Praktikum

Synchronisationsprobleme

Kaffee–Automaten

Banking 

Frühlingssemester 2020

M. Thaler

Überblick

In diesem Praktikum lernen sie zuerst am Beispiel eines Kaffee-Automaten verschiedene grundlegende Synchronisationsprobleme kennen und mit Hilfe von Locks (Mutexes) und Semaphoren lösen:

- gegenseitiger Ausschluss mit einem Lock
- Erzwingen einer einfachen Reihenfolge
- Erzwingen einer erweiterten Reihenfolge

Im zweiten Teil werden sie auf Basis dieser Grundlagen ein komplexeres Synchronisationsproblem bearbeiten, diesmal am Beispiel von Bank Transaktionen.

Inhaltsverzeichnis

1 Einführung

Das Lösen von Synchronisationsproblemen ist oft nicht einfach, weil Prozesse bzw. Threads gleichzeitig ablaufen, ihre Aktivitäten jedoch nach Vorgaben koordiniert werden müssen: man verliert schnell den Überblick. Systematisches Vorgehen mit Aufzeichnen der Abläufe und Synchronisationsbedingungen bewährt ich sich in diesem Fall.

1.1 Wie löst man Synchronisationsprobleme?

Gehen sie beim Lösen von Synchronisationsproblemen in folgenden Schritten vor:

Schritt 1: Prozesse (Threads) der Problemstellung identifizieren. Prozesse sind die Aktivitäten, die gleichzeitig (konkurrent) ausgeführt werden. In diesem Sinne sind sie eigenständige Ausführungs-Einheiten, deren zeitliches Verhalten synchronisiert werden muss.

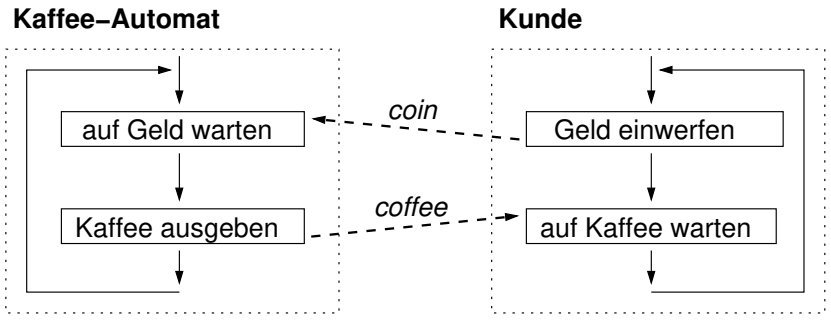
Schritt 2: Ausführungsschritte der einzelnen Prozesse (Threads) ermitteln. Erstellen sie eine Liste mit einer Spalte für jeden Prozess. Notieren sie für jeden Prozess stichwortartig die wesentlichen Aktionen in der gewünschten zeitlichen Reihenfolge. Tragen sie noch keine Synchronisationsoperationen ein, sondern Texte wie *warten auf Geld*, etc. Übertragen sie anschliessend die Liste in einen Ablaufgraphen (Siehe Beispiel in Fig. 1).

Schritt 3: Synchronisationsbedingungen ermitteln. Eine Synchronisationsbedingung ist eine zeitliche Beziehung (Abhängigkeit) zwischen Aktionen verschiedener Prozesse, die für das korrekte Arbeiten erforderlich ist. Zeichnen sie diese Beziehungen mit Pfeilen in den Ablaufgraphen aus Schritt 2 ein (Siehe Fig. 1).

Schritt 4: Benötigte Semaphore definieren. Für jede Synchronisationsbedingung wird ein eigener Semaphor benötigt. Notieren sie für jeden Semaphor einen Namen und den Wert, mit dem er initialisiert werden muss.

Schritt 5: Prozesse mit Semaphoroperationen ergänzen. Erweitern sie nun alle Prozesse aus Schritt 2 mit den notwendigen Semaphoroperationen (Siehe Pseudocode in Fig. 1).

