Praktikum

Synchronisation

Kaffee− und Eiscreme−Automaten

FS 2012

M. Thaler, J. Zeman

Inhaltsverzeichnis

1 Einführung

In diesem Praktikum lernen Sie zuerst am Beispiel eines Kaffee-Automaten verschiedene grundlegende Synchronisationsprobleme kennen und mit Hilfe von Mutexes und Semaphoren lösen. Im zweiten Teil werden Sie auf Basis dieser Grundlagen systematisch ein komplexes Synchronisationsproblem bearbeiten, diesmal am Beispiel eines Eiscreme-Automaten.

1.1 Hinweis zu den Programmen

Die folgenden Beispiele benötigen mehrere Threads oder Prozesse sowie Ressourcen für die Synchronisation. Für jedes Beispiel steht das Programm **`startApp.e`** zur Verfügung, das die notwendigen Threads oder Prozesse startet und stoppt, aber auch benötigte Ressource alloziert und wieder freigibt.

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 resp. 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 resp. *erzwungen*.

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) abgearbeitet 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 Daten sind in der gemeinsamen Datenstruktur `cData` abgelegt, auf die der Kaffee-Automat und die Kunden zugreifen können. Der Automat überprüft, ob die Anzahl Münzen und die Anzahl der Kaffeewahlen gleich gross sind, falls nicht, wird eine Fehlermedung 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 resp. sein können.

b) Wir haben für Sie einen Mutex vorbereitet: die Datenstruktur `cData` enthält die Mutex-Variable `mutex`, die in `startApp.c` initialisiert wird. Schützen Sie den Zugriff auf die gemeinsamen Daten mit einem Mutex so, dass alle Threads eine konsistente Sicht der Daten haben. Die dazu benötigten Funktionen aus der Pthread Bibliothek sind:

```
pthread_mutex_lock(&(cD.mutex));      pthread_mutex_unlock(&(cD.mutex));
```

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 das Programm so ändern würden, dass die gemeinsamen Daten in lokale Variablen kopiert werden und dann nur noch auf diese Variablen zugegriffen wird, können Sie dann auf die Synchronisation mit dem Mutex verzichten?

d) Wie oft kann ein einzelner Kunde einen Kaffee beziehen, bis der nächste Kunde oder der Kaffee-Automat 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 (fast) zufällig ist.
Im folgenden soll am Beispiel einer Geldeinwurfsperre eine erzwungene Verarbeitungsreihenfolge implementiert werden: der Kunde kann nur eine Münze einwerfen, wenn die Sperre offen ist, der Automat kann die Sperre wieder öffnen (und einen Kaffee ausgeben), wenn der Kunde eine Münze eingeworfen hat.
Für die Lösung dieses Problems benötigen wir Semaphore, die, im Gegensatz zu Mutexes, auch in verschiedenen Prozessen gesetzt resp. 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) Übersetzen Sie die Prozesse im Verzeichnis `basicSequence` mit `make` und analysieren Sie die Ausgabe der Programme.

Mit `startApp.e | grep -1 starting` können Sie z.B. ausgeben, bei welchem Iterationsschritt die einzelnen Kunden gestartet werden. Was fällt auf?

b) Ergänzen Sie den `coffeeTeller`- und den `customer`-Prozess so mit zwei Semaphoren, das der Ablauf eingehalten wird, den die Geldeinwurfsperre erzwingt. Mit welchen Werten müssen die beiden Semaphore initialisiert werden?

Wir haben für Sie zwei Semaphore vorbereitet (siehe `commonDefs.h` und `startApp.c`. Die benötigten Semaphor-Funktionen aus der POSIX Bibliothek sind:

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

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

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 der einfache Reihenfolge 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 (ein Semaphor kann beliebig oft erhöht resp. geschlossen werden). Sie finden die entsprechenden Prozesse im Verzeichnis `advancedSequence`.

2.4.1 Aufgaben

Hinweis: wir verwenden hier die gleichen Prozesse wie für die `basic Sequence`.

a) Ergänzen Sie den `coffeeTeller`- und den `customer`-Prozess so mit zwei Semaphoren, das der Ablauf eingehalten wird, den die Geldeinwurfsperre erzwingt, dass nun aber mehrere Münzen eingeworfen werden müssen um einen Kaffee zu bezahlen.

   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.

b) Wieso arbeitet Ihr Programm auch mit mehreren Kunden und einer beliebigen Anzahl Münzen korrekt? Hinweis: es könnte ja sein, dass verschiedene Kunden gleichzeitig Münzen deponieren.

2.5 Optional: Faire Kunden

Angenommen Sie müssten gewährleisten, dass alle Kunden in der Warteschlange genau einmal an die Reihe kommt, bevor ein Kunden einen zweiten Kaffee kaufen kann. Kann der Fall überhaupt eintreten? Mit welchem Synchronisationsmechanismus könnte dies gelöst werden?

2.6 Zusammenfassung

Wir haben drei grundlegenden Typen von Synchronistionsproblemen kennen gelernt: Mutex, einfache Sequenz und erweiterte Sequenz.

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

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

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

3 Der Eiscreme-Automat

3.1 Spezifikation

In der Mensa wird neu ein Eiscreme-Automat betrieben. Zu Beginn ist er vom Lieferant mit 10 Eiscremes gefüllt worden.

Der Automat führt im Betrieb immer die gleichen Schritte aus: er wartet bis ein Kunde vier Euros eingeworfen hat (der Automat akzeptiert nur 1 Euro Münzen) und gibt ein Eis aus, falls er nicht leer ist, andernfalls wartet er mit der Ausgabe, bis der Lieferant neues Eis bringt.

