```java
/*
 * 1 2 | 3 4 | 5 6 | 7 8 Array
 * 3 7 11 15 Blocksummen: Parallel
 * 3 10 21 36 Prefixsummen fÃ¼r die Blocksummen: Sequentiell
 * 0+1 0+1+2 | 3+3 3+3+4 | 10+5 10+5+6 | 21+7 21+7+8 Prefixsummen: Parallel
 */

import java.util.concurrent.ThreadLocalRandom;
import java.util.Arrays;

public class PrefixSumCalculator {
    private static int[] prefixSumSequential(int[] in) {
        int[] out = new int[in.length];

        out[0] = 0;
        for (int i = 1; i < in.length; ++i)
            out[i] = out[i - 1] + in[i - 1];

        return out;
    }

    private class Shared {
        int[] in;
        int[] out;
        int[] blockSum;
        int[] blockPrefix;
        int blockCount;
        int blockSize;
    }

    private Shared shared = new Shared();

    private class SumCalculator extends Thread {
        int block_id;
        int start_index;

        public SumCalculator(int block_id, int start_index) {
            this.block_id = block_id;
            this.start_index = start_index;
        }

        @Override
        public void run() {
            shared.blockSum[block_id] = 0;
            for (int i = this.start_index; i < this.start_index + shared.blockSize; ++i)
                shared.blockSum[block_id] += shared.in[i];
        }
    }

    private class BlockPrefixSumCalculator extends Thread {
        int block_id;
        int start_index;

        public BlockPrefixSumCalculator(int block_id, int start_index) {
            this.block_id = block_id;
            this.start_index = start_index;
        }

        @Override
        public void run() {
            shared.out[start_index] = shared.blockPrefix[block_id];
            for (int i = start_index + 1; i < start_index + shared.blockSize; ++i)
                shared.out[i] = shared.out[i - 1] + shared.in[i - 1];
        }
    }

    private int[] prefixSumParallel(int[] in, int blockSize) {
        if (in.length % blockSize != 0)
            throw new RuntimeException("Array must be evenly dividable by the block size");

        shared.in = in;
        shared.blockSize = blockSize;
        shared.blockCount = in.length / blockSize;
        shared.out = new int[in.length];
        shared.blockSum = new int[shared.blockCount];
        shared.blockPrefix = new int[shared.blockCount];

        Thread[] threads = new Thread[shared.blockCount];
        for (int i = 0; i < shared.blockCount; ++i) {
            threads[i] = new SumCalculator(i, i * blockSize);
            threads[i].start();
        }

```

```
84:         for (int i = 0; i < shared.blockCount; ++i) {
85:             try {
86:                 threads[i].join();
87:             } catch (InterruptedException e) {
88:             }
89:         }
90:
91:         shared.blockPrefix[0] = 0;
92:         for (int i = 1; i < shared.blockCount; ++i)
93:             shared.blockPrefix[i] = shared.blockPrefix[i - 1] + shared.blockSum[i - 1];
94:
95:         for (int i = 0; i < shared.blockCount; ++i) {
96:             threads[i] = new BlockPrefixSumCalculator(i, i * blockSize);
97:             threads[i].start();
98:         }
99:
100:        for (int i = 0; i < shared.blockCount; ++i) {
101:            try {
102:                threads[i].join();
103:            } catch (InterruptedException e) {
104:            }
105:        }
106:
107:        return shared.out;
108:    }
109:
110:    public static void main(String[] args) {
111:        int[] array = new int[100];
112:        int[] blockSizes = { 2, 5, 10 };
113:
114:        for (int i = 0; i < array.length; ++i)
115:            array[i] = ThreadLocalRandom.current().nextInt(1000);
116:
117:        for (int i = 0; i < blockSizes.length; ++i) {
118:            int[] correctPrefixSum = prefixSumSequential(array);
119:
120:            PrefixSumCalculator calculator = new PrefixSumCalculator();
121:            int[] parallelPrefixSum = calculator.prefixSumParallel(array, blockSizes[i]);
122:
123:            if (!Arrays.equals(correctPrefixSum, parallelPrefixSum))
124:                throw new RuntimeException("Arrays are not equal!");
125:        }
126:
127:        System.out.println("OK");
128:    }
129: }
```

 Moritz Lunk, bt712482 | Yannik Schroeder, bt722248 | Justin Stoll, bt709742

```
 1: import java.util.LinkedList;
 2: import java.util.Random;
 3:
 4: public class App {
 5:     public static class Processor {
 6:         LinkedList<Integer> list = new LinkedList<>();
 7:         Random generator = new Random();
 8:         final int listCapacity = 10;
 9:         int valueCount = 0;
10:
11:         public void produce() throws InterruptedException {
12:             while (true) {
13:                 synchronized (this) {
14:                     while (list.size() == listCapacity)
15:                         wait();
16:
17:                     int value = valueCount++;
18:                     list.add(value);
