
Diese Testangabe kann minimal Fehler/Halluzinationen beinhalten, wurde von AI in Textform umgewandelt.
Sollte aber grundlegend passen.

Testumgebung

Im Verzeichnis `/exam` finden Sie die C-Quelldateien, in welchen Sie Ihre Lösung implementieren sollen. Sie können jederzeit die Original-Dateien mit dem Shellkommando

```
$ fetch
```

wiederherstellen. Änderungen, die Sie an den Dateien vorgenommen haben, werden gesichert (`<Datei>-><Datei>~1`).

Im Verzeichnis befindet sich auch ein Makefile. Um Ihre Lösung zu kompilieren, verwenden Sie den Befehl `make`. Testen und debuggen Sie Ihr Programm wie gewohnt. Auch `gdb` steht Ihnen zur Verfügung. Übrig gebliebene Ressourcen (wie Semaphoren oder Shared Memory) können Sie mit dem Kommando `cleanup` entfernen.

Um Ihre Lösung zu prüfen, führen Sie

```
$ deliver
```

aus. Sie können `deliver` beliebig oft ausführen. Dabei wird Ihnen auch angezeigt, wie viele Punkte Sie derzeit auf Ihre Lösung erhalten würden.

Wenn Sie mit der Bewertung zufrieden sind, dann melden Sie sich bei der Aufsicht, die das Ergebnis entgegennimmt und Sie ausloggen wird.

Der Test besteht aus mehreren Aufgaben, die Sie **unabhängig voneinander und in beliebiger Reihenfolge** implementieren können. Für jede Aufgabe steht Ihnen eine DEMO-Funktion zur Verfügung, die Sie in der `main()`-Funktion einkommentieren können, wenn Sie eine Aufgabe überspringen möchten.

Beachten Sie folgende Bewertungskriterien:

- Ihr Programm muss ohne Fehler kompilieren, sonst erhalten Sie keine Punkte.
- Compiler-Warnings (das Programm wird mit `-Wall` übersetzt) führen zu Punkteabzügen.
- Bei Segmentation Faults (Speicherzugriffsverletzung) erhalten Sie für die entsprechende Aufgabe keine Punkte.

Einleitung

Implementieren Sie den Server eines Bankkonto-Transaktionssystems, der Transaktionen von Clients entgegennimmt und sicherstellt, dass die Kontostände korrekt aktualisiert werden. Die Client-Applikation ist bereits implementiert und wird Ihnen zur Verfügung gestellt. Ihre Aufgabe ist es, fehlende Funktionen des Servers zu vervollständigen.

Die Kommunikation zwischen Ihrem Server und den Clients erfolgt ausschließlich über einen gemeinsamen Speicher (en.: shared memory) und wird mittels benannter Semaphore (en.: named semaphores) synchronisiert:

- `sem_request`: Dient dazu, eine Buchungsanfrage eines Clients an den Server zu signalisieren.
- `sem_response`: Dient dazu, den Abschluss der Bearbeitung einer Buchungsanfrage durch den Server an den Client zu signalisieren.
- `sem_transaction`: Erlaubt es den wechselseitigen Ausschluss (en.: mutual exclusion) für die Änderung des Kontostands zu gewährleisten.

Der Ablauf einer Buchungsanfrage aus Sicht des Clients ist wie folgt:

1. Der Client initiiert eine Buchungsanfrage, indem er die Semaphore `sem_transaction` dekrementiert.
2. Anschließend schreibt der Client die für die Buchung notwendigen Daten in den gemeinsamen Speicher.
3. Der Client inkrementiert die Semaphore `sem_request`, um den Server die Bearbeitung der Anfrage zu signalisieren.
4. Der Client wartet auf die Antwort des Servers, indem er auf die Semaphore `sem_response` wartet.
5. Sobald die Semaphore `sem_response` dekrementiert werden kann, liest der Client das Ergebnis aus dem gemeinsamen Speicher und gibt die Semaphore `sem_transaction` wieder frei.

Client

```
wait(sem_transaction);
write to shared memory;
post(sem_request);
```

Server

```
...
server processes request;
...
```

Client

```
wait(sem_response);
read response;
post(sem_transaction);
```

1 Argumentbehandlung (30)

Vervollständigen Sie die Funktion `void parse_arguments(int argc, char *argv[], struct arguments *args)` in `transaction_server.c` mit der Argumentbehandlung. Die Funktion soll die übergebenen Parameter in der Struktur `args` speichern (siehe `transaction_server.h` für die Definition).

