Technical University of Denmark
DTU Compute

Concurrent programming
Fall 2014

Car control

Due November 19, 2014

Authors: Group 24

Michael Bøndergaard
(s113112)

Kristoffer Breitenstein
(s113135)

Contents

1 Introduction

The course teaches how to handle software with additional tasks within one software program. Software will have threads handle different tasks and have shared variables, which the threads will have limited or full access to. Concurrent thread can be used for many different objects and goals depending on the kind of project the software is written for. Threads can often be used as optimization in software, where different threads will handle different part of an calculation or another task. Especially with the modern processors running multiple cores at once, then one core can handle a task while another handles a different task.

The project is about concurrent processes running individually and using specific data in cooperation with the other processes. The project has 9 cars driving around a parking lot with the cars having different routes. The cars will have to make sure they do not cause accidents or end up in a dead lock situation. The cars will have to pass through an alley with only room the cars driving in one direction. The cars are run by a thread each to make sure the cars are not getting into an accident.
In the project the alley is managed by either a semaphore or a monitor written in Java. These are two different approaches of how to handle atomic actions in software. In other words mean the variable or a critical section is only available to one thread at a time.

In the end the cars will be driving with the alley acting as a traffic light. The cars only being able to drive in one direction at a time, while the others will have to wait till the alley is available. The cars also has a barrier, where the cars will wait until a specific number of cars are waiting. The traffic will end up going around the circuit in a smooth flow.

2 Step 1

The system given has the cars driving around the circuit in their specific routes. The cars does not stop, when they bump into each other, instead a red square is shown for each collision.

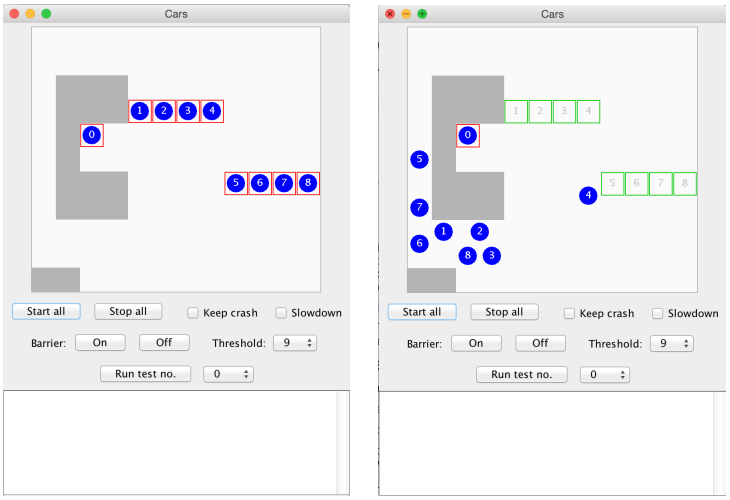
Step one's requirements are to

- make sure the cars do not collide
- cause deadlocks in the traffic
- have cars drive through the alley

2.1 Avoid collision

Overall the challenges is to stop the cars' collisions and have semaphores handle the alley. The deadlocks is a part in both the collision and the alley in a way.

Figure 1: The cars driving around the parking lot



The idea behind avoiding collision is that the cars can not move, if another car is either already in the way or both cars are trying to enter same field. The cars will in both cases stop. A car already being an obstacle, the waiting car will have to wait for the other car to move along. This situation will never cause a deadlock, but will have cars waiting for each other in a line.
The other case where two cars want to enter same field, can cause a deadlock with both cars waiting for the other car to move. A solution to the deadlock situation is to have a way to let one of the cars drive on some condition. The condition we have chosen is to let the cars with the highest number advance. The condition will have to be unique, because otherwise the cars will collide. The condition is unique, since no cars have the same number.

One problem with the solution is, when the cars are driving through the alley a direction will stop. The cars driving up the alley will stop for the cars with

a higher number. The solution is to let the cars in the alley being prioritized higher, than the cars waiting at the alley.

