182.709 Betriebssysteme UE
2. Test

22. Jänner 2018

Matrikelnummer

Zuname, Vorname

Ges. (50)	1.)(30)	2.)(20)

Allgemeines

Kopieren Sie sich die Angaben des Beispiels mit dem Shellkommando

`$ fetch <1|2>` (z.B. `fetch 1`)

in Ihr Homeverzeichnis, wobei 1 bzw. 2 die Nummer des Beispiels ist, das Sie in Folge bearbeiten möchten. Sie können `fetch` mehrmals ausführen. Eventuelle Änderungen werden gesichert (`<Datei>`→`<Datei>.~1~`).

Um Ihre Lösung zu erstellen, verwenden Sie `make`. Testen und debuggen Sie Ihr Programm wie gewohnt. Auch `gdb` steht Ihnen zur Verfügung. Bestehenden Shared Memory und bestehende Semaphore können Sie mit dem Befehl `cleanup` wieder freigeben.

Um Ihre Lösung zu prüfen, führen Sie

`$ deliver <1|2>` (z.B. `deliver 1`)

aus. Sie können `deliver` beliebig oft ausführen. Sind Sie mit der Bewertung **beider** Beispiele zufrieden, melden Sie sich bei der Aufsicht, die das Ergebnis entgegennimmt und Sie ausloggen wird.

Sie können den Prüfungsbogen für Notizen verwenden. Diese Eintragungen werden nicht bewertet.

Beachten Sie weiters folgende bei beiden Beispielen angewandten Bewertungskriterien:

- Ihr Programm muss ohne Fehler kompilieren, sonst erhalten Sie keine Punkte.
- Ihr Programm muss in jedem Fall ohne Segmentation Fault (Speicherzugriffsverletzung) terminieren, sonst erhalten Sie keine Punkte.
- Compiler Warnings (das Programm wird mit `-Wall` übersetzt) führen zu Punkteabzügen.
- Deaktivieren Sie vor der Bewertung durch `deliver` jegliche Debugausgaben auf `stdout`! Die Bewertung erfolgt unter anderem durch Prüfung der Ausgaben der Programme.

Fork, Exec, Unnamed Pipes (30)

Implementieren Sie die Prüfziffern-Validierung einer IBAN. Vervollständigen Sie dazu das Programm in `validate.c`.

Beispiele

```
$ ./validate GB29NWBK60161331926819
valid
$ ./validate AT0012345678901234567 8
invalid
```

Weitere *gültige* IBANs zum Testen: `AT1800`, `DE3600`, `GB22YK`.

Hinweise:

- Es steht Ihnen die Funktion `error_exit(..)` zur Verfügung, die Sie im Fehlerfall aufrufen können (beendet das Programm mit `EXIT_FAILURE`).

Dieses Beispiel ist in drei Teilaufgaben unterteilt (Datei: `validate.c`). Die Teilaufgaben sind jeweils in den Funktionen `task_1`, `task_2`, `taks_3` zu lösen. Vergessen Sie nicht die Demolösung auszukommentieren.

Task 1: Vorbereitung zur IBAN Validierung (8 Punkte)

Um die IBAN für die Validierung vorzubereiten sind 3 Schritte notwendig. In untenstehender Tabelle wird dies am Beispiel `GB29NWBK60161331926819` gezeigt. Die IBAN steht in der Variable `iban` bereits zur Verfügung.

Schritt	Ihre Aufgabe	Ergebnis
1	Verschieben Sie die ersten 4 Zeichen (Länderkennung und Prüfziffer, jeweils zweistellig) ans Ende der IBAN.	`NWBK60161331926819GB29`
2	Ersetzen Sie *alle* Buchstaben durch Zahlen, wobei `'A'`→`"10"`, `'B'`→`"11"`, .., `'Z'`→`"35"`.	`233211206016133192681916112916112 9`
3	Schreiben Sie die umgewandelte IBAN bzw. die Berechnung in einen String (die Variable `expr` ist dafür vorgesehen).	`233211206016133192681916112 9 % 97`

Die IBAN ist gültig (`"valid"`), wenn die Berechnung = 1 ergibt, sonst ungültig (`"invalid"`).

