POSIX/C Multithreading: Semaphoren

Aufgabe 1: Erzeuger/Verbraucher mit Semaphoren

Lösen Sie das Erzeuger/Verbraucher Problem aus Übung04/Aufgabe 4 jetzt mit Semaphoren. Condition Variablen und Mutex werden nicht mehr benötigt.

Die Synchronisation der Threads erfolgt nun so:

- Ein Semaphor `semFrei` führt mit seinem Zähler Buch über die noch freien Pufferplätze.
- Ein weiterer Semaphor `semBelegt` führt Buch über die bereits belegten Plätze.
- Die Erzeuger- und Verbraucherthreads aktualisieren die Semaphorzähler bei jedem Schreib- oder Lesevorgang mit entsprechenden P- und V-Operationen. Hierdurch wird erreicht, dass der Erzeuger blockiert, solange der Puffer voll ist und dass der Verbraucher blockiert, solange der Puffer leer ist.
- Der Puffer stellt natürlich auch einen kritischen Bereich dar. Der notwendige Mutex wird hier durch einen binären Semaphor `semBin` ersetzt.

Hinweise:

- `#include <semaphore.h>` ist erforderlich.
- Den Zähler der Semaphoren richtig initialisieren: bei einer anfangs leeren Queue sind alle Plätze frei und keine Plätze belegt.
  Ein mit eins initialisierter binärer Semaphor (Mutex) ist frei.
- Die Puffer-voll/leer-Tests (Schleifen) in `enqueue`/`dequeue` werden bei dieser Lösung für die Synchronisation nicht mehr benötigt.
  Allein für die Umschaltung der Schreib/Leseverzögerungszeiten verbleibt jeweils ein solcher Test. Diesen Test und die Umschaltung führen Sie aber erst aus, nachdem die Operationen am Puffer durchgeführt wurden - erst dann haben die Indexverweise einen neuen Wert bekommen.

Aufgabe 2: Drei Erzeuger und drei Verbraucher

Die Semaphor-Queue aus Aufgabe 1 sollte nun ohne Änderung auch mehrere Erzeuger und Verbraucher synchronisieren können.

Erstellen Sie weitere threads, sodass drei Erzeuger und drei Verbraucher auf die Queue zugreifen:

- ein Erzeuger schreibt fortlaufend die Großbuchstaben A bis Z
- ein Erzeuger schreibt fortlaufend die Kleinbuchstaben a bis z
- ein Erzeuger schreibt fortlaufend die Ziffern 0 bis 9
- drei Verbraucher, die fortlaufend je ein Zeichen aus der Queue konsumieren

Stellen Sie die Frequenzen der threads unterschiedlich ein. In der Queue sollten sich nun Ziffern und große und kleine Buchstaben befinden. Allerdings müssen die Sequenzen der drei Zeichenfolgen erhalten bleiben, z.B. so:

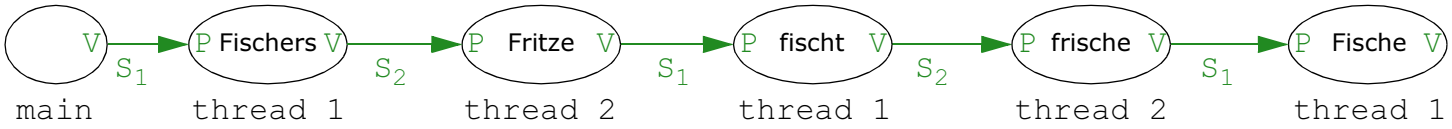
```
U0kV1W2lX3------G6d7HI8e9J0Kf1L2Mg3N4Oh5P6Qi7R8Sj9T
```

Aufgabe 3: Reihenfolgensynchronisation

Es soll eine Reihenfolgensynchronisation mit zwei threads wie folgt realisiert werden:
Die Wörter des Satzes Fischers Fritze fischt frische Fische sollen abwechselnd von den beiden threads ausgegeben werden.

Der main-thread initialisiert die Semaphoren S_1 und S_2 und startet die beiden anderen threads. Diese blockieren jedoch sofort an den Semaphoren. Mit einer V-Operation gibt main dann das Startsignal.

Präzedenzgraph:



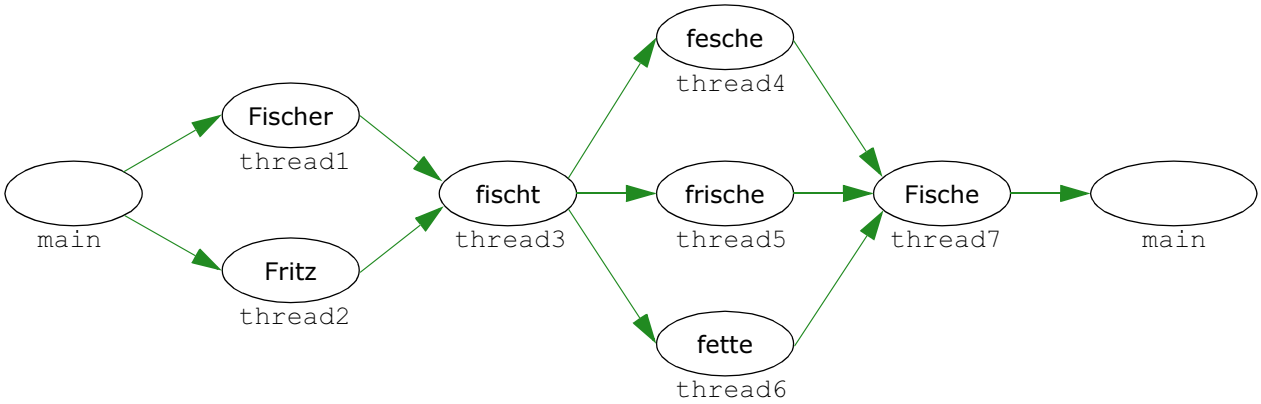
Aufgabe 4: Noch mehr Fische

Nun sollen sieben threads jeweils ein Wort derart ausgeben, dass z.B. folgender Satz entsteht:

Fritz Fischer fischt frische fesche fette Fische

Die Reihenfolge der Wörter in den unterstrichenen Bereichen ist beliebig, die threads für die Ausgabe von *Fritz* und *Fischer* bzw. *frische*, *fesche* und *fette* könnten auch in anderer Folge ausgeführt werden.

Präzedenzgraph:



Definieren Sie die Anzahl der notwendigen Semaphoren und notieren Sie im Graphen die P- und V-Operationen. Die threads `1`, `2` bzw. `4`, `5`, `6` sollen gleichzeitig frei werden, d.h. sie könnten parallel laufen.

Der main-thread erzeugt und initialisiert alle Semaphoren, startet die restlichen threads und gibt dann `thread1` und `thread2` frei.
Zum Schluß wartet `main` auf `thread7`, ebenfalls per Semaphor. Die bisher hier verwendeten `join`-Operationen sind nicht erforderlich.

Fügen Sie zufällige Wartezeiten ein, sodass immer eine andere Wortfolge entsteht.

Die Ausgaben der parallel ausführbaren threads (`1`,`2` und `4`,`5`,`6`) bilden einen kritischen Bereich, die Buchstaben könnten hier durcheinander geraten. Korrekterweise muss hier zusätzlich ein Mutex bzw. ein binärer Semaphor verwendet werden.