Unfortunately the collision detection does rarely have a collision on a tile. The collisions happen right, when a faster car want to enter a field, where another car is entering. The faster car will have to ask at the right moment for the collision to happen. The collisions does unfortunately happen.
A solution would be to use semaphores for every tile in the map to make sure no car would be at the tile at the same time. The cars would then be waiting correctly for the tile to be free. We unfortunately did not notice the issue until late in the project, while we tested other issues. The implementation seemed to work fine while working on step 1, but found out the accidents does happen so rarely.

2.2 Alley

2.2.1 Ideas and design

The idea for the traffic through the alley is to have the cars drive in one direction. If the cars would be driving from both directions, the result would be a deadlock in mid alley. The cars will have to wait for the alley being vacant to change direction. The direction is actually taken by whomever is first to arrive to the alley.
The idea for our alley class is to use semaphores to manage the traffic. The idea is to use 4 different semaphores to manage different aspects of the alley traffic. One semaphore is used to make sure the maximum amount of cars in the alley is no more than 4 cars. A semaphore is used to wait for the traffic being in the specific cars direction. A semaphore is used to make the bottom queue wait for the traffic to become the up direction. The last semaphore is used to make the leave action atomic, which we found necessary due to our spin results.

2.2.2 Implementation

The implementation of the alley can be found in appendix B.

The implementation includes the semaphore e, d, b and a, which all have different purposes in the program. The e semaphore is used to limit the number of cars entering the alley, which in our case is 4. The d semaphore manages the direction and is used to make the cars wait for a direction change. The semaphore b is used for the bottom queue, where the cars 3 and 4 are waiting for the alley to be vacant. The b semaphore is basically support for the d semaphore, and when the direction is changed to up the b semaphore will be released. Both the queues at the bottom will then go at once as the alley is vacant, this guaranties the bottom queues are set as a group.
The last semaphore a is used to make sure the leave method is being done atomically. We found using spin that there were some mistakes with the release of d, which has to be done atomically. The release of the direction would otherwise enter an invalid state for the direction. The direction could otherwise in rare entities have more coconuts to pass than actually was the purpose.

3 Step 2

The software spin uses Promela models to verify a concurrent model. The model shows rather the processes uses variables and other shared features in a valid and intended way. The processes should not be allowed to enter more than 4 cars at a time and only in one direction.

3.1 Analysis

The Promela code is a little different to get to work than the java code. The semaphore described previously result in the code found in appendix C. The code display the enter part followed by the leave part of the cars alley class in java. The result of the analysis shows with 8 threads the program runs correctly.

```
pan: resizing hashtable to -w29.. pan: out of memory
hint: to reduce memory, recompile with
  -DCOLLAPSE # good, fast compression, or
  -DMA=84    # better/slower compression, or
  -DHC # hash-compaction, approximation
  -DBITSTATE # supertrace, approximation
(Spin Version 6.1.0 -- 4 May 2011)
Warning: Search not completed
        + Partial Order Reduction
Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +
State-vector 84 byte, depth reached 108, ... errors: 0 ...
 2.69e+08 states, stored
6.5625499e+08 states, matched
9.2525499e+08 transitions (= stored+matched)
1.4695379e+08 atomic steps
hash conflicts: 4.0118364e+08 (resolved)
Stats on memory usage (in Megabytes):
28732.300       equivalent memory usage for
states (stored*(State-vector + overhead))
11478.569       actual memory usage for states
    (compression: 39.95%)
                state-vector as stored = 17 byte +
                    28 byte overhead
 4096.000       memory used for hash table (-w29)
    0.107       memory used for DFS stack (-m2000)
    4.417       memory lost to fragmentation
15570.259       total actual memory usage
pan: elapsed time 1.49e+03 seconds
pan: rate     180388 states/second
```

The analysis does not show any errors, but a warning is given "Search not complete". Other analysis have been run, which does not show the warning,

but these are with less processes. These analysis can be found in the appendix C. The problem is that the pan runs out of memory on the thin clients, which result in a termination. The program does the analysis, but takes a long time and so far the biggest analysis run has 6 processes without error. The error can occur on 6 processes, but does not occur every time. For 5 processes the analysis does seem to work every time, but does only have 1 car in the one direction and 4 in the other.
The analysis shows no errors which means the checks (found at the bottom of the code) do hold. The different semaphores are kept within the limits of the variables.

