Übungsblatt 5

Abgabe: Bis zum 17. Juni 2020 um 8:30 Uhr

Aufgabe 5.1: Koordination durch Semaphore (3 Punkte)

Nach einer Pressekonferenz müssen die teilnehmenden Politiker und Reporter mit einem Lift ins Erdgeschoss des Gebäudes. Der Lift kann leider nur drei Personen auf einmal aufnehmen. Es fahren immer zwei Politker zusammen mit einem Reporter (die Politiker wollen nicht von zu vielen Reportern umgeben sein, die Reporter wollen aber auf jeden Fall noch ein Interview mit Politikern). Es seien genau doppelt so viele Politiker wie Reporter anwesend.

Dieses Problem ist hier mit Zählsemaphoren gelöst worden: Politiker rufen bei ihrer Ankunft am Lift die Funktion `politicianArrives()` auf, Reporter ihr Äquivalent `reporterArrives()`. Innerhalb dieser Funktionen wird für eine passende Gruppe an Personen genau einmal die Funktion `EnterElevator` aufgerufen, die dazu führt, dass zwei Politiker und ein Reporter den Lift betreten und die Etage verlassen. Jede Wartezeit soll vermieden werden: sobald die nötige Anzahl an Politikern und Reportern am Lift angekommen ist, soll dieser betreten werden.

Die folgenden Funktionen werden verwendet:

```
void politicianArrives() {
  signal(politicianReady);
  wait(reporterReady);
}

void reporterArrives() {
  wait(politicianReady);
  wait(politicianReady);
  EnterElevator();
  signal(reporterReady);
  signal(reporterReady);
}
```

Die Semaphore `politicianReady` und `reporterReady` haben zu Beginn die Werte `0`. Funktioniert diese Lösung wie gewünscht? Falls ja: begründen Sie Ihre Antwort; falls nein: begründen Sie Ihre Antwort und geben Sie kurz an, wie die Lösung erweitert werden müsste, um das gewünschte Verhalten zu erzielen.

Aufgabe 5.2: Semaphore (2,5 + 1 + 3 + 0,5 = 7 Punkte)

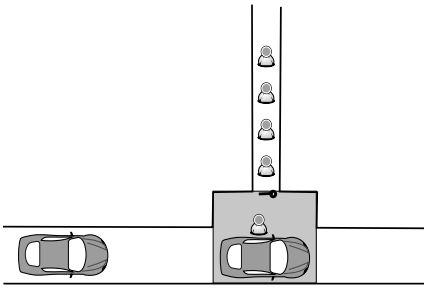
a) Der Zugriff auf geteilte Queues sollte atomar erfolgen, um zu verhindern, dass eine Queue in einen inkonsistenten Zustand kommen kann. Schreiben Sie in Pseudocode (wie z.B. in den Folien 38–57) Zugriffsfunktionen, die den exklusiven Zugriff auf eine Queue ermöglichen. Verwenden Sie für die Queue eine globale Variable `queue` vom Typ `list`, die Ihnen bereits die (nicht gegen gleichzeitige Zugriffe geschützten) Zugriffsfunktionen `void queue.add(element)` und `element queue.pop()` bereitstellt, welche zum Anhängen eines Elements vom Typ `element` an das Ende der Queue bzw. zum Entnehmen des ersten Elements der Queue dienen. Die Queue soll maximal Platz für `n` Elemente bieten.

   Erstellen Sie im Einzelnen:

   - Eine Funktion `void enqueue(element)`: Exklusives Anhängen eines Elements an die Queue. Kann aktuell kein Element angehängt werden, blockiert der Aufruf, bis ein Anhängen möglich ist.
   - Eine Funktion `element dequeue()`: Exklusive Entnahme des ersten Elements der Queue. Kann aktuell kein Element entnommen werden, blockiert der Aufruf, bis ein Anhängen möglich ist.