Synopsis

```
./transaction_server [-b BALANCE] [HISTORY]
```

Der Parameter `BALANCE` ist optional und gibt den Kontostand an. Der Kontostand ist ein positiver Integer-Wert ($b \geq 0$). Falls der Parameter `BALANCE` nicht übergeben wird, ist der Kontostand auf den Wert 0 zu setzen. Die Überprüfung, ob der Kontostand ein positiver Integer-Wert ist, ist in der Funktion `parse_arguments` durchzuführen.

Die Transaktionshistorie wird in eine Datei geschrieben, die als optionales Positionsargument `HISTORY` übergeben wird. Falls das Argument `HISTORY` nicht übergeben wird, ist der Pfad `history.txt` zu verwenden.

Im Falle eines fehlerhaften Aufrufs soll das Programm mit einer Usage-Meldung und dem Exit-Code `EXIT_FAILURE` terminieren. Rufen Sie dazu die Funktion `usage(char *msg)` auf.

Hinweise:

- Zum Parsen der Argumente kann die Funktion `getopt(3)` verwendet werden.
- Die Funktion `strtol(3)` kann verwendet werden, um einen String in einen Integer umzuwandeln.

1.1 Beispiele

Gültige Aufrufe:

```
./transaction_server
./transaction_server -b 1000
./transaction_server account_01.txt
./transaction_server -b 100000 account_02.txt
```

Ungültige Aufrufe:

```
./transaction_server -b
./transaction_server: option requires an argument -- 'b'
Usage: ./transaction_server [-b BALANCE] [HISTORY]
invalid option
./transaction_server file1 file2 file3
Usage: ./transaction_server [-b BALANCE] [HISTORY]
too many positional arguments
```

2 Gemeinsame Ressourcen anlegen (35)

Vervollständigen Sie die Funktion `void allocate_resources(const char *path)` mit dem Erzeugen des gemeinsamen Speichers und der Semaphoren (siehe `transaction_server.c`).

Legen Sie ein POSIX Shared Memory Objekt als gemeinsamen Speicher an. Nachdem der gemeinsame Speicher erzeugt wurde, soll dieser in den Speicher des Prozesses gemapped werden. Der File Descriptor des gemeinsamen Speichers soll in der Variable `shmfd` gespeichert werden und die Adresse des Mappings in der Variable `shmp`. Achten Sie darauf, dass sowohl der gemeinsame Speicher als auch dessen Mapping Lese- und Schreibzugriffe erlauben.

Weiters müssen 3 POSIX Semaphore angelegt werden. Die Adressen der Semaphore-Objekte sollen in den Variablen `sem_request`, `sem_response` und `sem_transaction` gespeichert werden, entsprechend den Bezeichnungen in der Einleitung. Überlegen Sie sich, auf welche Werte die Semaphore initialisiert werden müssen, damit die Kommunikation so erfolgen kann, wie in der Einleitung dargestellt.

In der Datei `common.h` sind bereits Macros für die Namen des geteilten Speichers, der geteilten Semaphore und der nötigen Permission definiert. Außerdem finden Sie die Definition der Datenstruktur `transaction_t`, die für den geteilten Speicher verwendet wird. Falls bereits ein gemeinsamer Speicher oder eine Semaphore mit gleichem Namen existiert, soll Ihr Server mit einem Fehlerstatus terminieren.

Schließlich soll die Datei mit dem im Parameter `path` übergebenen Dateipfad zum Schreiben geöffnet werden und eine Referenz auf das mit dieser Datei assoziierte Stream-IO `FILE`-Objekt in der Variable `fp_out` hinterlegt werden.

Falls es zu einem Fehler kommt, soll das Programm mit dem Exit-Code `EXIT_FAILURE` beendet werden. Sie können dazu die Funktion `error_exit("my err msg")` verwenden.

Hinweise:

- Unter `shm_overview(7)` finden Sie einen Überblick des POSIX Shared Memory und unter `sem_overview(7)` einen Überblick der POSIX Semaphore.
- Um die Größe des geteilten Speichers zu setzen wird `ftruncate(2)` verwendet, das Mapping kann mit `mmap(2)` erstellt werden.

