Einfache Prozesskommunikation: unidirektionale, anonyme Kommunikation mit Pipes

Dieses Hilfsblatt beschreibt essentielle Grundlagen der Prozesskommunikation via POSIX-API.

Es wird ausschließlich auf die Kommunikation über Pipes eingegangen. Schauen Sie sich die benutzten Systemaufrufe zusätzlich in der Dokumentation an!

Für die Bearbeitung der Praktikumsaufgabe müssen die beiden im Folgenden vorgestellten Mechanismen kombiniert werden.

Vorbemerkung

In UNIX und davon abgeleiteten Betriebssystemen werden für Zugriffe auf Dateien sog. *Filedeskriptoren* verwendet. Diese Deskriptoren sind ganzzahlige Werte und werden pro Prozess in einer Tabelle verwaltet. Beim Öffnen von Dateien, wird der Deskriptor jeweils hochgezählt.

Jedem Prozess werden bei dessen Erzeugung vom Kernel drei Deskriptoren zugeteilt:

- **`0`**: Standardeingabe (*stdin*),
- **`1`**: Standardausgabe (*stdout*),
- **`2`**: Standardfehlerausgabe (*stderr*).

Diese Filedeskriptoren sind in der Regel mit einem Terminal (aus historischen Gründen manchmal auch TTY (teletypewriter) genannt) verbunden. Der Ein- / Ausgabestrom kann umgeleitet werden, z. B. in eine Datei (**`ls -al 1> dir.txt`**). Durch diesen Mechanismus können auch Prozesse miteinander verbunden werden.

Erzeugung von Pipes

Eine Pipe stellt eine **unidirektionale** Kommunikationsschnittstelle zwischen zwei Prozessen dar.

Die Kommunikation wird über zwei (spezielle) Dateien, die durch Dateideskriptoren referenziert werden, vorgenommen. Dazu werden die Daten durch den ersten Prozess in das eine Ende der Pipe geschrieben, dort zwischengespeichert und am anderen Ende durch den zweiten Prozess nach dem FIFO Prinzip ausgelesen.

Syntax

```
#include <unistd.h>
int pipe(int fdes[]);
```

Parameter

- **`fdes`**: Das Feld ist ein Platzhalter für zwei Dateideskriptoren. Die referenzierten Dateien werden geöffnet[1]. In die durch den Deskriptor **`fdes[1]`** adressierten Datei können Daten **geschrieben** und von der durch **`fdes[0]`** adressierten Datei können Daten **gelesen** werden. In einigen Systemen ist die Pipe bidirektional (beide Dateien sind sowohl zum Lesen als auch

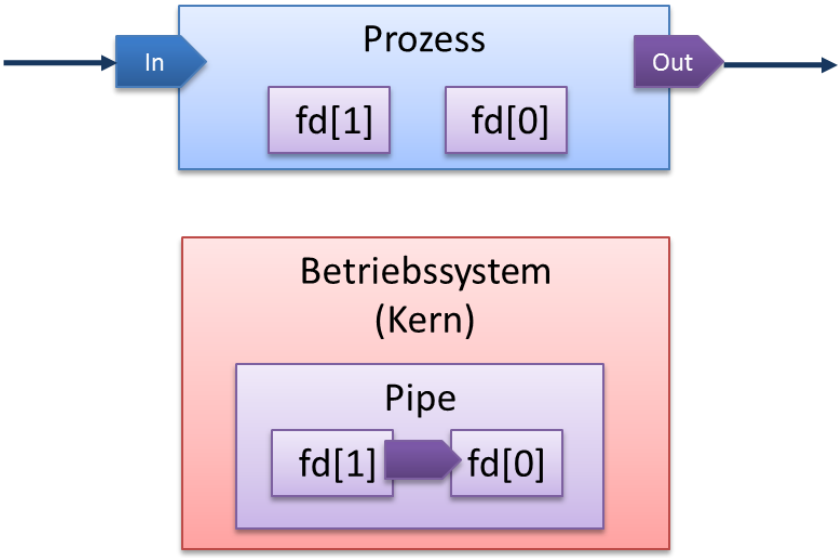
[1] Genau genommen wird nur eine Datei geöffnet, aus der sowohl gelesen werden und in die geschrieben werden kann.

zum Schreiben geöffnet); POSIX spezifiziert aber nur **`fdes[1]`** als Schreibdeskriptor und **`fdes[0]`** als Lesedeskriptor.

