Parallel Programming Exercise Session 6

Spring 2025



Post-Discussion Ex. 5 (+ some theory) Ex.5 Theory Tasks **Ex.5** Programming Tasks Theory Pre-Discussion Ex. 6 **Big Kahoot** Theory Recap based on Kahoot results

If we still have time: hard Wait/Notify exam task

Exam Preparation Session

Monday, March 31, 11:15 – 12:00 Tuesday, April 1, 10:15 – 12:00

HG F 5 / HG F 7

Hosted by ??? / Vera Schubert and Jackson Stanhope

Theory + Post-Discussion Ex. 5

Ex. 5 Theory Tasks

Recall: Amdahl's vs Gustafson's Law

The key goal is to:

- → Understand the main difference and implications (i.e., when to use which formula)
- → Know how to derive the formulas based on your understanding, not because you memorized them for the exam

Recall: Amdahl's vs Gustafson's Law



Amdahl's Law Derivation



Less time for the parallel part

Gustafson's Law Derivation

Gustafson's Law



More work in the same time

- Work with 1 processor
- Work with p processors
 - sequential fraction

$$W_1 = W_1 f + W_1 (1-f)$$

W

 $W_{\rm p}$

f

$$W_p = W_1 f + W_1 (1-f) p$$

$$S_p$$
 - speedup
 $S_p = W_p/W_1$
 $S_p = f + (1-f)p$

fib(4) task graph

```
public class Fibonacci {
    public static long fib(int n) {
        if (n < 2) {
            return n;
        }
        spawn task for fib(n-1);
        spawn task for fib(n-2);
        wait for tasks to complete
        return addition of task results
    }
</pre>
```

}

fib(4) task graph FJ



```
public class Fibonacci {
    public static long fib(int n) {
        if (n < 2) {
            return n;
        }
        spawn task for fib(n-1);
        spawn task for fib(n-2);
        wait for tasks to complete
        return addition of task results
}</pre>
```

What is a task?

What is an edge?

spawn, same procedure, wait

fib(4) simplified task graph



Simpler at the expense of not modelling joins and inter-process dependencies

```
public class Fibonacci {
    public static long fib(int n) {
        if (n < 2) {
            return n;
        }
        spawn task for fib(n-1);
        spawn task for fib(n-2);
        wait for tasks to complete
        return addition of task results
}</pre>
```

What is a task?

Call to Fibonacci

What is an edge?

spawn (no dependency within same procedure)

fib(4) simplified task graph



Simpler at the expense of not modelling joins and inter-process dependencies

```
public class Fibonacci {
    public static long fib(int n) {
        if (n < 2) {
            return n;
        }
        spawn task for fib(n-1);
        spawn task for fib(n-2);
        wait for tasks to complete
        return addition of task results
}</pre>
```

What is a task?

Call to Fibonacci

What is an edge?

spawn (no dependency within same procedure)

How would you memoize in this example?

- Shared array (initialize with -1)
- We don't need to synchronize since different threads will write the same value to the same entry
- Reading the values is no problem

essentials

Task Graphs



Critical path: path from start to end that takes the longest (for some metric)

Example: #nodes

essentials

fib(4) task graph FJ



public class Fibonacci { public static long fib(int n) { **if** (n < 2) { return n; spawn task for fib(n-1); spawn task for fib(n-2); wait for tasks to complete return addition of task results

What is a task?

What is an edge?

spawn, same procedure, wait

critical path length is 7 tasks

fib(4) simplified task graph



```
public class Fibonacci {
    public static long fib(int n) {
        if (n < 2) {
            return n;
        }
        spawn task for fib(n-1);
        spawn task for fib(n-2);
        wait for tasks to complete
        return addition of task results
    }
</pre>
```

What is a task?

Call to Fibonacci

What is an edge?

critical path length is 4 tasks

spawn (no dependency within same procedure)

Task Graph Simplified

Task: Call to add() Cut-off: 1

Adding eight numbers:



Task Graph Simplified

Task: Call to add() Cut-off: 1

Adding eight numbers:

What is the corresponding task graph?



essentials

Task Graph Simplified

Task: Call to add()

Cut-off: 1

What is the corresponding task graph?

