutzerspezifisch
  /etc/vimrc     # Vim zentral
  ~/.vimrc       # Vim benutzerspezifisch
  /etc/nanorc    # Nano/Pico zentral
  ~/.nanorc      # Nano/Pico 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 Konfigurations-Datei, so wird dies in der Regel stillschweigend übergangen und das im Programm eingebaute Standardverhalten aktiviert.

HINWEIS: Werden Einstellungen in einer Konfigurations-Datei 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!