Rückgabewert

- 0 bei Erfolg, sonst -1.

Werden 2 Prozesse über eine Pipe gekoppelt, so müssen lesender und schreibender Prozess unterschieden werden. Der jeweils nicht verwendete Deskriptor (beim lesenden Prozess ist dies also **`fdes[1]`**, beim schreibenden **`fdes[0]`**) sollte jeweils mit **`close()`** geschlossen werden, da über das Schließen der Dateien die Terminierung des Prozesses durch den jeweils anderen Prozesses erkannt wird. [2]

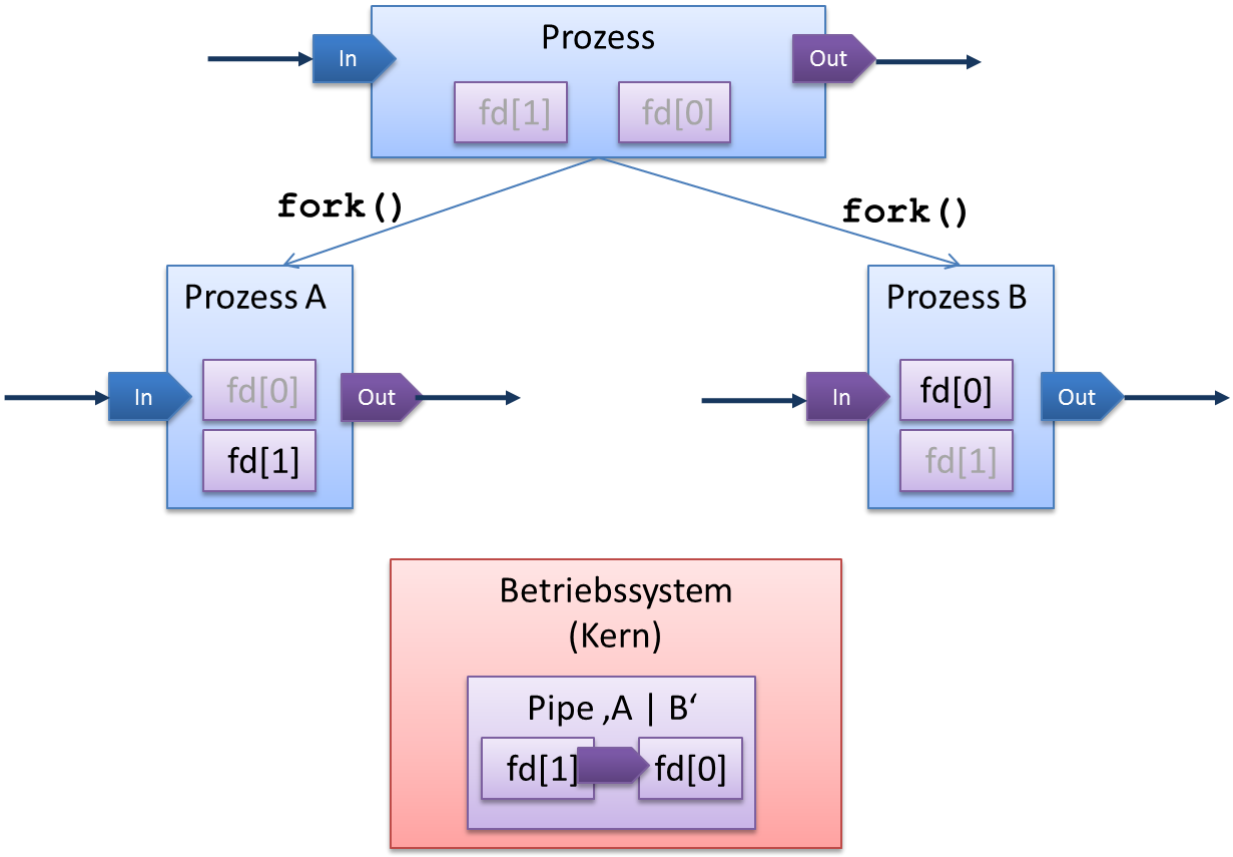


In vielen UNIX-Systemen können die geöffneten Dateien und damit auch die vom Prozess verwendeten Pipes unter Angabe der PID unter **`/proc/<PID>/fd`** eingesehen werden.

