Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

Informatik

Algorithmen & Datenstrukturen (AD)

Übung: Weiterführende Konzepte (N3)

Themen:	Semaphore, Java Thread Pool Executors, Future- und Callable-Schnittstelle, Callables und Executors, Atomic-Variablen, Thread-sichere Container, Blocking Queues
Zeitbedarf:	ca. 240min.

Roger Diehl, Version 1.0 (FS 2017)

1 Bounded Buffer (ca. 30')

1.1 Lernziele

- Sie können das Konzept der **„guarded blocks“** mit weiterführenden Konzepten umsetzen.

1.2 Grundlagen

Diese Aufgabe setzt die Aufgabe 3 aus der Übungsserie „Thread Steuerung (N2) voraus. Zudem benötigen Sie den AD Input N31 (Blocking Queues) und AD Input N22 (Java Thread Pool Executors).

1.3 Aufgabe

Sie haben in der Aufgabe 3 aus „Thread Steuerung (N2) eine Klasse `BoundedBuffer` erstellt, die folgende Methoden beinhaltete.

- `put` – legt ein Element in den Buffer
- `put` mit Timeout – versucht ein Element in den Buffer abzulegen
- `get` – entnimmt ein Element dem Buffer
- `get` mit Timeout – versucht ein Element dem Buffer zu entnehmen
- `front` – liefert das erste Element im Buffer zurück
- `back` – liefert das letzte Element im Buffer zurück
- `push` – fügt ein neues Element nach dem LIFO Prinzip in den Buffer ein
- `pop` – entfernt ein Element nach dem LIFO Prinzip aus dem Buffer
- `empty` – dient der Abfrage ob der Buffer leer ist
- `full` – dient der Abfrage ob der Buffer voll ist
- `size` – gibt die Anzahl Elemente im Buffer zurück

Wählen Sie eine Blocking Queue aus dem Package `java.util.concurrent`, welche die obigen Methoden enthält (möglicherweise lauten einige Methodennamen anders). Ersetzen Sie die selbst erstellte Klasse `BoundedBuffer` mit der gewählten Blocking Queue und machen Sie die gleichen Tests, wie in N2. Für die Thread-Erzeugung setzen Sie einen Thread Pool Executor ein.

1.4 Reflektion

Reflektieren Sie die Aufgabe (hilft auch bei einer eventuellen Präsentation) und beantworten Sie sich die folgenden Fragen:

- Worin liegt der grösste Unterschied, zwischen Ihrem Bounded Buffer und der gewählten Blocking Queue? Haben Sie Unterschiede in den Tests der beiden Aufgaben festgestellt?
- Wie sind Sie mit den anders lautenden Methodennamen umgegangen?

2 Parkhaus (ca. 90')

2.1 Lernziele

- Sie können Threads mit Hilfe eines Synchronisationsmechanismus beim Zugriff auf gemeinsame Ressourcen steuern.
- Sie können Thread Pool Executoren, sowie Future- und Callable-Schnittstellen anwenden.

2.2 Grundlagen

Diese Aufgabe basiert auf vielen Themen. Beispielsweise dem AD Input N31 (Callables und Executors , Java Thread Pool Executors und Blocking Queues), aber auch N21 – Thread Steuerung, Input N22 – Threadpools.

Stellen Sie sich die fiktive Stadt „Pannobile“ vor. Idyllisch an einem ruhigen See gelegen. In der Ferne verschneite Berggipfel. Inmitten von Grünflächen attraktive Unterhaltungs- und Einkaufsmöglichkeiten. Die Strassen gesäumt mit japanischem Ahorn…ähm ich schweife ab.

Jedes Wochenende strömen zehntausende von Besuchern und potentiellen Kunden nach „Pannobile“. Deshalb will die Stadtplanung drei Parkhäuser bauen, eines direkt an der Autobahn und zwei näher zum Zentrum. Die Frage ist: Wie gross sollen die Parkhäuser sein? Sie sollen das per Simulation ermitteln.