Die Kunden führen alle immer die gleichen Schritte aus: benutzt gerade ein anderer Kunde den Automaten, wartet er, bis der Automat frei wird (Mutex). Ist der Automat frei, wirft der Kunde vier 1-Euro Münzen ein und wartet bis das Eis ausgegeben wird (das kann auch länger dauern, wenn der Automat gerade leer ist) und gibt den Automaten dann wieder frei.

Der Lieferant kommt alle drei Tage vorbei und füllt 10 Eiscremes nach, unabhängig davon wie viele Eiscremes noch vorhanden sind.

Die Daten zum Automaten, der Anzahl Kunden und zum Lieferanten sind wie folgt vordefiniert (siehe File `./iceCreamTeller/icecommonDefs.h`):

Preis einer Eiscreme	PRIZE	4
Anzahl Eiscremes pro Lieferung	QUANTITY	10
Anzahl Kunden	CUSTOMERS	4
Maximale Anzahl Nachfüllungen	REFILLS	30
Zeitdauer zwischen Lieferungen (Tage)	DELIEVERY_RATE	3

Hinweis: die Zahlenwerte sind so gewählt, dass sie zu übersichtlichen Resultaten führen und nicht unbedingt der Realität entsprechen, die Werte lassen sich aber jederzeit durch realistische Werte ersetzen.

3.2 Wie löst man dieses Synchronisationsproblem?

Schritt 1: Prozesse des Systems identifizieren Die Prozesse sind diejenigen Aktivitäten, die gleichzeitig ausgeführt werden, in diesem Sinne auch eigenständige Einheiten sind und deren zeitliches Verhalten synchronisiert werden muss.

Schritt 2: Ausführungsschritte der einzelnen Prozesse ermitteln Erstellen Sie eine Liste mit einer Spalte pro Prozess. Notieren Sie stichwortartig für jeden Prozess die wesentlichen Aktionen in der gewünschten zeitlichen Reihenfolge gemäss Spezifikation. Hier sollen Sie noch keine Synchronisationsoperationen eintragen, sondern Texte wie *warten auf Lieferant*, etc.

Schritt 3: Synchronisationsbedingungen ermitteln Eine Synchronisationsbedingung ist eine zeitliche Beziehung resp. Abhängigkeit zwischen den Aktionen verschiedener Prozesse, die für das korrekte Funktionieren erforderlich ist. In Frage kommen hier: *Mutex*, *einfache Reihenfolge* und *erweiterte Reihenfolge*.

Zeichnen Sie diese Beziehungen mit Pfeilen in die oben erstellte Liste ein.

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

Schritt 5: Das Programm mit Semaphoroperationen ergänzen Erweitern Sie nun die Prozesse aus Schritt 2 mit den Semaphoroperationen.

3.3 Aufgabenstellungen

3.3.1 Aufgabe 1

a) Beschreiben Sie den Eiscreme-Automaten mit Hilfe der 5 Schritte auf Papier, alle Schritte sind schriftlich zu dokumentieren.

b) Implementieren und testen Sie den Eiscreme-Automaten. Realisieren Sie sämtliche Synchronisationsbedingungen mit Semaphoren, d.h. mit `sem_wait(sem)` und `sem_post(sem)`.

Im File `iceCreamTeller.tgz` haben wir Rahmenprogramme für die Implementation des Automaten vorbereitet, entpacken Sie dieses File aber erst, wenn Sie Aufgabe a) gelöst haben.

c) Setzen Sie `REFILLS` auf `50` und die Lieferzeit auf einen Tag. Testen Sie das Programm erneut und analysieren Sie das Resultat.

3.3.2 Aufgabe 2

Wenn sich vor dem Automaten eine Schlange von Kunden gebildet hat, hat sich ein neuer Kunde einfach in die Schlange eingereiht: neu soll er selber entscheiden ob er warten will oder nicht. In einem zweiten Schritt soll die Kapazität des Automaten eingeschränkt werden (Anzahl Eiscremes).

Verwenden Sie für den Zugang zum Automaten die Semaphorfunktion `sem_trywait(sem)`. Für `sem` > 0 verhält sich `sem_trywait(sem)` wie `sem_wait(sem)`, der Rückgabewert in diesem Fall ist 0. Für `sem` $= 0$ blockiert `sem_trywait(sem)` jedoch nicht und der Rückgabewert ist negativ, zusätzlich wird `errno` auf `EINVAL` gesetzt.

Wichtiger Hinweis: vermeiden Sie die Verwendung von `sem_getvalue(sem, val)` soweit wie möglich: Sie werden dann nicht verleitet den Semaphorwert mit einer `if`-Abfrage für den Eintritt in einen geschützten Bereich zu verwenden (wieso ist das problematisch?).

a) Müsste ein neuer Kunde am Automat warten, wirft er zuerst eine Münze und entscheidet so, ober er warten will oder nicht. Passen Sie Ihr Programm entsprechend an. Für das Werfen der Münze steht die Prozedur `tossCoin()` in `random.c` zur Verfügung, die zufällig eine 0 oder eine 1 zurück gibt. Testen Sie Ihr Programm mit den ursprünglichen Werten in `commonDefs.h`. Sie dürfen hier keine neue Semaphore einführen.

b) Ergänzen Sie nun das Programm so, dass sich im Automaten maximal 20 Eiscremes befinden können: der Lieferant füllt nur so viele Eiscremes nach, bis diese Grenze erreicht ist. Überlegen Sie sich, ob die Implementation zu einer *race condition* oder einem *dead lock* führen kann. Bei Bedarf dürfen Sie eine neue Semaphore einführen.