Schritt 6: Implementation Implementieren und testen sie das vollständige Programm.




```
coin   = sem_open(...,0);
coffee = sem_open(...,0);

while (1) {                          while (1) {
      . . .                                . . .
    sem_wait(coin);        coin        sem_post(coin);
    sem_post(coffee);      coffee      sem_wait(coffee);
      . . .                                . . .
}                                    }
```

Fig. 1: Ablaufgraph und Pseudocode für 2 Prozesse und zwei Semaphore

2 Problemstellungen zur Synchronisation

2.1 Der Kaffee-Automat

Als Beispiel verwenden wir einen Automaten, der Kaffee verkauft. Der Kunde muss zum Kauf eines Kaffees zuerst eine bzw. mehrere Münzen einwerfen und anschliessend den gewünschten Kaffee wählen. Der Automat gibt dann das entsprechende Getränk aus.
Im ersten Beispiel werden der Automat und die Kunden mit Threads modelliert und tauschen Daten über gemeinsame Speichervariablen aus. Im zweiten und dritten Beispiel werden der Automat und die Kunden mit Prozessen modelliert, dabei wird der Ablauf mit Hilfe von Semaphoren gesteuert bzw. *erzwungen*.

Hinweis: die Programme zu den folgenden Aufgaben können alle mit **`startApp.e`** gestartet werden. Dieses Programm startet und stoppt Threads und Prozesse, alloziert und deallokiert die Ressourcen (Mutexes, Semaphore).

2.2 Mutual Exclusion

Greifen mehrere Threads (oder Prozesse) auf gemeinsame Daten zu, können sogenannte *Race Conditions* entstehen. Das Resultat ist in diesem Fall abhängig von der Reihenfolge, in der die Threads (Prozesse) ausgeführt werden.

Im vorliegenden Beispiel wirft der Kunde eine 1 Euro Münze ein und drückt anschliessend auf eine von zwei Kaffeewahltasten. Dabei wird die Anzahl Münzen (`coinCount`) und die gewählte Kaffeesorte (`selCount1, selCount2`) inkrementiert. Diese Variablen sind in der Datenstruktur `cData` abgelegt, auf die gemeinsam Kaffee-Automat und Kunden zugreifen können. Der Automat überprüft, ob die Anzahl Münzen und die Anzahl der Kaffeewahlen gleich gross sind, falls nicht, wird eine Fehlermeldung ausgegeben und alle Zähler auf Null gesetzt.

2.2.1 Aufgaben

a) Übersetzen sie die Programme im Verzeichnis `mutex` mit `make` und starten sie den Kaffee-Automaten mit `startApp.e` mehrmals hintereinander.

   Analysieren sie die Datenwerte in der Fehlermeldungen, beschreiben sie was die Gründe dafür sind bzw. sein können.

b) Schützen sie nun den Zugriff auf die gemeinsamen Daten mit einem Mutex so, dass alle Threads eine konsistente Sicht der Daten haben.

   Wir haben für sie einen Mutex vorbereitet: die Datenstruktur `cData` enthält die Mutex-Variable `mutex`, die in `startApp.c` initialisiert wird. Die Funktionen für das Schliessen und das Öffnen des Mutex (Locks) aus der Pthread Bibliothek sind:

   ```
   pthread_mutex_lock(&(cD->lock));      pthread_mutex_unlock(&(cD->lock));
   ```

   Überprüfen sie, ob der Kaffee-Automat nun keine Fehlermeldungen mehr ausgibt. Erhöhen sie dazu auch die Anzahl Kunden `CUSTOMERS` in `commonDefs.h`, z.B. auf 10.

c) Im Thread des Kaffee-Automaten wird an verschiedenen Orten mehrmals auf die gemeinsamen Daten in `cD` zugegriffen. Wenn sie die gemeinsamen Daten in lokale Variablen kopieren und dann nur noch auf diese lokalen Variablen zugreifen würden, könnten sie dann auf die Synchronisation mit dem Mutex verzichten?

d) Wie oft kann ein einzelner Kunde einen Kaffee beziehen, bis der nächste Kunde an die Reihe kommt? Hier reicht eine qualitative Aussage.

2.3 Einfache Reihenfolge

Wie sie im ersten Beispiel festgestellt haben, verhindert ein Mutex zwar dass *Race Conditions* auftreten, die Verarbeitungsreihenfolge der Threads lässt sich jedoch nicht beeinflussen und ist zufällig.