2.3 Aufgabe

a) Wir benötigen zuerst die Klasse `CarPark`. Diese soll die Zufahrten und Ausfahrten eines Parkhauses kontrollieren. Das heisst, wenn keine Parkplätze mehr vorhanden sind, werden die Autos an der Zufahrt gehindert. Folgende Eigenschaften besitzt die Klasse `CarPark`:
   - Kapazität des Parkhauses
   - Anzahl freie Parkplätze
   - Kontrollierte Zufahrt - Bei der Zufahrt gibt es Wendemöglichkeiten, d.h. ein Auto kann das Warten auf einen Parkplatz abbrechen.
   - Ausfahrt
   - Optional: Mittlere Belegung über eine bestimmbare Zeitdauer

b) Wir benötigen zudem die Klasse `Car`. Ein Auto, bzw. der Autofahrer, kann zum Parkhaus hinfahren, ins Parkhaus einfahren, dort Parken und schliesslich wieder ausfahren. Dabei sind folgende Randbedingungen einzuhalten:
   - Die Parkdauer ist zufällig. Es soll aber eine Mindestparkdauer geben.
   - Die Dauer der Zufahrt zum jeweiligen Parkhaus soll immer gleich lang sein. Optional kann die Dauer leicht variieren.

   Für die Simulation sind folgende Arten von Autofahrern wichtig:

   - Autofahrer, der das erste direkt an der Autobahn gelegene Parkhaus nimmt. Egal wie lange er warten muss, wenn es voll ist.
   - Autofahrer, der das erste, dann das zweite, dann das dritte Parkhaus nimmt, wenn jeweils ein Parkhaus voll ist. Wenn es keine freien Parkplätze hat, fährt er wieder weg.
   - Autofahrer, der das erste, dann das zweite, dann das dritte Parkhaus nimmt, wenn jeweils ein Parkhaus voll ist. Allerdings wartet er eine bestimmte Zeit bei einem vollen Parkhaus, bevor er zum nächsten fährt. Wenn es keine freien Parkplätze hat, fährt er wieder weg.
   - Autofahrer, der das Parkhaus aussucht, das am meisten freie Parkplätze hat. Wenn es keine freien Parkplätze hat, fährt er wieder weg.
   - Alle Autofahrer haben gemein, dass wenn sie geparkt haben wieder wegfahren.

c) Wir benötigen nun eine Simulation für die Belegung der Parkhäuser. Dabei sind folgende Randbedingungen einzuhalten:
   - Die Simulation hat eine festgelegt Dauer.
   - Nach Ablauf der Simulation beendet die Applikation.
   - Die Simulation legt die Zeiteinheit fest, z.B. 1 Stunde sind 600mSec.
   - Die Kapazität des Parkhäuser wir zu Beginn der Simulation festgelegt.
   - Die Autos fahren eigenständig, d.h. nebenläufig, umher.
   - Es können tausende von Autos unterwegs sein.
   - Am Schluss wird eine Statistik erstellt, die auflistet:
     - Wie viele Autos nach „Pannobile“ gefahren sind.
     - Wie viele Autos parken konnten.
     - Wie viele Autos wieder weggefahren sind ohne zu parken.
     - Optional: Die mittlere Auslastung der Parkhäuser.

Für alle Teilaufgaben gilt: Nutzen Sie die Möglichkeiten, die Ihnen das Paket `java.util` und `java.util.concurrent` bietet.

An dieser Stelle ist Ihre Aufgabe erledigt und die Aufgabe der Stadtplanung beginnt. Sie muss nun die Grösse der Parkhäuser herausfinden, so dass bei einer potentiellen Anzahl Besucher möglichst viele parken (und ihr Geld in „Pannobile“) können. Mit der Randbedingung, dass die Parkhäuser möglichst klein sind.