Für die Berechnung dieser großen Zahl (die nicht in einen 64bit-Integer passt), steht das Programm `./calc` zur Verfügung. In weiterer Folge sollen Sie `./calc` als Kindprozess starten.

Task 2: Fork und Unnamed Pipes (6 Punkte)

Erstellen Sie einen Kindprozess. Falls dies fehlschlägt, beenden Sie das Programm mit dem Fehlercode `EXIT_EFORK`.

Im Kindprozess rufen Sie die Funktion `task_3(fd, expr)` auf (das Argument `expr` muss die umgewandelte IBAN beinhalten).

Der Elternprozess soll das Resultat von der Pipe (Variable `fd`) *lesen* (die Variable `result` ist dafür vorgesehen, welche am Ende des Programms mit 1 verglichen wird).

Task 3: Exec und Unnamed Pipes (8 Punkte)

Führen Sie das Programm `./calc` mit `exec*(..)` aus, um die Berechnung `EXPRESSION` durchzuführen. Wenn `./calc` fehlschlägt, beenden Sie den Prozess.

```
SYNOPSIS:
  ./calc EXPRESSION
```

`./calc` *schreibt* das Ergebnis nach Standard Output (File Descriptor `STDOUT_FILENO`). Leiten Sie daher Standard Output auf die Pipe (Variable `fd`) um.

Beispiele `./calc`

```
$ ./calc "233211206016133192681916112916112 9 % 97"
1
$ ./calc 233211206016133192681916112916112 9 % 97
1
$ ./calc 100%97
3
$ ./calc "100" "%" "97"
3
```

Notiz: Die Implementierung der Kommunikation über die Pipe ist Teil von `task_2` und `task_3`. Beim Aufruf von `deliver 1` wird die Kommunikation aber separat getestet (Black-Box-Tests werden als “Task 4” ausgeführt).

Synchronisation (20)

In diesem Beispiel sollen Sie einen Server einer Bank vervollständigen, der Bankkonten verwaltet. Clients können sich mit dem Server verbinden und Beträge auf Bankkonten einzahlen, bzw. von Bankkonten abheben. Dieses Service kann von mehreren Clients gleichzeitig genutzt werden. Server und Client kommunizieren dabei über ein Shared Memory. Ein Client schreibt eine IBAN, einen Betrag sowie die Art der Buchung in das Shared Memory. Der Server führt die Buchung durch und ersetzt den Betrag mit dem aktuellen Kontostand und schreibt das Resultat zurück in das Shared Memory. Der Client gibt anschließend die erhaltene Kontoinformation aus. Dazu müssen Client und Server den Zugriff auf das Shared Memory synchronisieren, wobei Folgendes zu beachten ist:

- Ein Client darf seine Anfrage erst dann in das Shared Memory schreiben, wenn der vorherige Client die Antwort des Servers ausgelesen hat.
- Der Server darf mit der Verarbeitung erst dann beginnen, wenn der Client seine Anfrage vollständig in das Shared Memory geschrieben hat.
- Der Client darf die Antwort erst dann auslesen, wenn der Server die Anfrage abgearbeitet hat und seine Antwort in das Shared Memory geschrieben hat.

Dieses Beispiel ist in drei Teilaufgaben unterteilt (Datei: `server.c`). Die Teilaufgaben sind jeweils in den Funktionen `task_1a`, `task_1b`, `task_2`, `task_3` zu lösen. Vergessen Sie nicht die Demolösung auszukommentieren. Um belegte Ressourcen wieder freizugeben, steht Ihnen der Befehlt "cleanupßur Verfügung.

Task 1: Anlegen gemeinsamer Ressourcen (10 Punkte)

Legen Sie in `task_1a` einen gemeinsamen Speicher (POSIX Shared Memory) geeigneter Größe an (siehe `common.h`) und mappen diesen in den Addressraum des Servers. Außerdem erzeugen Sie in `task_1b` die für die Synchronisation notwendigen Semaphore (POSIX Named Semaphore) mit korrekten Initialwerten.

Das Anlegen der gemeinsamen Ressourcen soll fehlschlagen, falls diese bereits existieren. Es müssen die Namen und `PERMISSIONS` aus `common.h` verwendet werden.