Bei der Erstellung der Zugriffsfunktionen können Sie auf die ungeschützten Operationen zurückgreifen. Sie können beliebig Mutexe, Zählsemaphore und Zählvariablen verwenden. Machen Sie jeweils deutlich, ob ein verwendetes Sempahor ein Mutex oder ein Zählsemaphor ist, auf welchen Wert ein Mutex/Zählsemaphor initialisiert wird und – im Fall von Zählsemaphoren – ob ein bestimmter Maximalwert gegeben sein soll (bei einem Mutex ist der Maximalwert immer 1).

b) Der Betreiber einer Achterbahn auf dem Öcher Bend hat Sie beauftragt, ein Kontrollsystem zu entwickeln, dass den Zugang von Besuchern zu den Wagen der Bahn regelt. Es steht eine Einstiegsplattform zur Verfügung, auf der genau ein Wagen Platz findet; Wagen können nur nacheinander auf die Plattform einfahren. Ein Wagen soll so lange auf der Einstiegsplattform warten, bis genau zwei Besucher eingestiegen sind. Erst wenn dies geschehen ist, fährt der Wagen ab und auf der Einstiegsplattform wird Platz für den nächsten Wagen. Wagen fahren stets leer ein; Besucher steigen auf einer anderen Plafform aus, die hier nicht betrachtet werden soll.



Es kommen jederzeit neue Wagen und Besucher an. In der aktuellen Lösung rufen Wagen und Besucher jeweils die folgenden Funktionen auf, wenn sie an der Plattform ankommen:

```
1  int eingefahreneWagen = 0;
2  int verfügbareSitze = 0;
3
4  void AnkunftWagen() {
5          while (eingefahreneWagen > 0) {noop;}
6          eingefahreneWagen = 1;
7          fahreAufPlattform();
8          öffneTüren();
9          verfügbareSitze = 2;
10         while (verfügbareSitze > 0) {noop;}
11         schließeTüren();
12         verlassePlattform();
13         eingefahreneWagen = 0;
14 }
15
16 void AnkunftBesucher() {
17         while (verfügbareSitze < 1) {noop;}
18         betreteWagen();
19         verfügbareSitze = verfügbareSitze - 1;
20 }
```

Diese Algorithmen verwenden Busy Waiting, was an sich schon unschön ist, bergen darüber hinaus aber auch ernsthafte Probleme. Welche zwei Probleme können bei Verwendung dieser Algorithmen auftreten? Geben Sie jeweils an, wie die Probleme eintreten können.

c) Sorgen Sie dafür, dass die Algorithmen `AnkunftWagen` und `AnkunftBesucher` wie gewünscht funktionieren. Sie können nach Belieben Mutexe, Zählsemaphore und Zählvariablen verwenden. Busy waiting ist allerdings nicht mehr erlaubt. Machen Sie jeweils deutlich, ob ein verwendetes Sempahor ein Mutex oder ein Zählsemaphor ist, auf welchen Wert es initialisiert wird und – im Fall von Zählsemaphoren – ob ein bestimmter Maximalwert gegeben sein soll (bei einem Mutex ist der Maximalwert immer 1).

Sie brauchen nicht die gleichen Variablen wie oben zu verwenden, sondern können Sie durch Ihre eigenen Variablen und Semaphore ersetzen. Die Funktionen `fahreAufPlattform()`, `öffneTüren()`, `schließeTüren()`, `verlassePlattform()` und `betreteWagen()` sollen allerdings weiter verwendet werden.

d) Nehmen Sie an, dass neben der normalen Warteschlange noch eine VIP-Warteschlange existiert; immer dann, wenn ein VIP-Besucher ankommt, ruft er eine Funktion `AnkunftVIP` auf. Solche Besucher sollen bevorzugt einsteigen können. Lässt sich dieses Problem auch mit Hilfe von Semaphoren lösen? Wenn ja, wie?

Hinweis: Sie brauchen keinen Pseudocode abzugeben, sondern nur kurz zu skizzieren, ob eine Lösung möglich ist und wie sie ggfs. aussehen könnte.