3 Transaktionen Durchführen (35)

Implementieren Sie nun in der Funktion `void process_transactions(int balance)` die Transaktionsabwicklung (siehe `transaction_server.c`). Der initiale Kontostand wird mit dem Funktionsargument `balance` übergeben.

Der Server soll in einer Schleife wiederholt auf Anfragen von Clients warten und Buchungsanfragen verarbeiten. Für die Kommunikation mit den Clients wird das in der Einleitung erklärte Synchronisations-Protokoll verwendet.

Warten Sie zunächst auf die Anfrage eines Clients. Sobald diese erfolgt ist, überprüfen Sie, ob die Transaktion, die im gemeinsamen Speicher hinterlegt wurde, zulässig ist. Eine Transaktion ist dann zulässig, wenn der Kontostand nach der Durchführung nicht negativ ist und der Betrag der Transaktion nicht 0 ist. Wenn die Transaktion zulässig ist, schreiben Sie die Buchungszeile im CSV-Format in die Datei, die durch den Zeiger `FILE *fp_out` referenziert wird. Eine Buchungszeile hat das folgende Format:

```
<id>;<amount>;<balance>;<new-balance>;<message>\n
```

Die ID einer Buchungszeile ist eine fortlaufende Zahl, beginnend bei 0, und wird jedes Mal nach einer erfolgreichen Buchung um 1 erhöht. Die Buchungszeile enthält außerdem den Geldbetrag der Transaktion, den Kontostand vor der Durchführung, den neuen Kontostand sowie den Nachrichtentext (siehe `transaction_t` in `common.h`). Im Anschluss wird an die Zeile noch ein Zeilenumbruch (Newline-Zeichen, `\n`) angehängt. Nachdem Sie die Buchungszeile geschrieben haben, aktualisieren Sie den Kontostand sowie die ID der Transaktion und setzen den Status der Transaktion auf den Wert `TRANSACTION_STATUS_SUCCESS`. Falls eine Transaktion unzulässig ist, setzen Sie den Status auf `TRANSACTION_STATUS_FAILED` und die ID des Clients bleibt unverändert.

In diesem Fall wird keine Buchungszeile geschrieben. Ein Beispiel für eine zulässige und eine unzulässige Buchungszeile finden Sie in untenstehender Tabelle.

Abschließend informiert der Server den Client über die erfolgreiche Bearbeitung seiner Anfrage mittels des bereits erklärten Synchronisations-Protokolls. Falls es zu einem Fehler kommt, soll das Programm mit dem Exit-Code `EXIT_FAILURE` beendet werden. Sie können dazu die Funktion `error_exit("my err msg")` verwenden.

Buchungstext	Betrag	Buchungszeile
Gehalt	-20.000€	0;20000;0;20000;Gehalt
Lebensmittel	-1.000€	1;-100;20000;19900;Lebensmittel
Villa	-1.000.000€	- (keine Ausgabe)
Miete	-1.000€	3;-1000;19900;18900;Miete

Hinweise:

- Die Funktion `fprintf(3)` kann verwendet werden, um formatierten Text in eine Datei zu schreiben.

Implementierung Testen

Um Ihre Implementierung zu testen, steht Ihnen das Client-Programm `client` zur Verfügung. Diesem Programm übergeben Sie den Pfad zu einer Datei mit durchzuführenden Transaktionen (z. B. `transactions_01.txt`), welche Zeile für Zeile abgearbeitet werden. Um Ihre Implementierung zu testen, können Sie natürlich eigene Testfälle erzeugen, indem Sie eine Textdatei anlegen, in der jede Zeile eine Transaktion darstellt:

```
<message>:<amount>
```

Synopsis

```
./client TRANSACTIONS
```

Der Client übergibt die durchzuführenden Buchungen an den Server.

Verwendungs-Beispiel:

```
./transaction_server &
[1] 4025
$ ./client transactions_01.txt
[./client] waiting for server
[./client] writing request
[./client] release server
[./client] Transaction 00 successful.
[./client] releasing next client
...
[./client] detach shared memory
```

Hinweis:
Das `&` am Ende des Server-Aufrufs startet diesen als Hintergrundprozess. Mit dem Befehl `fg` kann ein Hintergrundprozess wieder in den Vordergrund geholt und dann mit `Ctrl+C` beendet werden:

```
$ fg
./transaction_server
^C
$
```