Wenn Sie möchten, können Sie der Stadtplanung auf der Suche helfen.

2.4 Reflektion

Reflektieren Sie die Aufgabe (hilft auch bei einer eventuellen Präsentation) und beantworten Sie sich die folgenden Fragen:

- Wie viele Autos konnten Sie maximal nebenläufig „fahren“ lassen?
- Wie effizient arbeitet Ihre Simulation?
- Wie fair sind Ihre Parkhäuser?
- Falls Sie die Stadtplanung unterstützt haben: Was raten Sie den Stadtplanern von „Pannobile“, welche Kapazitäten sollen die Parkhäuser haben?
  - Wie viele Varianten haben Sie rechnen lassen?

3 Speed Count (ca. 60')

3.1 Lernziele

- Sie können Thread-sichere Zugriffe auf einzelnen Variablen von elementaren Datentypen nachvollziehen, modifizieren und erstellen.
- Sie können einen Performance-Vergleich mit Thread-sicheren Zugriffen auf Variablen machen.
- Sie können Thread Pool Executoren, sowie Future- und Callable-Schnittstellen anwenden.

3.2 Grundlagen

In dieser Aufgabe betreiben wir etwas Speed Count – nein nicht das illegale Zählen beim Black-Jack.

Wir wollen den Performance-Gewinn bei Atomic-Variablen „sehen“. Deshalb basiert diese Aufgabe auf dem AD Input N31 (Atomic-Variablen und Future- und Callable-Schnittstelle) und dem AD Input N22 (Java Thread Pool Executors).

3.3 Aufgabe

Im AD Input N31 sind zwei Thread-sichere Zähler vorgestellt worden. Diese beiden Zähler sollen Sie einander gegenüberstellen und den Performance-Unterschied messen. Um den Test nachvollziehbar (auch vom Code) zu gestalten benötigen Sie folgende Klassen, bzw. Algorithmen.

- Erstellen Sie eine gemeinsame Schnittstelle `Count` für die beiden Thread-sichere Zähler Klassen. Die Zähler Klassen sollen die Schnittstelle `Count` implementieren.
- Weiter brauchen Sie einen Task, der ein Zähler Objekt „bearbeiten“, d.h. Rauf- und Runterzählen kann.
- Zudem benötigen Sie eine Klasse, welche die Thread-sicheren Zähler parallel testen kann.

Der Zähler Test muss folgende Eigenschaften aufweisen.

- Der Test muss nachvollziehbar und vor allem reproduzierbar sein.
- Um die Performance auszuweisen muss der Test die Zeit eines Testdurchlaufes möglichst genau messen. Bedenken Sie, dass Ihre JVM eine Preemptive Multitasking Maschine ist. Ein einzelner Testlauf wird also immer eine etwas andere Zeit ergeben.
- Der Test muss auch die Korrektheit der Zähler überprüfen, soweit dies möglich ist.
- Der Test soll nach getaner Arbeit sauber beenden. Ein `System.exit` ist nicht erlaubt.

Nutzen Sie die Möglichkeiten, die Ihnen das Paket `java.util` und `java.util.concurrent` bietet.

Optional: Fällt Ihnen noch eine eigene Implementation für einen Thread-sicheren Zähler ein? Wenn ja, testen Sie diesen selbstverständlich mit.

3.4 Reflektion

Reflektieren Sie die Aufgabe (hilft auch bei einer eventuellen Präsentation) und beantworten Sie sich die folgenden Fragen:

- Was können Sie über die Performance der beiden Thread-sicheren Zähler aussagen?
- Was stellen Sie bei den Test Resultaten fest?
- Wie erklären Sie sich die Test Resultate?
- Welche Genauigkeit erreicht Ihr Test?

4 Suche nach grossen Primzahlen (ca. 60')

4.1 Lernziele

- Sie können `Callable`, `Future` und `ExecutorService`, sowie Thread Pool Executoren anwenden.
- Sie frischen den Umgang mit Stream und File-Handling auf.

