J. Gutenberg-Universität Mainz
Prof. Dr. Reinhold Kröger
Fabian Meyer M.Sc.

Betriebssysteme
WS 2012/13

Übungsblatt 8
Praktische Übungen

In dieser Übung werden UNIX-Systemaufrufe zur Signalisierung zwischen Prozessen sowie zur UNIX-Interprozesskommunikation mittels Pipes vertieft.

Signalisierung stellt eine einfache Form der Prozesskommunikation dar. Die Signalisierung unter UNIX ist dabei auf verwandte Prozesse beschränkt. In der Vorlesung wurden die zur Signalisierung zur Verfügung stehenden Systemdienste besprochen (vgl. Kap. 6.1). Konstanten, Strukturen usw. sind in der Include-Datei `<signal.h>` enthalten.

Pipes erlauben die unidirektionale, nachrichtenorientierte Prozesskommunikation. Einer Pipe sind dazu zwei Enden zugeordnet, die sich als übliche Dateideskriptoren darstellen. An einem Ende können Nachrichten in die Pipe geschrieben werden, am anderen Ende werden sie entnommen. Semantisch wird dabei ein Bytestrom zwischen den Enden der Pipe transportiert. Einer Pipe ist ein beschränkter Puffer zugeordnet. Die notwendige Synchronisation zwischen Erzeugern und Verbrauchern (Blockieren eines Verbrauchers bei leerem Puffer, Blockieren eines Schreibers bei vollem Puffer) erledigt das Betriebssystem. Im Falle einer sogenannten "Named Pipe" (auch FIFO genannt, Erzeugung mittels `mkfifo()`) werden ein üblicher Dateiname zur Benennung der Pipe und die üblichen Dateioperationen (open, close, read, write) verwendet. Da der Dateinamensraum für alle Prozesse gemeinsam nutzbar ist, ist über eine Named Pipe eine Kommunikation zwischen beliebigen Prozessen auf einem UNIX-Rechner möglich. Im Falle der klassischen "anonymen" Pipe existiert dagegen keine Repräsentierung der Pipe im Dateisystem, und die Pipe erlaubt ausschließlich die Kommunikation zwischen verwandten Prozessen. Dazu wird die Pipe von einem (Vater)-Prozess erzeugt (anonyme Pipe mittels `pipe()`). Ihm stehen (zunächst relativ sinnlos) beide Enden der Pipe als File-Deskriptoren geöffnet zur Verfügung. Nach der Erzeugung eines Sohn-Prozesses ist es diesem ebenfalls möglich, die Enden der Pipe über dieselben Dateideskriptoren zu nutzen. (Die Menge der offenen Datei-Deskriptoren wird an den Sohn vererbt). Damit ist eine Kommunikation zwischen Vater und Sohn, bzw. im Falle mehrerer erzeugter Söhne auch zwischen diesen möglich. Die nicht genutzten Enden der Pipe müssen mittels `close()` geschlossen werden. Für den Umgang mit Pipes stehen die in der Vorlesung besprochenen Systemdienste (vgl. Folie 6-7ff) zur Verfügung. Konstanten, usw. sind in der Include-Datei `<fcntl.h>` enthalten.

Aufgabe 8.1:

Erstellen Sie ein C-Programm `sig_empf1.c`, das eingehende Signale durch jeweils einen eigenen Handler verarbeiten soll und folgende Eigenschaften besitzt:

(a) Das Programm gibt zunächst seine Prozess-Id aus, (damit später durch die Shell einfach auf den Prozess Bezug genommen werden kann).

(b) Bei Empfang des Signals `SIGUSR1` erfolgt die Ausgabe "Signal SIGUSR1 empfangen".

(c) Bei Empfang des Signals `SIGUSR2` erfolgt die Ausgabe "Signal SIGUSR2 empfangen".

(d) Das Programm stellt sich einen Wecker, der alle ALARM_PERIOD=3 sec abläuft. Bei Ablauf erfolgt die Ausgabe "Timer abgelaufen".

(e) Bei Empfang des Signals `SIGTERM` erfolgt die Ausgabe "Programmende" und die Beendigung des Programms.

