Hochschule RheinMain
Fachbereich DCSM - Informatik
Prof. Dr. Robert Kaiser
Sebastian Flothow
Alexander Schönborn
Daniel Schultz

Betriebssysteme

WS 2020/21
LV 3122

Aufgabenblatt 4
Bearbeitungszeit 2 Wochen
Abgabetermin: 25.01.2021, 4:00 Uhr

In diesem Praktikum werden UNIX-Systemaufrufe zur Signalisierung zwischen Prozessen sowie zur UNIX-Interprozesskommunikation mittels Pipes genutzt. 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 der Vorlesung). Konstanten, Strukturen usw. sind in der Include-Datei `<signal.h>` enthalten.

Pipes erlauben auf UNIX-Systemen eine 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 interner 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 diese Pipe erlaubt ausschließlich die Kommunikation zwischen verwandten Prozessen. Dazu wird die anonyme Pipe von einem (Eltern-)Prozess mittels `pipe()` erzeugt. Ihm stehen (zunächst relativ sinnlos) beide Enden der Pipe als File-Deskriptoren geöffnet zur Verfügung. Nach der Erzeugung eines Kind-Prozesses ist es diesem ebenfalls möglich, die Enden der Pipe über dieselben Dateideskriptoren zu nutzen. (Die Menge der offenen Datei-Deskriptoren wird bei `fork()` an das Kind vererbt). Damit ist eine Kommunikation zwischen Eltern-Prozess und Kind, bzw. im Falle mehrerer erzeugter Kinder 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 6-27ff) zur Verfügung. Konstanten, usw. sind in der Include-Datei `<fcntl.h>` enthalten.

Aufgabe 4.1 (Signale Empfangen und Blockieren):

Eilmeldung! Computerprobleme bei der Zwergenbank! Droht der nächste Bankencrash? Eil...

Die Zwergenbank benutzt eine einfache Datenbank zur Verwaltung ihrer insgesamt 100 Kundenkonten. Für jedes Konto wird der Name des Kontoinhabers und der aktuelle Kontostand gespeichert. Die Bank erlaubt Ein-/Auszahlungen und Überweisungen zwischen Konten innerhalb der Bank. Der maximale Überweisungsbetrag beträgt 1000 Goldstücke. Die Bilanzsumme der Bank (und damit auch die Summe aller Kontostände) beträgt 10000 Goldstücke.

Durch einen Bedienfehler eines Bankmitarbeiters ist eine Finanztransaktion nicht korrekt durchgeführt worden, und ein Betrag von 3 Goldstücken konnte nur mit großen Mühen wieder korrekt verbucht werden. Aufgrund dieses Vorfalls hat die Bankenaufsicht in Zwergenfurt einen sogenannten *Stresstest* für die Zwergenbank angeordnet.

Erweitern Sie dazu das C-Programm `db_stress.c`, welches folgende Grundstruktur besitzt:

- Das Programm arbeitet 1000 zufällige Überweisungen auf einer Datenbank ab.
- Eine Überweisung dauert zwischen 300 und 800 ms.
- Die einzelnen Datenbank-Transaktionen müssen unbedingt konsistent bleiben und dürfen nicht abgebrochen werden.
- Am Programmende berechnet das Programm die Bilanzsumme aller Konten. Diese muss weiterhin 10000 Goldstücke betragen.

