Vorwort: Für alle zu implementierenden Aufgaben werden entsprechende Demonstrationsmöglichkeiten erwartet, also Unittests, eine Main-Methode, Visualisierung oder sprechende Ausgabe auf der Konsole.

Alle Mechanismen zur Synchronisation müssen selbst implementiert werden! Es dürfen lediglich Synchronisationprimitive (hier: Mutexe, Semaphore, Monitore) verwendet werden!

1 Erzeuger-Verbraucher-Problem

1.1 Problematik

Das *Erzeuger-Verbraucher-Problem* (engl. *Producer Consumer Problem*, folgend *EVP*) ist eine Problemstellung aus der Prozesssynchronisierung. Dabei gibt es einen zentralen Puffer, in welchen Daten (hier: *Workitems*) hinterlegt werden. Die Form des Puffers ist zunächst irrelevant, kann aber zum Beispiel als Stack, Queue oder Ringpuffer organisiert sein. Im Rahmen dieser Übung wird von einer Queue mit fester Größe ausgegangen.
Auf den Puffer kann schreibend (*Producer*) und lesend (*Consumer*) zugegriffen werden. Dabei können zwei Probleme auftreten:

1. Die lesenden/ schreibenden Zugriffe müssen untereinander synchronisiert sein, um zu verhindern, dass Workitems mehrfach gelesen werden. Außerdem muss bei Puffern mit beschränkter Größe sichergestellt werden, dass nicht zwei schreibende Zugriffe versuchen, den letzten Platz zu belegen und damit einen Überlauf hervorrufen.

2. Lesende und schreibende Zugriffe müssen zueinander synchron sein: ist der Puffer leer, kann erst wieder daraus gelesen werden, wenn ein neues Element hineingeschrieben wurde. Umgekehrt dürfen nur dann neue Elemente eingefügt werden, wenn noch Platz im Puffer vorhanden ist.

Die Stärke dieses Aufbaus besteht in der Zentralisierung der Synchronisierung (nämlich dem einen Knotenpunkt, dem Puffer) und der Möglichkeit, beliebig viele Producer und Consumer hinzuzuschalten, falls die Situation es erfordert.

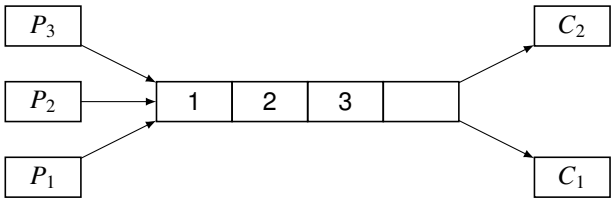

Figure 1: Darstellung eines EVP mit drei Producern und zwei Consumern. Als Puffer fungiert eine Queue mit vier Plätzen, von denen bereits drei Plätze belegt sind.

1.2 Beispiel

Ein Onlinedienst bietet die Konvertierung von Dateien in andere Formate an. Benutzer können die Dateien über ihren Browser hochladen. Sobald sie konvertiert sind, werden sie dem Benutzer per Mail zugeschickt.
Dabei können viele Threads gleichzeitig die Dateien der Benutzer entgegennehmen und in einen Puffer schreiben. Im Hintergrund kann ein einziger Thread den Puffer Stück für Stück abarbeiten, da es sich um eine nicht zeitkritische Aufgabe handelt.
Sollten von den Benutzern viele Beschwerden eingehen, dass die Konvertierung zu lange dauert, kann man weitere Threads hinzuschalten, um das Abarbeiten zu beschleunigen ohne dabei die Architektur der Software anpassen zu müssen.

1.3 Ablauf

Der Puffer besitzt einen Mechanismus, um exklusiven Zugang zu sich zu gewähren (Lock, Semaphor, Monitor, ...). Producer und Consumer laufen jeweils in einem eigenen Thread.

1.3.1 Schreiben

Möchte ein Teilnehmer in den Puffer schreiben, lässt er sich exklusiven Zugang erteilen. Er prüft, ob noch genug Platz im Puffer ist. Falls nicht, legt er sich schlafen. Ansonsten legt er sein Workitem in den nächsten freien Platz. In beiden Fällen gibt es den exklusiven Zugang wieder auf.

1.3.2 Lesen

Der lesender Zugriff funktioniert analog: mit exklusivem Zugang zum Puffer prüft der Consumer, ob mindestens ein Workitem im Puffer enthalten ist. Falls nicht, legt er sich schlafen. Sonst liest und entfernt er das Workitem aus dem Puffer. In beiden Fällen gibt es den exklusiven Zugang wieder auf.

1.4 Aufgaben

Implementiert ein Parkhaus als EVP!

1. Implementiert eine Klasse `Car`. Diese dient lediglich der Anschaulichkeit des Beispiels und kann beliebig gestaltet sein. (2p)
2. Implementiert einen generischen `Buffer`. Dieser soll mindestens die folgenden Operationen unterstützen:
   - `push(e):` fügt *e* in den Puffer ein. Ein Element einzufügen, obwohl der Puffer voll ist, löst einen Fehler aus! (1p)
   - `pop():` nimmt das nächste Element aus dem Puffer heraus und gibt es zurück. Ein Element zu entnehmen, obwohl der Puffer leer ist, löst einen Fehler aus! (1p)

- `full()`: prüft, ob der Puffer voll ist. (1p)
- `empty()`: prüft, ob der Puffer leer ist. (1p)

Jeder Puffer hat außerdem eine feste Größe `size` und ein Mutex, mithilfe dessen man vorübergehend exklusiven Zugriff auf den Puffer erhält (4p).

3. Implementiert einen Consumer, der Autos aus einem `Buffer<Car>` nimmt. Dazu soll er in zufälligen Abständen prüfen, ob noch ein Auto im Parkhaus steht. Falls ja, "parkt" er dieses aus, indem er es aus dem Buffer entfernt. Dazu muss während des Prüfens und während des Entnehmens exklusiver Zugriff auf den Puffer erlangt werden! Ist das Parkhaus leer, schläft der Consumer stattdessen. War der Puffer vor der Entnahme komplett voll, weckt er alle schlafenden Producer auf. (6p)

4. Implementiert einen Producer, der Autos in denselben `Buffer<Car>` schreibt. Dieser prüft in zufälligen Abständen, ob noch Platz im Parkhaus ist. Falls ja "parkt" er dort ein Auto ein. Sollte der Puffer vor dem Einparken leer gewesen sein, weckt er alle schlafenden Consumer auf. (6p)

5. Implementiert eine Main-Methode die die Anzahl an Producern und Consumern in der Form zweier Programmparametern entgegennimmt. Erstellt für jeden Producer und jeden Consumer einen eigenen Thread und lasst diese dann endlos arbeiten. Implementiert eine geeignete (Konsolen-) Ausgabe um darzustellen, dass sowohl die Producer als auch die Consumer die Regeln des EVPs in 1.3 einhalten. (4p)

6. Erstellt eine geeignete Dokumentation des Programms in UML und als Petry-Netz (7p)