Adding eight numbers:



essentials

Task Graph FJ

Task: fork, join, continuation Cut-off: 1

What is the corresponding task graph?



Task Graph Simplified

Task: Call to add() Cut-off: 1

Adding eight numbers:

What is the corresponding task graph?



essentials

Task Graph Simplified

Task: Call to add() Cut-off: 1

Adding eight numbers:







essentials

Task Graph FJ

Task: fork, join, continuation Cut-off: 1

Adding eight numbers:

What is the corresponding task graph?





Task Graphs

A wide task graph \rightarrow higher potential parallelism

A deep task graph \rightarrow more sequential dependencies

"Easy" points

- Usually tasks about Amdahl's / Gustafson's Law, Pipelining, Task Graphs are the easy tasks in the exam
- You can definitely collect about ~25% of the points by solving those tasks.
- Practicing for those tasks is straightforward
 - Get familiar with the differences of Amdahl and Gustafson. Be able to derive the formulas by yourself
 - Understand formulas intuitively: Amdahl, Gustafson, Pipelining
 - Practice Task Graphs

Ex. 5 Programming Tasks

Task 1: Search And Count

Search an array of integers for a certain feature and count integers that have this feature:

- Light workload: count number of non-zero values.
- Heavy workload: count how many integers are prime numbers.

We will study single threaded and multi-threaded implementation of the problem.

Task 1 A: Search And Count - Sequential

```
public class SearchAndCountSingle {
  private int[] input;
  private Workload.Type type;
```

```
private SearchAndCountSingle(int[] input, Workload.Type wt) {
  this.input = input;
  this.type = wt;
}
```

```
private int count() {
    int count = 0;
    for (int i = 0; i < input.length; i++) {
        if (Workload.doWork(input[i], type)) count++;
     }
    return count;
}</pre>
```

Straightforward implementation. Simply iterate through the input array and count how many times given

event occurs.

Divide and Conquer

Basic structure of a divide-and-conquer algorithm:

- 1. If problem is small enough, solve it directly
- 2. Otherwise
 - a. Break problem into subproblems
 - b. Solve subproblems recursively
 - c. Assemble solutions of subproblems into overall solution

Divide and Conquer



What determines a task?

ii) start index

i) input array

iii) length/end index

These are fields we want to store in the task



SearchAndCountThreadDivideAndConquer.java

- → Divide and conquer
- → Do not create more threads than numThreads

Divide and Conquer Parallelization

• • •



Task B:

Extend your implementation such that it creates only a fixed number of threads. Make sure that your solution is properly synchronized when checking whether to create a new thread

How to achieve this?

Divide and Conquer Parallelization



Option 1:

Shared counter with synchronized/atomic access

Option 2:

Assign unique sequential id to each task. Spawn threads for first N tasks.

- + no synchronization required
- imbalanced amount of work

Alternative approach (from homework submissions)

Instead of using ids, we give the child tasks "numThreads / 2" since that's the amount of threads that are allowed to be created in the subtrees.

Let's take a look at the master solution


Implement it with ExecutorService

Let's take a look at the master solution.

ExecutorService

TPS01-J. Do not execute interdependent tasks in a bounded thread pool

Created by Dhruv Mohindra, last modified by Carol J. Lallier on Jun 22, 2015

Bounded thread pools allow the programmer to specify an upper limit on the number of threads that can concurrently execute in a thread pool. Programs must not use threads from a bounded thread pool to execute tasks that depend on the completion of other tasks in the pool.

A form of deadlock called thread-starvation deadlock arises when all the threads executing in the pool are blocked on tasks that are waiting on an internal queue for an available thread in which to execute. Threadstarvation deadlock occurs when currently executing tasks submit other tasks to a thread pool and wait for them to complete and the thread pool lacks the capacity to accommodate all the tasks at once.