Im Grundgerüst (ohne Signal-Behandlung) ist das Programm noch nicht sicher. Sie können das Programm jederzeit per `STRG+C` abbrechen. Erweitern Sie das Programm wie folgt:

a) Das Programm gibt am Anfang seine Prozess-ID aus, damit wir später von der Shell aus Signale an das Programm senden können. Visualisieren Sie den Fortschritt des Programms. Lassen Sie das Programm nach jeder Transaktion einen Punkt ausgeben.

b) Starten Sie das Programm mehrfach und brechen Sie die Ausführung mit `STRG+C` ab. Können Sie Fehler in der Datenbank entdecken?

c) Wenn Sie auf der Konsole `STRG+C` drücken, wird das Signal `SIGINT` an das aktuell laufende Programm geschickt. Damit das Programm während einer laufenden Transaktion nicht versehentlich beendet wird, soll dieses Signal mittels `sigprocmask()` während der Ausführung *einer* Transaktion kurzfristig blockiert werden.

d) Mit `sigaction()` können Funktionen registriert werden, die bei eingehenden Signalen ausgeführt werden sollen. Beim Empfang des Signals `SIGINT` soll das Programm den Text „Abbruch“ sowie die aktuelle Bilanzsumme der Datenbank ausgeben und sich danach beenden.

e) Beim Empfang des Signals `SIGUSR1` soll das Programm den Text „Info“ und die aktuelle Bilanzsumme der Datenbank ausgegeben. Da `SIGUSR1` während einer Überweisung noch nicht masktiert wird, können Sie Inkonsistenzen beobachten. Lösen Sie das Problem.

f) Nach Vorgaben der Bankenaufsicht soll der Stresstest die Datenbank zyklisch alle 5 Sekunden auf Konsistenz hin prüfen. Einen Timer können Sie mit `alarm()` realisieren. Beim Empfang des Signals `SIGALRM` soll das Programm den Text „*“ ausgeben und die Bilanzsumme prüfen. Lösen Sie eventuelle Inkonsistenzen bei der Ausführung.

g) Verhindern Sie, dass sich die Signale während ihrer Ausführung gegenseitig unterbrechen.

Hinweise:

- Nach der Ausgabe von „.“ oder „*“ `fflush(stdout)` nicht vergessen!
- `$ ./db_new` erzeugt eine neue Datenbank mit gültigen Kontoständen.
- `$ ./db_dump` zeigt den aktuellen Inhalt der Datenbank und die Bilanzsumme an.
- `$ kill -SIUGUSR1 <pid>` schickt das Signal `SIGUSR1` an den Prozess mit der angegebenen PID. Sie können nicht reagierende Prozesse mit SIGTERM beenden.

Aufgabe 4.2 (Anonyme Pipe):

Schreiben Sie ein Programm `filter.c`, welches in einem per `argv[1]` übergebenen Verzeichnis die Dateien auflistet, die einem ebenfalls per `argv[2]` übergebenen Suchmuster entsprechen:

```
$ ./filter <Verzeichnis> <Suchmuster>
```

Dabei sollen mittels einer anonymen Pipe die Ausgaben eines ersten Kind-Prozesses (hier `/bin/ls -la` *<Verzeichnis>*) als Eingaben an einen zweiten Kind-Prozess (hier `/bin/grep` *<Muster>*) weitergeleitet werden, ähnlich wie bei der Verwendung des Pipe-Symbols „`|`“ auf der Shell:

```
$ ls -la /home/staff | grep profs
...
drwx--x--x 113 kaiser       profs          12288 Dez 15 14:19 kaiser
drwxr-xr-x  42 kroeger      profs           4096 Jan 21  2014 kroeger
...
```

Die zwei kommunizierenden Kind-Prozesse sollen nun von einem gemeinsamen Eltern-Prozess `filter` erzeugt werden, der vor den `fork()`-Aufrufen die Pipe mittels `pipe()` erzeugt und an seine Kinder vererbt.

Dann müssen Sie in den beiden Kind-Prozessen die File-Deskriptoren für die Ein- (0) bzw. Ausgabe (1) mittels `dup2()` auf die jeweiligen Pipe-Enden „umbiegen“.

Schließen Sie danach die nicht benötigten Enden der Pipe in den Kind-Prozessen vor der Überlagerung des Filter-Programms durch `ls` bzw. `grep` sowie im Eltern-Prozess.

Achten Sie darauf, dass der Eltern-Prozess auf die Beendigung seiner Kinder warten muss.