(f) Starten Sie das Programm, signalisieren Sie die Signale `SIGUSR1`, `SIGUSR2`, `SIGTERM` und `SIGKILL` von der shell aus an Ihr laufendes Programm. Benutzen Sie dazu das Dienstprogramm `kill`. Beobachten Sie die Ausgabe des Programms.

Aufgabe 8.2:

Erstellen Sie ein Programm `sig_sender.c`, das aus einem kleinen Menu heraus ein und zwei wählbare Signale an einen anderen Prozess (der z.B. das Programm aus Aufgabe 8.1 ausführt) erzeugen kann und folgende Eigenschaften besitzt:

(a) Zur Identifikation des empfangenden Prozesses sollen alle Möglichkeiten des Parameters `pid` im Aufruf `kill` benutzt werden können.

(b) Im Falle von zwei Signalen soll die Wartezeit zwischen dem Senden des ersten Signals und dem Senden des zweiten Signals wählbar sein.

(c) Benutzen Sie das Programm `sig_sender`, um die Signale `SIGUSR1`, `SIGUSR2`, `SIGTERM` und `SIGKILL` an das laufende Programm entsprechend Aufgabe 8.1 zu senden.

(d) Was passiert, wenn als `pid` 0 gewählt wird, und warum?

Aufgabe 8.3 (optional):

Erstellen Sie ein Programm `sig_empf2.c`, das in Erweiterung von Aufg. 8.1 folgende Eigenschaften besitzt:

(a) Es wird ein globaler Integer-Zähler count eingeführt.

(b) ALARM_PERIOD wird auf 1 sec gesetzt. Bei Ablauf erfolgt die Ausgabe "Timer abgelaufen" sowie der aktuelle Wert von count.

(c) Bei Empfang des Signals `SIGUSR1` soll die Ausgabe "Signal SIGUSR1 empfangen, Start Zählen" erfolgen, anschließend in einer Schleife der Zähler count bis MAX_COUNT (z.B. 10.000.000.000) gezählt werden und abschließend die Ausgabe "bis MAX_COUNT gezählt" erfolgen. (Genauer sollen Sie MAX_COUNT so groß wählen, dass bei Eintreffen des Signals `SIGUSR2` entsprechend (d) die Schleife noch nicht zu Ende ist.)

(d) Bei Empfang des Signals `SIGUSR2` erfolgt weiter die Ausgabe "Signal SIGUSR2 empfangen".

(e) Bei Empfang des Signals `SIGTERM` erfolgt weiter die Ausgabe "Programmende" und die Beendigung des Programms.

(f) Benutzen Sie das Programm `sig_sender` entsprechend Aufgabe 8.2, um Signale an das laufende Programm `sig_empf2` zu senden. Erzeugen Sie insbesondere ein Signal `SIGUSR2`, wenn Sie kurz zuvor ein Signal `SIGUSR1` erzeugt haben.

(g) Beobachten Sie das Programm `sig_empf2`, und protokollieren Sie die eintretenden Ereignisse bzgl. der Signalbearbeitung von `SIGUSR1`, `SIGUSR2` und `SIGALRM`. Was können Sie zur Atomarität der Signal-Handler berichten?

(h) Modifizieren Sie das Programm `sig_empf2.c` so zu einem Programm `sig_empf3.c`, dass während der Signalbearbeitung von `SIGUSR1` keine Signale `SIGUSR2` und `SIGALRM` wahrgenommen werden. Verfahren Sie erneut entsprechend (f) und (g).

Aufgabe 8.4:

In dieser Teilaufgabe werden auf shell-Ebene eine Named Pipe erzeugt sowie Erzeuger/Verbraucher-Prozesse gestartet, die über die Pipe kommunizieren (vgl. Kap. 6.2.6).

(a) Erstellen Sie mittels des Dienstprogramms `mknod -p <name>` eine Named Pipe. (Achtung: Dienstprogramm und Systemaufruf haben gleichen Namen)! Überprüfen Sie den Typ des erzeugten Eintrags mittels `ls`.