This problem can be confusing because the program can function correctly when fewer threads are needed. The issue can be mitigated, in some cases, by choosing a larger pool size. However, determining a suitable size may be difficult or even impossible.

Similarly, threads in a thread pool may fail to be recycled when two executing tasks each require the other to complete before they can terminate. A blocking operation within a subtask can also lead to unbounded queue growth [Goetz 2006].

Fork/Join: recommended for Divide and Conquer tasks as they have strong task interdependency

ExecutorService: for handling many independent requests where tasks are standalone

Divide and Conquer vs Fork/Join

Divide And Conquer

Fundamental design pattern based on recursively breaking down a problem into smaller problems that can be combined to give a solution to the original problem

Fork/Join

A framework that supports Divide and Conquer style parallelism

Divide and Conquer vs Fork/Join



a framework that supports Divide and Conquer style parallelism problems are solved in parallel

Search And Count

```
protected Integer compute() {
  if (// work is small)
```

```
// do the work directly
```

else {
 // split work into pieces

```
// invoke the pieces and
wait for the results
```

// combine the results

```
private int length;
                                                        private int cutOff;
protected Integer compute() {
                                                        private Workload.Type type;
  if (length <= cutOff) {</pre>
    int count = 0;
   for (int i = start; i < start + length; i++) {</pre>
      if (Workload.doWork(input[i], type)) count++;
    return count;
  else {
    int half = (length) / 2;
    SearchAndCountMultiple sc1 =
      new SearchAndCountMultiple(input, start, half, cutOff, type);
    SearchAndCountMultiple sc2 =
      new SearchAndCountMultiple(input, start + half, length - half, cutOff, type);
    sc1.fork();
    sc2.fork();
    int count1 = sc1.join();
    int count2 = sc2.join();
    return count1 + count2;
```

essentials

public class SearchAndCountMultiple extends RecursiveTask<Integer> {

private int[] input;

private int start;



Lock Object

Shared object that satisfies the following interface

```
public interface Lock{
    public void lock(); // entering CS
    public void unlock(); // leaving CS
}
```

providing the following semantics

new Lock	make a new lock, initially "not held"
acquire	blocks (only) if this lock is already currently <i>"held"</i> Once <i>"not held",</i> makes lock <i>"held"</i> [all at once!]
release	makes this lock <i>"not held"</i> If >= 1 threads are blocked on it, exactly 1 will acquire it



Required Properties of Mutual Exclusion

Safety Property

§ At most one process executes the critical section code



Required Properties of Mutual Exclusion

Safety Property

§ At most one process executes the critical section code



Liveness

§ Minimally: acquire_mutex must terminate in finite time when no process executes in the critical section

Almost-correct pseudocode

```
class BankAccount {
  private int balance = 0;
                                           One lock for
  private Lock lk = new Lock();
                                           each account
  ...
  void withdraw(int amount) {
     lk.lock(); // may block
     int b = getBalance();
     if(amount > b)
       throw new WithdrawTooLargeException();
     setBalance(b - amount);
     lk.unlock();
  }
  // deposit would also acquire/release lk
}
```

Almost-correct pseudocode

```
class BankAccount {
  private int balance = 0;
                                           One lock for
  private Lock lk = new Lock();
                                           each account
  ...
  void withdraw(int amount) {
     lk.lock(); // may block
     int b = getBalance();
     if(amount > b)
       throw new WithdrawTooLargeException();
     setBalance(b - amount);
     lk.unlock();
  }
  // deposit would also acquire/release lk
}
```

Lock won't be released if exception is thrown!

Solution: Use try/finally block!



Always gets executed (even after exception or return)

Possible mistakes

Incorrect: Use different locks for withdraw and deposit

- § Mutual exclusion works only when using same lock
- § **balance** field is the shared resource being protected

Poor performance: Use same lock for every bank account

§ No simultaneous operations on different accounts

Incorrect: Forget to release a lock (blocks other threads forever!)