19:                     System.out.println("Produced: " + value);
20:
21:                     notifyAll();
22:                 }
23:             }
24:         }
25:
26:         public void consume() throws InterruptedException {
27:             while (true) {
28:                 synchronized (this) {
29:                     while (list.isEmpty())
30:                         wait();
31:
32:                     int value = list.removeFirst();
33:                     System.out.println("Consumed: " + value);
34:                     notifyAll();
35:                 }
36:
37:                 Thread.sleep(generator.nextInt(100));
38:             }
39:         }
40:     }
41:
42:     public static void main(String[] args) {
43:         Processor processor = new Processor();
44:         int threadCount = Integer.parseInt(args[0]);
45:         System.out.println("Thread count: " + threadCount);
46:
47:         for (int i = 0; i < threadCount; ++i) {
48:             Thread thread;
49:
50:             if (i % 2 == 0) {
51:                 thread = new Thread(() -> {
52:                     try {
53:                         processor.consume();
54:                     } catch (InterruptedException e) {
55:                         e.printStackTrace();
56:                     }
57:                 });
58:             } else {
59:                 thread = new Thread(() -> {
60:                     try {
61:                         processor.produce();
62:                     } catch (InterruptedException e) {
63:                         e.printStackTrace();
64:                     }
65:                 });
66:             }
67:
68:             thread.start();
69:         }
70:         try {
71:             Thread.sleep(30000);
72:         } catch (InterruptedException e) {
73:         }
74:
75:         System.exit(0);
76:     }
77: }
```

```c
/**
 * a) Bei aggressiven Optimierungen stellt der Compiler fest, dass der Wert von var_updated_flag
 * innerhalb der while-Schleife nicht veraendert werden kann. Solange eine Variable nicht mit
 * volatile gekennzeichnet wird, geht der Compiler nicht davon aus, dass der Wert von auÃ\237erhalb
 * geaendert werden kann und ueberprueft ihn daher nur einmal.
 * Bonuspunkt: Die Antwort ist doch in der Aufgabe gegeben?! Da puts potenziell var_updated_flag
 * veraendern koennte, muss der Wert nach jedem Aufruf wieder neu geladen und verglichen werden.
 *
 * b) Der Compiler folgt der as-if-rule: "The rule that allows any and all code transformations that
 * do not change the observable behavior of the program". In Einzelthread-Code duerfen Anweisungen
 * in unterschiedlichen Reihenfolgen ausgefuehrt werden, was die Reihenfolge der Anweisungen
 * generiert vom Compiler und zur Ausfuehrungszeit auf der CPU beeinflussen kann, solange sich
 * dadurch das Verhalten des Programms im Einzelthread-Modus nicht aendert. Beim Multithreading ist
 * die Reihenfolge von Speicherzugriffen zwischen den verschiedenen Threads meist nicht sequentiell,
 * was zu dem beobachteten Phaenomen fuehrt. Loesen laesst sich das, indem Memory-Ordering explizit
 * angegeben wird (vgl. https://en.cppreference.com/w/cpp/atomic/memory_order.html)
 *
 * c) Weil Pthread-Objekte eben genau das kapseln und intern verwenden
   d)
    volatile verhindert das Umordnen der der Instruktionen auf die volatile Variable.
    Barrieren werden implizit von Java erzeugt, wenn diese benÃ¶tigt werden.

    Beim Verwenden von sychronized werden beim Betreten des gesperrten Bereichs alle geteilten Daten
   neu geladen und beim Verlassen alle geteilten Daten zurÃ¼ckgeschrieben. Das bedeutet, dass alle
   Ã\204nderungen immer vollstÃ¤ndig sichtbar sind, wenn der nÃ¤chste Thread auf die Daten zugreift. Da
   die Daten dadurch immer auf dem neuesten Stand sind, wird volatile nicht benÃ¶tigt.
 */

#include <stdatomic.h>
#include <stdbool.h>
#include <stdio.h>

#define OLD 0
#define NEW 1
atomic_bool var_updated_flag = false;
int var = OLD;

void thread_a() // executed by thread a
{
    var = NEW;
    puts("Thread a has updated var!");

    // Alternativ kann man auch eine Memory-Barrier explizit angeben mit (gcc-builtin):
    // __sync_synchronize();

    // memory_order_release erzwingt, dass die vorige var = NEW Zuweisung in anderen Threads
    // sichtbar ist und nicht umsortiert werden kann
    atomic_store_explicit(&var_updated_flag, true, memory_order_release);
    puts("Thread a has notified thread b!");
}

int thread_b() // executed by thread b
{
    while (!atomic_load_explicit(&var_updated_flag, memory_order_acquire)) {
        // puts("Thread b is still waiting!");
    }
    puts("Thread b was notified!");
    if (var == NEW) {
        puts("Thread b has correctly read the new value of var!");
        return 0;
    } else // var == OLD
    {
        puts("Thread b has unexpectedly read the old value of var!");
        return -1;
    }
}
```