Im folgenden soll eine erzwungene Verarbeitungsreihenfolge implementiert werden:

Ein Kunde benutzt den Automat für einen Kaffekauf exklusiv, d.h. alle Schritte des Kunden werden innerhalb eines Mutex ausgeführt. Ist ein Kunde an der Reihe, wartet er bis der Automat bereit ist, wirft eine Münze ein, wartet auf den Kaffee und gibt anschliessend den Automaten für den nächsten Kunden frei.

Der Automat meldet zuerst in einer Endlos-Schleife, dass er für die Geld-Eingabe bereit ist, wartet dann auf die Eingabe einer Münze, gibt den Kaffee aus und meldet anschliessend wieder, wenn er bereit ist, etc.

Für die Lösung dieses Problems benötigen wir Semaphore, die, im Gegensatz zu Mutexes, auch in verschiedenen Prozessen gesetzt bzw. zurückgesetzt werden dürfen. Den Kaffee-Automat und die Kunden implementieren wir mit Prozessen. sie finden die entsprechenden Prozesse im Verzeichnis `basicSequence`.

2.3.1 Aufgaben

a) Beschreiben sie den Kaffee-Automaten mit Hilfe der 6 Schritte aus Abschnitt 1 auf Papier, dokumentieren sie dabei alle Schritte schriftlich.

b) Implementieren sie nun den Kaffee-Automaten. Ergänzen sie dazu den `coffeeTeller`- und den `customer`-Prozess so mit vier Semaphoren, dass die vorgegebenen Ablaufbedingungen eingehalten werden. Mit welchen Werten müssen die Semaphore initialisiert werden?

   Wir haben für sie vier Semaphore vorbereitet: Achtung, sie sind aber noch auskommentiert (siehe `commonDefs.h` und `startApp.c`. Die benötigten Semaphor-Funktionen aus der POSIX Bibliothek sind:

   ```
   sem_wait(&semaphor);        sem_post(&semaphor);
   ```

   Analysieren sie die Ausgabe der Prozesse (mehrmals starten). Was fällt auf?

c) Gibt Ihr Programm den Output in der korrekten Reihenfolge aus? Falls nicht, wie könnte das gelöst werden?

2.4 Erweiterte Reihenfolge

Die Preise steigen dauernd ... auch der Kaffee wird immer teurer, er kostet nun 3 Euro. Da der Automat nur 1 Euro Stücke annehmen kann, muss der Kunde 3 Münzen einwerfen. Erweitern sie die Prozesse aus Aufgabe 2.3 so, dass eine vordefinierte Anzahl Münzen eingegeben werden muss (die Anzahl Münzen ist in `commonDefs.h` als `NUM_COINS` definiert). Verwenden sie keine zusätzlichen Semaphore, sondern nutzen sie, dass wir *Counting Semaphore* verwenden. Die vordefinierten Prozesse finden sie im Verzeichnis `advancedSequence`.

2.4.1 Aufgabe

Passen sie den `coffeeTeller`- und den `customer`-Prozess so an, dass der Kunde mehrere Münzen einwerfen muss, bis der Automat einen Kaffee ausgeben kann.

Hinweis: POSIX Semaphore sind *counting semaphore*, können aber nicht auf vordefinierte Werte gesetzt werden (ausser bei der Initialisierung). Abhilfe schafft hier das mehrmalige Aufrufen von `sem_post()`, z.B. in einer for-Schleife.

2.5 Zusammenfassung

Wir haben drei grundlegenden Typen von Synchronistionsproblemen kennen gelernt:

Mutex nur ein Prozess bzw. Thread kann gleichzeitig auf gemeinsame Daten zugreifen.
Beispiel: entweder liest der Kaffee-Automat die Daten oder ein Kunde verändert sie.

Einfache Reihenfolge ein Prozess wartet auf die Freigabe durch einen anderen Prozess.
Beispiel: der Kaffee-Automat wartet auf die Eingabe einer Münze.

Erweiterte Reihenfolge ein Prozess wartet auf mehrere Freigaben durch einen anderen Prozess.
Beispiel: der Kaffee-Automat wartet auf die Eingabe von drei Münzen.