Da bei einem Aufruf von **`fork()`** ein Kindprozess die geöffneten Dateien vom Elternprozess erbt, können somit Informationen zwischen Eltern und Kindprozess ausgetauscht werden bzw. zwischen mehreren Kindern, die alle die Dateideskriptoren der Pipe erben können.

Nach Benutzung der Pipe müssen die verbleibenden mit **`pipe()`** erzeugten Deskriptoren mit **`close()`**wieder geschlossen werden.

[2] Aus diesem Grund spricht man auch von Halb-Duplex-Verhalten. Schließt man die Deskriptoren nicht und behält die initial gegebene bidirektionale Kommunikation bei, so arbeitet man im Duplex-Verhalten vorliegen.



Duplizieren von Dateideskriptoren

Findet ein **exec<l|v>[e|p]** Systemaufruf statt, so werden normalerweise alle offenen Dateien des Prozesses vorher geschlossen (*close-on-exec*), denn **exec** lädt einen Maschinen-Code in einen laufenden Prozess. Ansonsten könnten Konflikte speziell beim schreibenden Zugriff auf Dateien entstehen.

Um Dateideskriptoren zu erzeugen, die auch einen Aufruf eines **exec** Befehls überstehen, dienen die Befehle **dup()** bzw. **dup2()**.

Syntax: dup()

```
#include <unistd.h>
int dup(int fdes);
```

Parameter

- **fdes**: Der Dateideskriptor, der dupliziert werden soll.

Rückgabewert

- Bei Erfolg wird ein Deskriptor zurückgeliefert, der auf die Datei (oder Pipe) referenziert, auf die auch **fdes** zeigt (beide benutzen dieselbe interne Struktur).
- Bei einem Fehler wird -1 zurückgegeben.

Dieser Deskriptor überlebt den **exec** Systemaufruf, d. h. ein mit **exec** gestartetes Programm kann diesen Deskriptor zur Ein- oder Ausgabe benutzen. Dies wird intern dadurch erreicht, dass der duplizierte Deskriptor mit entsprechenden Attributen (Flags) versehen wird.

Der zurückgelieferte Deskriptor ist der kleinste freie Deskriptor im System.

Anwendungsbeispiel

Soll z.B. die Ausgabe eines Befehls in eine Datei umgeleitet werden, kann wie folgt verfahren werden:

```
/* Im Beispiel erfolgt keine Fehlerbehandlung!  */
/* Machen Sie es besser!                        */
int fd;
fd = open("TestDatei.dat", O_RDWR | O_CREAT | O_TRUNC);

/* stdout Strom wird geschlossen.       */
/* Damit ist die 1 als Deskriptor frei. */
close(STDOUT_FILENO);

/* mit dup() wird der Deskriptor 1 der Testdatei zugewiesen */
dup(fd);

/* ls wird aufgerufen, Ausgabe geht in Testdatei! */
execlp("ls", "ls", "-l", ".", (char *)NULL);
```

Syntax: dup2():

```
#include <unistd.h>
int dup2(int fdes1, int fdes2);
```

Parameter

- **fdes1**: Der Dateideskriptor, der dupliziert werden soll.
- **fdes2**: Zeigt nach erfolgreicher Ausführung auf dieselbe Datei wie **fdes1**. Ist **fdes2** eine geöffnete Datei, so wird diese zuvor geschlossen.

Rückgabewert

- Siehe **dup()**

Die Funktion **dup2(int fdes1, int fdes2)** verbindet den Filedeskriptor **fdes1** mit dem Filedeskriptor **fdes2**. **dup2()** verkürzt die Anzahl der Anweisungen im Vergleich zur Nutzung von **dup()** um eine Zeile. Ein weiterer Vorteil ist, dass das Schließen und Duplizieren des alten Filedeskriptors eine **atomare** Operation darstellt. Damit wird verhindert werden, dass nach dem Schließen und einem Prozesswechsel, bei dem unter Umständen die Dateitabelle des Prozesses verändert wird, anschließend ein anderer als der gewünschte Deskriptor dupliziert wird.