4.2 Grundlagen

Diese Aufgabe basiert auf dem AD Input N31 (Callables und Executors) und dem OOP Input O13_IP_Datenströme.

Die feste Länge der primitiven Datentypen `int` und `long` für Ganzzahlwerte reicht für diverse numerische Berechnungen nicht aus. Besonders wünschenswert sind beliebig grosse Zahlen in der Kryptografie. Für solche Anwendungen gibt es im `math`-Paket die Klasse: `BigInteger` für Ganzzahlen.

Der Konstruktor `BigInteger(int numbits, Random rnd)` liefert eine Zufallszahl aus dem Wertebereich 0 bis 2^`numBits`–1. Alle Werte sind gleich wahrscheinlich.

Die Methode `boolean isProbablePrime(int certainty)` gibt an, ob das `BigInteger`-Objekt mit der Wahrscheinlichkeit `certainty` eine Primzahl ist. Die Methode ist blockierend. Je grösser `certainty` ist, desto mehr Zeit nimmt die Prüfung in Anspruch.

Hinter dieser Methode steckt der Miller-Rabin-Test, eine Weiterentwicklung des Fermat-Tests. Der Miller-Rabin-Test ist ein probabilistisches Verfahren. Es ist effizient, aber es liefert die richtige Antwort nur in 99,9...9 % aller Fälle. Die Anzahl der Neunen hängt von der Anzahl (`certainty`) der Iterationen des Miller-Rabin-Tests ab.

`https://de.wikipedia.org/wiki/Miller-Rabin-Test`

4.3 Aufgabe

Produzieren Sie `BigInteger`-Objekte, die mindesten 1024 Binärstellen besitzen[1]. Die `BigInteger`-Objekte sollen auf Prim getestet werden und zwar mit der höchst möglichen Genauigkeit. Die `BigInteger`-Objekte, die Primzahlen sind, sollen von einem Print-Task entgegen genommen werden, der sie in ein File speichert.

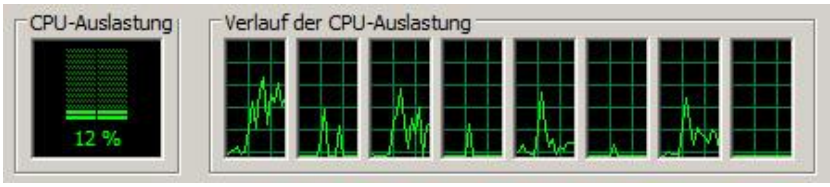
Ziel ist es innert möglichst kurzer Zeit 100 grosse Primzahlen zu finden.

Sequentiell kann man dies so machen:

```
int n = 0;
while (n < 100) {
    BigInteger bi = new BigInteger(1024, new Random());
    if (bi.isProbablePrime(Integer.MAX_VALUE)) {
        n++;
        LOG.info(n + ". " + bi.toString().substring(0, 20) + "...");
    }
}
```

[1] Primzahlen stellen die Basis für verschiedene Verschlüsselungsverfahren dar. Das Prinzip des RSA-Verfahrens zum Beispiel besteht darin, dass man leicht aus zwei geheim zuhaltenden grossen Primzahlen `p` und `q` das öffentlich bekanntzugebende Produkt `n=p*q` berechnen, aber nur sehr schwer die öffentlich bekannte Zahl `n` wieder in ihre beiden geheimen Primfaktoren `p` und `q` zerlegen kann. Sein Reiz besteht darin, dass der erforderliche Aufwand zur "Faktorisierung" bisher nicht bekannt ist. Man nimmt an, dass eine Zahl `n` von mindestens 1024 Binärstellen mit heutiger Rechner-Technologie nicht "faktorisiert" werden kann.