(b) Schreiben Sie ein C-Programm `erzeug.c`, das die Pipe zum Schreiben öffnet (Flag `O_WRONLY`) und zyklisch ITERATIONS=20 mal Nachrichten in der Pipe ablegt. Eine Nachricht soll die Form <pid>:<i> haben, wobei pid die Prozess-Id des Prozesses angibt und i ein Nachrichtenzähler ist. (Schließen Sie jede Nachricht mit "`\n`" ab). Nach jedem Ablegen einer Nachricht soll der Prozess für SLEEP_TIME=2 sec schlafen.

(c) Schreiben Sie ein C-Programm `verbr.c`, das die Pipe zum Lesen öffnet (Flag `O_RDONLY`) und zyklisch aus der Pipe liest und die gelesenen Nachrichten auf dem Bildschirm ausgibt. (Der Einfachheit halber kann der Verbraucher zeichenweise lesen und ausgeben. Beachten Sie, wann die EOF-Bedingung (end-of-file) auftritt.)

(d) Starten Sie Erzeuger und Verbraucher im Hintergrund aus einer Datei `run`:

```
erzeug &
verbr &
```

(e) (Optional zum Experimentieren): Versuchen Sie, die Kapazität einer Pipe experimentell zu ermitteln. Idee: Öffnen Sie die Pipe als Schreiber im Nonblocking-Mode. Legen Sie Zeichen für Zeichen ab, bis Sie blockieren würden.

Aufgabe 8.5:

In dieser Teilaufgabe werden mehrere Erzeuger-Prozesse verwendet. Ferner soll eine abzulegende Nachricht aus mehreren Teilnachrichten bestehen. Modifizieren Sie das Programm `erzeug.c` aus Aufg. 8.4, so dass:

(a) dem Programm beim Aufruf über `argv[]` eine Nummer und zwei Wartezeit-Parameter sleep1 und sleep2 übergeben werden.

(b) In jedem Iterationsschritt soll die in der Pipe abzulegende Nachricht aus PARTS=3 Teilnachrichten bestehen, die einzeln in die Pipe geschrieben werden und folgenden Aufbau haben: `pid:i:p`, wobei p den Teil der Nachricht bezeichnet. Jede Teilnachricht soll später auf einer separaten Zeile erscheinen. Nach jeder Teilnachricht soll der Prozess für sleep2 sec (z.B. 1 sec) schlafen.

(c) Nach Ausgabe aller Teilnachrichten einer Nachricht schlafe der Prozess für sleep1 sec (sleep1 entspricht damit SLEEP_TIME in Aufgabe 8.4).

(d) Nutzen Sie die übergebene Nummer, um in der Ausgabe durch den Verbraucher ein unterschiedliches Einrücken der Teilnachrichten der verschiedenen Prozesse zu erreichen (z.B. 5*Nummer Leerzeichen zu Beginn der Zeile mit der Teilnachricht).

(e) Erweitern Sie das `run`-Skript entsprechend.

(f) Beobachten Sie die Ausgabe für verschiedene Wertepaare von sleep1 und sleep2. Erscheinen die Teilnachrichten einer Nachricht immer in aufeinander folgenden Zeilen? Welche Rückschlüsse ziehen Sie in Bezug auf die Unteilbarkeit der Operationen?

(g) Was könnten Sie tun, um alle Teilnachrichten einer Nachricht atomar abzulegen? Überlegen und implementieren Sie eine Lösung auf der Basis von Semaphoren unter Nutzung Ihrer Bibliothek aus Aufgabe 6.2.

Aufgabe 8.6:

Verwenden Sie in dieser Aufgabe statt der Named Pipe aus Aufgabe 8.4 eine übliche anonyme Pipe. Die kommunizierenden Prozesse Erzeuger und Verbraucher sollen von einem gemeinsamen Vater erzeugt werden, der vor den fork-Aufrufen die pipe mittels `pipe()` erzeugt und an seine Kinder vererbt. Schließen Sie die nicht benötigten Enden der Pipe vor der Überlagerung durch das Erzeuger- bzw.Verbraucher-Programm.