Anwendungsbeispiel

Soll z. B. die Ausgabe eines Befehls in eine Datei umgeleitet werden, kann wie folgt verfahren werden:

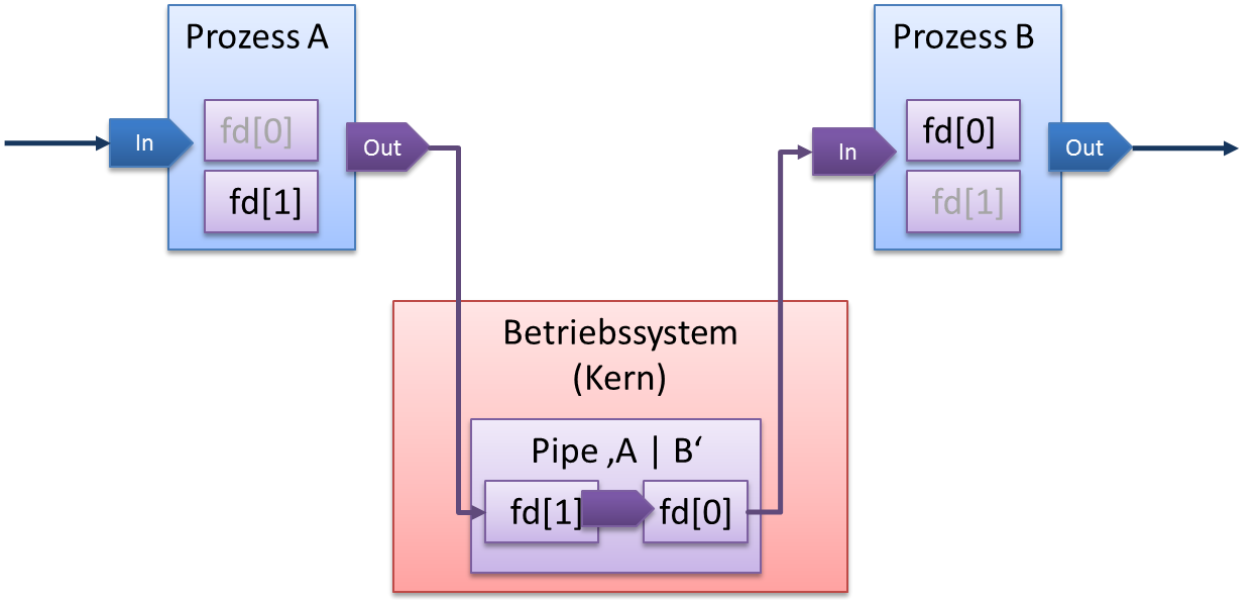
```
/* Im Beispiel erfolgt keine Fehlerbehandlung!  */
/* Machen Sie es besser!                        */
int fd;
fd = open("TestDatei.dat", O_RDWR | O_CREAT | O_TRUNC);

/* mit dup2() wird stdout der Testdatei zugewiesen */
dup2(fd, STDOUT_FILENO);

/* ls wird aufgerufen, Ausgabe geht in Testdatei! */
execlp("ls", "ls", "-l", ".", (char *)NULL);
```

Um eine Interprozess-Kommunikation zu ermöglichen, muss die Ausgabe des ersten Prozesses in die Eingabe des zweiten Prozesses umgelenkt werden. Will man eine einstufige Pipe realisieren, so kann man wie folgt vorgehen:

- Im ersten Prozess den nicht benötigten Eingabe-Deskriptor schließen und die Ausgabe der Pipe **`fd[1]`** duplizieren und die Standard-Ausgabe (Deskriptor 1) in **`fd[1]`** per **`dup2()`** umleiten.
- Im zweiten Prozess den nicht benötigten Ausgabe-Deskriptor schließen und die Eingabe der Pipe **`fd[0]`** duplizieren und die Standard-Eingabe (Deskriptor 0) in **`fd[0]`** per **`dup2()`** umleiten.



Referenzen

- W. Richard Stevens, Stephen A. Rago: Advanced Programming in the UNIX Environment. Second Edition, Addison-Wesley Professional, 2013. ISBN 0321525949
- Bruce Molay: Understanding UNIX/Linux programming, Prentice Hall, 2003. ISBN 0321637739