§ Previous slide is wrong because of the exception possibility!

```
if(amount > b) {
    lk.unlock(); // hard to remember!
    throw new WithdrawTooLargeException();
}
```

Re-entrant lock

A re-entrant lock (a.k.a. recursive lock)

"remembers"

- § the thread (if any) that currently holds it
- § a count



When the lock goes from held to not-held, the count is set to 0

If (code running in) the current holder calls **lock (acquire)**:

- § it does not block
- § it increments the count

On unlock (release):

- § if the count is > 0, the count is decremented
- § if the count is 0, the lock becomes *not-held*

Re-entrant locks work

- § This simple code works fine provided 1k is a reentrant lock
- § Okay to call **setBalance** directly
- § Okay to call withdraw (won't block forever)

```
int setBalance(int x) {
  lk.lock();
  balance = x;
  lk.unlock();
}
void withdraw(int amount) {
  lk.lock();
  ...
  setBalance(b - amount);
  lk.unlock();
```

Race condition

A **Race Condition** occurs in concurrent programming when the correctness of the system depends on the specific interleaving or ordering of operations executed by multiple threads or processes.

Typically, problem is some *intermediate state* that "messes up" a concurrent thread that "sees" that state

Note: This lecture makes a big distinction between *data races* and *bad interleavings*, both instances of race-condition bugs

§ Confusion often results from not distinguishing these or using the ambiguous "race condition" to mean only one

The distinction

Data Race [aka *Low Level Race Condition, low semantic level*] Erroneous program behavior caused by insufficiently synchronized accesses of a shared resource by multiple threads, e.g. Simultaneous read/write or write/write of the same memory location

(for mortals) always an error, due to compiler & HW

Bad Interleaving [aka *High Level Race Condition, high semantic level*] Erroneous program behavior caused by an unfavorable execution order of a multithreaded algorithm that makes use of otherwise well synchronized resources.

"Bad" depends on your specification

The prefix-sum problem

Given int[] input,

```
produce int[] output where:
```

```
output[i] = input[0] + input[1] + ... + input[i]
```

```
Sequential prefix-sum
```

```
int[] prefix_sum(int[] input){
    int[] output = new int[input.length];
    output[0] = input[0];
    for(int i = 1; i < input.length; i++)
        output[i] = output[i-1] + input[i];
    return output;
}</pre>
```

Does not seem parallelizable

- Work: O(n), Span: O(n)
- This algorithm is sequential, but a different algorithm has: Work O(n), Span O(log n)

How to compute the prefix-sum in parallel?









input	6	4	16	10	16	14	2	8
output	6	10	26	36	52	66	68	76











Parallel prefix-sum

The parallel-prefix algorithm does two passes

- Each pass has O(n) work and O(log n) span
- So in total there is O(n) work and O(log n) span
- So like with array summing, parallelism is n/log n

First pass builds a tree bottom-up: the "up" pass

Second pass traverses the tree top-down: the "down" pass





Parallel Patterns

- We are now quite familiar with how to parallize algorithms
- There are a few recurring patterns that are important to know

Map, Reduction, Stencil, Scan, Pack

Reduction

- A reduction is an operation that produces a single answer from a collection (array etc) via an **associative** operator.
- Needs to be associative. Otherwise divide-and-conquer won't work

Example: array sum

Map

- Operates on each element of the input data independently (each array element)
- Output is the same size \rightarrow no size reduction
- Doesn't have to be the same operation on each element

Example: add two arrays

Stencil

- Like map but can take more than one element as input
- Generalization of map and thus also no size reduction

Example:

Image \rightarrow apply averaging filter on each pixel Update a value based on its neighbors

Never do it in-place because you would then take values that are already output values.

Scan

- Collection of data $X \rightarrow$ return collection of data Y
- Y(i) = functionOf(Y(i 1) & X(i))
- Seems sequential because of dependencies
- Can parallelize if function is associative \rightarrow O(log(n)) span

Example: parallel prefix sum

Pack

Collection of data X → return collection of data X if fulfill condition


Pack

- First compute bit vector
- Then find index in result array (prefix sum on bit vector)



Pre-Discussion Exercise 6

Assignment 6

- Task Parallelism:
- Merge Sort
- Longest Sequence

Merge sort algorithm

In this exercise you will implement the merge sort algorithm using task parallelism.