3 International Banking

Die *International Bank of Transfer (IBT)* besitzt in 128 Ländern Filialen und stellt für 2048 spezielle Handels-Kunden in jeder Filiale ein Konto zur Verfügung. Gelder dieser Kunden werden dauernd zwischen den Filialen hin- und hertransferiert, dazu beschäftigt die Bank sogenannte *Pusher*. Pusher heben Geldbeträge von Konten in einer Filiale ab und buchen sie auf den entsprechenden Konten in irgend einer (auch in der eigenen) Filiale wieder ein. Die Beträge liegen zwischen 1000 und 100'000 Dollar und werden zufällig ausgewählt, die Wahl der beiden Filialen ist ebenfalls zufällig.

3.1 Implementation

Im folgenden arbeiten wir mit einer PThread-basierten Implementation der IBT, die Pusher werden dabei mit Threads implementiert. Die Filialen der Bank sind als Array von Strukturen implementiert, wobei pro Filiale ein Lock (`branchLock`) und ein Array von Konten (`Accounts`) definiert ist. Die Konten sind wiederum Strukturen mit dem Kontostand (`account`) und dem Lock (`acntLock`), siehe dazu auch den Source Code. Die Zugriffe auf die Gelder sind implementiert (Funktionen `withdraw()`, `deposit()`, `transfer()`), aber nicht synchronisiert.

Hinweis: es ist von Vorteil hier mit mehreren CPUs zu arbeiten. Falls sie eine VM verwenden, setzen sie die Anzahl CPUs auf das Maximum.

3.2 Aufgabenstellungen

3.2.1 Aufgabe 1

a) Wechseln sie ins Verzeichnis `banking/a1`, übersetzen sie das Programm und starten sie es mit dem Skript `./startApp`. Analysieren und erklären sie die Resultate. Notieren sie sich zudem die Laufzeiten für 1, 2 und 4 Threads.

b) Synchronisieren sie die Kontenzugriffe so, dass möglichst viele Zugriffe gleichzeitig ausgeführt werden können und die Zugriffe atomar sind. Sie dürfen nur eines der beiden Locks **`branchLock`** bzw. **`acntLock`** verwenden: welches wählen sie und wieso? Begründen sie ihre Antwort und testen sie ihre Lösung.

3.2.2 Aufgabe 2

Ihr Chef meint, dass es wohl aus Sicherheitsgründen besser wäre, sowohl die Filialen und die jeweiligen Kontenzugriffen zu "locken".

a) Wechseln sie ins Verzeichnis `banking/a2` und kopieren sie `banking.c` aus Aufgabe 1. Implementieren sie diese zusätzlichen Anforderungen. Analysieren sie die Resultate. Was stellen sie fest im Vergleich mit den Resultaten aus Aufgabe 1? Was raten sie ihrem Chef?

b) Ein Kollege meint, es wäre effizienter beim Abheben des Betrags zuerst das Konto zu locken und dann die Filiale, hingegen beim Einbuchen zuerst die die Filiale und dann das Konto. Was für eine Antwort geben sie ihrem Kollegen? Hinweis: falls sie nicht sicher sind: probieren sie es aus.

3.2.3 Aufgabe 3

Das International Banking Committe (IBC) erlässt neue Richtlinien, die unter anderem fordern, dass die Gesamtbilanz einer Bank über sämtliche Filialen zu jeder Zeit konsistent sein muss.

a) Erklären sie wieso die Implementationen aus Aufgabe 1 und 2 diese Anforderungen nicht erfüllen.

b) Ihr Entwicklungsteam kommt zum Schluss, dass den Pushern neu nur noch eine Funktion `transfer()` für die Überweisung von Beträgen zwischen den Filialen und Konten zur Verfügung gestellt werden darf.

   Welche Locks bzw. welches Lock muss verwendet werden, damit die Forderung des IBC erfüllt werden kann? Wechseln sie ins Verzeichnis `banking/a3` und ergänzen sie die Funktion `transfer()` in `banking.c` um die entsprechenden Lock-Funktionen.

   Wichtiger Hinweis:
   es darf kein neues Lock eingeführt werden und die Gesamtbilanz über sämtliche Filialen muss jederzeit konsistent sein.

c) Testen und analysieren sie das Programm und vergleichen sie die Resultate (Funktionalität, Laufzeit) mit den Lösungen aus Aufgabe 1 und 2. Notieren sie sich, was ihnen bei dieser Aufgabe wichtig erscheint.