Aber die CPU-Auslastung bei diesem Algorithmus ist minimal. Es wird gerade einmal ein Kern eingesetzt (im Schnitt)…



…und es dauert seine Zeit, bis man 100 grosse Primzahlen gefunden hat.

```
...
2017-03-30 15:51:30,440 INFO  - 97. 60559010175509212419...
2017-03-30 15:51:30,548 INFO  - 98. 76755878890045474159...
2017-03-30 15:51:31,086 INFO  - 99. 15775723834218964265...
2017-03-30 15:51:35,350 INFO  - 100. 67266396461594438466...
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
Total time: 1:11.934s
Finished at: Thu Mar 30 15:51:35 CEST 2017
Final Memory: 7M/309M
------------------------------------------------------------------------
```

Parallelisieren Sie den Algorithmus!

- Überlegen Sie sich, wie Sie den Algorithmus nebenläufig ausführen können.
- Überlegen Sie sich, welche Information die Applikation benötigt, damit sie weiss, wann die 100 (oder wie viel auch immer) grossen Primzahlen erreicht sind.
- Bedenken Sie, nicht jede zufällige generierte `BigInteger` Zahl ist eine Primzahl.
- Überlegen Sie sich, wie Sie die parallel anfallenden `BigInteger` Primzahlen in ein File speichern wollen.
- Die Applikation soll nach getaner Arbeit sauber beenden. Ein `System.exit` ist nicht erlaubt.

4.4 Reflektion

Reflektieren Sie die Aufgabe (hilft auch bei einer eventuellen Präsentation) und beantworten Sie sich die folgenden Fragen:

- Wie lange dauert es jetzt?
- Wie viele Threads lassen Sie laufen?
- Was passiert, wenn die Anzahl Threads verdoppeln, vervierfachen, verzehnfachen?
- Können Sie die Applikation noch schneller machen?

5 Optional: Container Thread-sicher machen

5.1 Lernziele

- Sie kennen die Wrapper Klassen um herkömmliche Datenstrukturen Thread-sicher zu gestalten.
- Sie können `Callable`, `Future` und `ExecutorService`, sowie Thread Pool Executoren anwenden.

5.2 Grundlagen

Diese Aufgabe basiert auf dem AD Input N31 (Callables und Executors, sowie Thread-sichere Container).

Für diese Aufgabe gibt es Produzenten, die Integer-Zahlen in der folgenden Art produzieren und die Summe der produzierten Zahlen zurückgeben.

```
private final List<Integer> list;
//...
long sum = 0;
for (int i = 0; i < maxRange; i++) {
    sum += i;
    list.add(i);
}
return sum;
```

Und es gibt EINEN Konsumenten, der die Integer-Zahlen ausliest und die Summe der gelesenen Zahlen zurückgibt.

```
private final List<Integer> list;
//...
long sum = 0;
Iterator<Integer> iterable = list.iterator();
while (iterable.hasNext()) {
    sum += iterable.next();
}
return sum;
```

5.3 Aufgabe

a) Implementieren Sie je eine Produzenten- und Konsumenten-Klasse vom Typ `Callable`.

b) Erstellen Sie eine Demonstration (Klasse oder Test) bei der Sie die Produzenten und Konsumenten jeweils auf eine gemeinsame Liste (z.B. `LinkedList`) zugreifen lassen. Lesen Sie alle Produzentensummen aus, addieren diese zu einem Total und vergleichen das mit der Konsumentensumme.

   Was stellen Sie fest?

c) Machen Sie aus der gemeinsamen Liste eine synchronisierte Liste.

   Was stellen Sie jetzt fest?

d) Wie müssen Sie Ihre Demonstration umbauen, wenn Sie eine `ConcurrentModificationException` sehen wollen?

e) Wenn Sie möchten können Sie die Demonstration auch mit einer Blocking Queue implementieren und dann vergleichen, welche Thread-sicheren Container schneller, d.h. effizienter sind.