The merge sort algorithm partitions the array into smaller arrays, sorts each one separately and then merges the sorted arrays.

- By default, the partitioning of the array continues recursively until the array size is 1 or 2, which then is sorted trivially.
- Try larger cutoff values (e.g partition arrays down to minimum size 4 instead of 2) and see how this affects the algorithm performance.
- Discuss the asymptotic running time of the algorithm and the obtained speedup.

Longest Sequence

Given a sequence of numbers:

find the longest sequence of the same consecutive number.

If multiple sequences have the same length, return the first one (the one with lowest starting index) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] [1, 1, 0, 0]

Longest Sequence

Task:

Implement task parallel version that finds the longest sequence of the same consecutive number.

Challenge:

The input array cannot be divided arbitrarily. For example:









Wir betrachten 3-Wege-Kreisel (siehe Abbildung 1a). Der Kreisverkehr kann in 3 Zonen unterteilt werden: Z0, Z1 und Z2. Die Einfahrten in den Kreisverkehr sind mit E0, E1 und E2 gekennzeichnet. Die Ausfahrten sind mit X0, X1 und X2 gekennzeichnet.

Beispiel: Ein an E0 ankommendes Auto, welches den Kreisel bei Ausfahrt X2 verlassen will, muss zunächst anstehen, bis die zuvor an E0 angekommenen Autos abgefahren sind. Das Auto muss dann warten, bis die Zone Z0 frei ist, um in den Kreisverkehr einfahren zu können (siehe Abbildung 1b). Vor dem Überqueren der Zone Z1 muss das Auto warten, bis Z1 frei ist (siehe Abbildung 1c). Das Auto kann dann bei X2 den Kreisel verlassen. **Um Deadlocks zu vermeiden, dürfen sich im Kreisverkehr nie mehr als 2 Autos gleichzeitig befinden!** We consider 3-way roundabouts (see Figure 1a). The roundabout can be segmented into 3 zones: Z0, Z1, and Z2. The entries into the roundabout are labeled E0, E1, and E2. The exits are labeled X0, X1, and X2.

Example: A car arriving at E0, intending to leave the roundabout at exit X2, must first wait in line until the cars that have arrived earlier at E0 have left. The car then has to wait for zone Z0 to be free to enter the roundabout (see Figure 1b). Before crossing zone Z1, the car has to wait for Z1 to be free (see Figure 1c). The car can then leave at X2. To avoid deadlocks, no more than 2 cars can be in the roundabout at any time!



Wir betrachten 3-Wege-Kreisel (siehe Abbildung 1a). Der Kreisverkehr kann in 3 Zonen unterteilt werden: Z0, Z1 und Z2. Die Einfahrten in den Kreisverkehr sind mit E0, E1 und E2 gekennzeichnet. Die Ausfahrten sind mit X0, X1 und X2 gekennzeichnet.

Beispiel: Ein an E0 ankommendes Auto, welches den Kreisel bei Ausfahrt X2 verlassen will, muss zunächst anstehen, bis die zuvor an E0 angekommenen Autos abgefahren sind. Das Auto muss dann warten, bis die Zone Z0 frei ist, um in den Kreisverkehr einfahren zu können (siehe Abbildung 1b). Vor dem Überqueren der Zone Z1 muss das Auto warten, bis Z1 frei ist (siehe Abbildung 1c). Das Auto kann dann bei X2 den Kreisel verlassen. Um Deadlocks zu vermeiden, dürfen sich im Kreisverkehr nie mehr als 2 Autos gleichzeitig befinden! We consider 3-way roundabouts (see Figure 1a). The roundabout can be segmented into 3 zones: Z0, Z1, and Z2. The entries into the roundabout are labeled E0, E1, and E2. The exits are labeled X0, X1, and X2.