Task 2: Synchronisation mittels Semaphoren (5 Punkte)

Vervollständigen Sie die Service-Loop (`task_2`), welche Anfragen der Clients nacheinander bearbeitet. Dazu platzieren Sie die notwendigen Semaphoroperationen vor und nach einem Aufruf von `task_3(shmp);`. Das Synchronisationsprotokoll ist im Wesentlichen durch den Client vorgegeben und kann dem Pseudo-Code in `clientps.c` entnommen werden.

Um den Server zu terminieren, werden die Signale `SIGTERM` oder `SIGINT` verwendet. Der vorgegebene Signal-Handler setzt ein Flag `quit`, das in der Bedingung der Service-Loop geprüft wird. Der Server terminiert dann mit `EXIT_SUCCESS`. Berücksichtigen Sie daher Signale in der Fehlerbehandlung für die eingefügten Semaphoroperationen!

Hinweis:

- Voraussetzung für Task 2 ist ein vorangegangenes korrektes Anlegen gemeinsamer Ressourcen (Task 1). Bearbeiten Sie diesen Task nur, wenn Sie Task 1 korrekt implementiert haben oder mit Verwendung der Musterlösung zu Task 1.

Task 3: Buchung durchführen (5 Punkte)

Implementieren Sie die Durchführung der Buchung in der Funktion `task_3`.

Die Buchungs-Anfrage eines Clients besteht dabei aus der IBAN `iban`, der Durchführungsart `cmd` sowie den zu buchenden Betrag `amount` (siehe `common.h`).

Suchen Sie in dem Array an verfügbaren Konten `bank_accounts` (siehe `server.h`) nach der IBAN. Ist das Konto vorhanden, führen Sie die Buchung durch, d.h. verändern Sie den aktuellen Kontostand `balance` gemäß der Anfrage.

Schreiben Sie den neue Kontostand ins Shared-Memory nach `amount`. Wird die IBAN nicht gefunden, soll der Server `-1` in `amount` schreiben um einen Fehler zu symbolisieren.

Verfügbare Kontodaten zum Testen finden Sie in `bank_accounts.txt`.

Beispiel Eine Client-Anfrage könnte wie folgt aussehen:

`"AT1212345123456789O1"`	`char iban[LEN_IBAN]`
`"WITHDRAW"`	`bank_account_cmd_t cmd`
`"100"`	`int amount`

Angenommen der Kontostand beträgt 1000 (EUR), dann sieht der Shared-Memory Inhalt nach der Bearbeitung durch den Server wie folgt aus:

`"AT1212345123456789O1"`	`char iban[LEN_IBAN]`
`"WITHDRAW"`	`bank_account_cmd_t cmd`
`"900"`	`int amount`

Hinweis:

- Voraussetzung für Task 3 ist ein korrektes Anlegen der gemeinsamen Ressourcen (Task 1) und eine korrekte Synchronisation (Task 2). Bearbeiten Sie diesen Task nur, wenn Sie Task 1 und 2 korrekt implementiert haben oder mit Verwendung der Musterlösungen von Task 1 und 2.

Synopsis und Beispiele

Zuerst muss der Server gestartet werden, bevor der Client (ggf. in einem weiteren Terminal) ausgeführt werden kann.

```
$ ./server
```

Der Client ist bereits vorgegeben und kann nicht modifiziert werden. Sie können eine Client-Anfrage erzeugen, indem Sie den Client ausführen:

```
$ ./client
Usage: ./client IBAN -d|-w AMOUNT
```

wobei `IBAN` eine gültige IBAN ist, `-d` zum Einzahlen (deposit) bzw. `-w` zum Auszahlen (withdraw) des gewünschten Betrages `AMOUNT` verwendet wird. Die Ausgabe des Clients könnte wie folgt aussehen:

```
$ ./client AT121234512345678901 -w 100
./client: waiting to become next client
./client: writing request
./client: release server
./client: waiting for server
./client: reading result
./client: New Balance of "AT121234512345678901": 900
./client: releasing next client
./client: detach shared memory
```

Der Server kann z.B. durch Senden von `SIGINT` (Strg-C) beendet werden:

```
^Cserver exiting regularly
```

Beachten Sie, dass Server-Instanzen bei Ausführung von `deliver` beendet werden.