Aufgabe 5.3: Umgang mit Semaphoren in C (7 + 3 = 10 Punkte)

Als Anhang zum Übungsblatt finden Sie das Programm `restaurant_geruest.c`; es versucht ein Restaurant zu simulieren. Das Restaurant hat folgende Eigenschaften:

- Es dürfen aufgrund der aktuellen Schutzmaßnahmen nur 8 Gäste in das Restaurant.
- Die Schlange vor dem Restaurant darf höchstens 10 Gäste lang sein, da sonst der Gehweg versperrt werden würde.
- Ein Gast wartet nicht länger als 2 Sekunden in der Schlage, um in das Restaurant zu kommen.
- Ein Gast bleibt zwischen 3 und 8 Sekunden im Restaurant.

Sie werden feststellen, dass so gut wie alles bereits implementiert ist. Die einzelnen Gäste werden über Prozesse simuliert. Sie werden vom Hauptprozess erzeugt und versuchen sich in die Schlange einzureihen. Noch nicht implementiert ist zum einen, dass sie höchstens 2 Sekunden auf Einlass warten, und zum anderen die Beschränkung der Anzahl der sich gleichzeitig im Restaurant befindenden Gäste.

a) Es treten eine ganze Reihe an Synchronisationsproblemen auf, da das Restaurant durch einen Shared-Memory-Bereich von allen Prozesses verwendet wird. Sie finden im Programm bereits eine Semaphore, die sich in genau diesem gemeinsamen Speicherbereich befindet; sie soll genutzt werden, um die Anzahl der Gäste im Restaurant zu modellieren und die Beschränkung der Gästezahl sicherzustellen. Sie benötigen allerdings eine weitere Semaphore, um den Zugriff auf die synchronisationskritischen Variablen im gemeinsamen Speicher zu schützen.

Verwenden Sie unbenannte POSIX-Semaphoren (dies bedeutet, dass sie mit `-pthread` linken müssen); lesen sie sich als Hilfestellung die man pages zu `sem_init`, `sem_wait`, `sem_timedwait`, `sem_post` sowie `sem_destroy` durch. Es wird auch empfohlen, die man page zu `sem_overview` zu lesen. Je nach System sowie Art und Weise, wie `sem_timedwait` verwendet wird, kann es nötig sein, noch mit `-lrt` zu linken. Achten Sie darauf, dass sie bei Terminierung freien Speicher / Shared Memory wieder freigeben.

b) Um die Sicherheit der Gäste zu erhöhen, wurde beschlossen, nur Gäste in das Restaurant hereinzulassen, die eine Maske tragen. Ein Kellner wird beauftragt, alle 3-6 Sekunden 3 bis 5 weitere neue Einwegmasken am Eingang bereitzulegen, die sich die Gäste nehmen, nachdem ein Platz im Restaurant freigeworden ist. Wenn ein Platz im Restaurant frei ist, allerdings noch keine Maske bereitliegt, sind die Gäste bereit beliebig lange auf das Eintreffen neuer Masken zu warten. Es wird nicht davon ausgegangen, dass Gäste ihre eigenen Masken mitbringen.

Implementieren Sie den Kellner als Kindprozess des Root-Prozesses. Modellieren Sie den Sachverhalt mittels Semaphoren und achten Sie darauf, dass sich der Prozess bei einem SIGINT-Signal selber terminiert. Der Root-Prozess wartet allerdings auf die Beendigung aller seiner Kindprozesse.

Sie finden ein paar Hinweise in Form von Kommentaren im Quellcode; es ist ratsam, diese zu befolgen. Achten Sie auf die Effizienz Ihres Programms, d.h. verwenden Sie nur dann Synchronisation, wenn es wirklich von Nöten ist. Es ist nicht zulässig, Sperren so zu verwenden, dass die Prozesse nur noch nacheinander ablaufen können.

Wie immer muss ihr Programm mit `-Wall` ohne Warning compilieren.