Übungsblatt 5

Abgabe: 16. Juni 2016

Aufgabe 5.1: Game of (critical) ~~Thr~~Zones (1 + 2 + 1 + 2 = 6 Punkte)

In dieser Aufgabe wollen wir Dachen bekämpfen und gleichzeitig etwas zum Thema Synchronisation lernen. Dazu spielen wir das Spiel *The Deadlock Empire* auf `https://deadlockempire.github.io/`.

Spielen Sie alle Level des Spiels bis auf "High-Level Synchronization Primitives" durch[1], um Experte in Sachen Synchronisation zu werden.

Konkret für diese Aufgabe: geben Sie Ihre Lösung in Form eines Schedules für die folgenden Level an:

a) Boolean Flags Are Enough For Everyone

b) Simple Counter

c) Deadlock

d) Semaphores

Geben Sie Ihre Schedules in der Form an, wie es das folgende Beispiel zum Level **Tutorial: Non-Atomic Instructions** zeigt. Ein Latex-Template für die Tabellen finden sie im gemeinsamen Bereich in L2P.

Level:

Lösung:

Thread 0	Thread 1	Variable (a = 0)
temp = a + 1		
	temp = a + 1	
	a = temp	a = 1
a = temp		a = 1
if (a == 1)		
critical_section()		
	if (a == 1)	
	critical_section()	

Hinweis: Weitere Tipps sowie Unterschiede zu den in der Vorlesung vorgestellten Mechanismen.

Das Spiel benutzt Mechanismen der Programmiersprache C#. Diese lassen sich aber sehr einfach in die aus der Vorlesung bekannten C-Mechanismen übersetzten.

[1] Sie können diese Level natürlich auch spielen. Wir benutzten die dort verwendeten Primitive jedoch nicht in der Vorlesung.

So sind z.B. Typen wie `System.Boolean` gleichbedeutend zu einem `bool` in C usw. Das Konzept des `Monitors` (siehe Lock Level) existiert auf diese Art und Weise nicht in C. Der Monitor erlaubt es, ein beliebiges Objekt zu sperren oder zu entsperren; dann darf nur ein Thread gleichzeitig seine Methoden nutzen. Dies kann man in C nachbilden, indem man einen Mutex pro Objekt verwendet und in jeder Methode mit diesem Mutex den Zugriff prüft. Im Spiel werden die Monitore nur als Mutex genutzt, daher ist `Monitor.Enter(xyz)` vergleichbar mit `pthread_mutex_lock(xyz)` (bzw. `wait(xyz)` als generellen Pseudocode für einen Mutex `xyz`) und `Monitor.Exit(xyz)` mit `pthread_mutex_unlock(xyz)` (bzw. `signal(xyz)`) usw.

Analog lassen sich die Semaphorzugriffe auf C bzw. den in der Vorlesung verwendeten Pseudocode abbilden. Für eine Semaphore `ss` (die in C bzw. Pseudocode z.B. durch `sem_init(ss...)` (bzw. `init(ss,..)`) angelegt werden kann) ist ein `ss.Wait()` gleichbedeutend zu einem `sem_wait(ss)` (Pseudocode: `wait(ss)`) und `ss.Signal()` entspricht `sem_post(ss)` (Pseudocode: `signal(ss)`).

Aufgabe 5.2: 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 5.3: Semaphore (3 + 1 + 3 + 3 + 1 = 11 Punkte)

a) Der Zugriff auf Queues in einem Betriebssystem sollte atomar erfolgen, um zu verhindern, dass eine Queue in einen inkonsistenten Zustand kommen kann. Schreiben Sie in Pseudocode 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

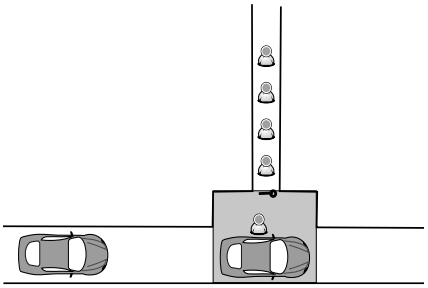
Anhängen eines Elements vom Typ `element` an das Ende der Queue bzw. zum Entnehmen des ersten Elements der Queue dienen. Die Queue soll maximal Platz für `n` Elemente bieten.

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 Ihrer Funktionen können Sie auf die ungeschützten Queue-Zugriffsfunktionen 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) Implementieren Sie ihre Lösung aus c) in C. Nutzen Sie dazu die Datei `bend.c`, die mit dem Übungsblatt in L2P verfügbar ist. In dieser Datei finden Sie auch die Funktionen `AnkunftBesucher()` und `AnkunftWagen()`. Der Hauptprozess erzeugt nach und nach Kindprozesse; diese führen eine der beiden Funktionen aus. Synchronisieren Sie den Code, so dass das System korrekt funktioniert – als Beispiel haben wir einen Besucherzähler eingebaut, der bereits über einen Mutex (bzw. genauer: eine Zählsemaphore mit Startwert 1, die wie ein Mutex genutzt wird) korrekt synchronisiert ist. Sie können benannte oder unbenannte Semaphoren nutzen. (Achtung: letztere sind nicht auf allen Plattformen verfügbar!) Um Ihnen die Wahl zu lassen, haben wir schon ein compile-time `#define` eingebaut, mit dem sie die vorhandene Implementierung zwischen benannten und unbenannten Semaphoren hin und her schalten können.

   Ihre Lösung muss mit `gcc -std=c11 -Wall -Wextra -pedantic -pthread bend.c` compilieren!

   Um zwischen benannten und unbenannten Semaphoren zu schalten, können sie zusätzlich noch mit `-DUSE_NAMED_SEMAPHORES` compilieren – dann nutzt der bereits implementierte Teil benannte Semaphoren.

   Geben Sie bei Ihrer Abgabe an, ob benannte oder unbenannte Semaphoren genutzt werden. Sie können z.B. das `#define` fest im Programmkopf verankern.

e) 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 oder C-Code abzugeben, sondern nur kurz zu skizzieren, ob eine Lösung möglich ist und wie sie ggfs. aussehen könnte.