Example: A car arriving at E0, intending to leave the roundabout at exit X2, must first wait in line until the cars that have arrived earlier at E0 have left. The car then has to wait for zone Z0 to be free to enter the roundabout (see Figure 1b). Before crossing zone Z1, the car has to wait for Z1 to be free (see Figure 1c). The car can then leave at X2. To avoid deadlocks, no more than 2 cars can be in the roundabout at any time!

Warum?

Vervollständigen Sie das Code-Skelett der Klasse Car gemäss der Beschreibung in den Kommentaren. Ihre Lösung sollte notify(), notifyAll(), wait(), das synchronized-Schlüsselwort und die vorhandenen Locks und Atomic Integers verwenden, um das Programm korrekt zu synchronisieren, aber Sie sollen keine unnötige Synchronisation einführen. In dieser Aufgabe müssen Sie Exceptions nicht behandeln. Complete the code skeleton of the class Car according to the description in the comments. Make sure that your solution is properly synchronized by using notify(), notifyAll(), wait(), the synchronized keyword, and the provided locks and atomic integers, where required, but do not introduce unnecessary synchronization. For this task, you do not need to handle exceptions.

```
public class Entry {
    private int index;
    // Die Ticketnummer, die dem Auto gegeben wird,
    // welches sich als nächstes hinten an der Warteschlange anstellt.
    private AtomicInteger next ticket = new AtomicInteger(initialValue:0);
    // Die Ticketnummer des Autos am Anfang der Warteschlange.
    private AtomicInteger next car in line = new AtomicInteger(initialValue:0);
    Entry(int index) {
        this.index = index;
    public int get_index() {
        return index;
    public int get ticket() {
        return next_ticket.getAndIncrement();
    public int get_next_car_in_line() {
        return next_car_in_line.get();
    public void car_clears_entry(int car_ticket) {
        int expected_ticket = get_next_car_in_line();
        assert car_ticket == expected_ticket : "Cars must wait their turn.";
        next car in line.incrementAndGet();
```

```
class Zone {
    private int index;
    public final Lock mutex = new ReentrantLock();
    Zone(int index) {
        this.index = index;
    }
    public int get_index() {
        return index;
    }
```

class Car implements Runnable {

```
// this.zones enthält alle Zonen, welche das Auto passieren will, geordnet.
private final List<Zone> zones = new ArrayList<>();
private final Entry entry, exit;
private final int ticket_number;
private final AtomicInteger n cars on roundabout;
Car(Zone[] zones, Entry entry, Entry exit, AtomicInteger n_cars_on_roundabout) {
    this.entry = entry;
    this.exit = exit;
    this.ticket_number = entry.get_ticket();
    int zone_index = this.entry.get_index();
    this.n_cars_on_roundabout = n_cars_on_roundabout;
    while (zone_index != this.exit.get_index()) {
        this.zones.add(zones[zone index]);
        zone_index = (zone_index + 1) % zones.length;
@Override
public void run() {
```

```
@Override
public void run() {
    while (true) {
        synchronized (/* TODO */) {
            if (this.ticket_number > this.entry.get_next_car_in_line()) {
                // Vorrang für Autos, die früher bei derselben Einfahrt ankamen.
            } else {
                break;
    /*
     * Bevor das Auto in den Kreisverkehr einfährt,
     * darf höchstens ein Auto im Kreisverkehr sein.
     */
    while (true){
    Zone current_zone = this.zones.get(index:0);
    // Stellen Sie sicher, dass die entsprechende Zone frei ist...
    // ...und dass das nächste wartende Auto an dieser Einfahrt passieren kann.
    for (int i = 1; i < this.zones.size(); ++i) {</pre>
        Zone zone = this.zones.get(i);
        // Stellen Sie sicher, dass die n\u00e4chste Zone frei ist,
        // und machen Sie die vorherige Zone frei.
        current_zone = zone;
      Das Auto verlässt den Kreisverkehr
```

```
@Override
public void run() {
    while (true) {
        synchronized (/* TODO */) {
            if (this.ticket_number > this.entry.get_next_car_in_line()) {
                // Vorrang für Autos, die früher bei derselben Einfahrt ankamen.
            } else {
                break;
    /*
     * Bevor das Auto in den Kreisverkehr einfährt,
     * darf höchstens ein Auto im Kreisverkehr sein.
     */
    while (true){
    Zone current_zone = this.zones.get(index:0);
    // Stellen Sie sicher, dass die entsprechende Zone frei ist...
    // ...und dass das nächste wartende Auto an dieser Einfahrt passieren kann.
    for (int i = 1; i < this.zones.size(); ++i) {</pre>
        Zone zone = this.zones.get(i);
        // Stellen Sie sicher, dass die n\u00e4chste Zone frei ist,
        // und machen Sie die vorherige Zone frei.
        current_zone = zone;
      Das Auto verlässt den Kreisverkehr
```

```
@Override
public void run() {
    while (true) {
        synchronized (/* TODO */) {
            if (this.ticket_number > this.entry.get_next_car_in_line()) {
                // Vorrang für Autos, die früher bei derselben Einfahrt ankamen.
            } else {
                break;
    /*
     * Bevor das Auto in den Kreisverkehr einfährt,
     * darf höchstens ein Auto im Kreisverkehr sein.
     */
   while (true){_ intt = n_cars_on_roundabout.get();
    Zone current_zone = this.zones.get(index:0);
    // Stellen Sie sicher, dass die entsprechende Zone frei ist...
    // ...und dass das nächste wartende Auto an dieser Einfahrt passieren kann.
    for (int i = 1; i < this.zones.size(); ++i) {</pre>
        Zone zone = this.zones.get(i);
        // Stellen Sie sicher, dass die nächste Zone frei ist,
        // und machen Sie die vorherige Zone frei.
        current_zone = zone;
      Das Auto verlässt den Kreisverkehr
```

```
@Override
public void run() {
    while (true) {
        synchronized (/* TODO */) {
            if (this.ticket_number > this.entry.get_next_car_in_line()) {
                // Vorrang für Autos, die früher bei derselben Einfahrt ankamen.
            } else {
                break;
    /*
     * Bevor das Auto in den Kreisverkehr einfährt,
     * darf höchstens ein Auto im Kreisverkehr sein.
     */
   while (true){_____ intt = n_cars_on_roundabout.get();
    Zone current_zone = this.zones.get(index:0);
    // Stellen Sie sicher, dass die entsprechende Zone frei ist... -
                                                                                 current zone.mutex.lock();
    // ...und dass das nächste wartende Auto an dieser Einfahrt passieren kann.
    for (int i = 1; i < this.zones.size(); ++i) {</pre>
        Zone zone = this.zones.get(i);
        // Stellen Sie sicher, dass die nächste Zone frei ist,
        // und machen Sie die vorherige Zone frei.
        current_zone = zone;
      Das Auto verlässt den Kreisverkehr
```

```
@Override
public void run() {
    while (true) {
        synchronized (/* TODO */) {
            if (this.ticket_number > this.entry.get_next_car_in_line()) {
                // Vorrang für Autos, die früher bei derselben Einfahrt ankamen.
            } else {
                break;
     * Bevor das Auto in den Kreisverkehr einfährt,
     * darf höchstens ein Auto im Kreisverkehr sein.
     */
   while (true){_ intt = n_cars_on_roundabout.get();
    Zone current_zone = this.zones.get(index:0);
    // Stellen Sie sicher, dass die entsprechende Zone frei ist... -
                                                                                  current zone.mutex.lock();
    // ...und dass das nächste wartende Auto an dieser Einfahrt passieren kann.
    for (int i = 1; i < this.zones.size(); ++i) {</pre>
        Zone zone = this.zones.get(i);
                                                                    entry.car_clears_entry(this.ticket_number);
        // Stellen Sie sicher, dass die n\u00e4chste Zone frei ist,
        // und machen Sie die vorherige Zone frei.
        current_zone = zone;
      Das Auto verlässt den Kreisverkehr
```