3.2 Implementation

The code in appendix C is a close implementation of the code in java, which can be found in appendix B.

Some of the challenges has been to get the jSpin code to do the exact same as the java code, but we have received some help from the TA to get it right. The challenges was to make the wait function along with the semaphore code to behave as intended. The semaphore was made with inlines for the P and V function. The inline is similar to methods and is easy to use within the project. The P function is done atomic and decrements the specific semaphore value by 1. The reason for begin atomic is to do the entire inline before another process can access the inline. For the V function the semaphore is simply incremented by 1.

4 Step 3

This step concerns the barrier. The barrier has to stop the cars until all cars are waiting at the barrier. Then they should all be released for a single round, and wait when they arrive at the barrier again time.

4.1 Ideas & Design

Before and during the implementation, some ideas were discussed. The cars had to stop at the barrier. Then they had to be released, but only until they arrive at the barrier again. Also the cars should be released, when the barrier is turned off.
The idea was to make the on and off function quite similar to the on and off function from the Gate-class. The only difference it that the off function had to release the waiting cars.
The hard problem was, how to do the sync function right. A counter is used to count waiting cars, and a single semaphore is used to make the cars wait.
The sync function should keep track of number of cars, and whether the barrier is on or off.

4.2 Implementation

The on and off functions were implemented as the on and off functions from the Gate-class almost. The difference is that the off function releases the cars

waiting at the barrier. The on function takes the only coconut in the semaphore b.

The following code is the sync function:

```
try {
        a.P();
} catch (InterruptedException e) {
        throw new InterruptedException();
}
cars++;
a.V();

if (barrierOn) {
        if (cars < 9) {
                try {
                        b.P();
                } catch (InterruptedException e) {
                        cars--;
                        throw new InterruptedException();
                }
        } else {
                for (int i = 0; i < cars; i++) {
                        b.V();
                }
                b.P();
        }
}

try {
        a.P();
} catch (InterruptedException e) {
        cars--;
        throw new InterruptedException();
}
cars--;
a.V();
```

As seen, the sync function increases the cars counter, checks on the barrierOn boolean. Nothing happens in the case that the barrier is not on, meaning the barrier is off. The cars counter will just be decreased again and the car will go on.
In the case that the barrier is on, the car will wait for a coconut in the semaphore, b. Until the cars counter reaches 9, the cars will wait. The semaphore, b, had only a single coconut, which was taken by the on function. When the 9th car comes along, the cars counter will not be less than 9, and it will release a number of coconuts corresponding to the cars counter. This means, that all cars will be released, and when they arrive again, they will wait, since the semaphore has no coconuts.

The on and off functions are to be seen in the appendix, section **??**.

4.3 Extra B

In addition to the simple barrier a threshold is added. The threshold is a number which decides how many cars should be waiting before they are released. The only difference for the sync function is that the number 9 is replaced by the integer threshold. This integer is changed by the BarrierSet function in CarControl class. This function is important for describing how the threshold is changed.
The BarrierSet function is shown in code:

```
if (!bar.barrierOn) {
        bar.threshold = k;
} else {
        if (k > bar.threshold) {
                bar.threshold = k;
        } else {
                bar.threshold = k;
                if (bar.cars >= k) {
                        int d = bar.cars / k;
                        for (int i = 0; i < k * d; i++) {
                                bar.b.V();
                        }
                }
        }
}
```

The k is the number of which the threshold should be equal to. From the code, it is seen that if the barrier is not on, the threshold should be changed right away without any further considerations. On the other side when the barrier is on, it is checked whether the new threshold is bigger or smaller than the previous one. Is it bigger, then it should be changed right away. But if the new threshold, k, is smaller a new check is to be made. In the case that k is smaller than the threshold, it is important to check whether the number of waiting cars are larger or equal to k. Are there more cars waiting than the new threshold, then k number of cars should be released, or d times k cars depending on whether there are more than twice as many cars waiting as the new threshold. Of course when the number of cars waiting is less than the new threshold, the threshold should just be changed without releasing any cars.

5 Step 4

The task of step 4 is to implement the alley and the barrier by monitors instead of semaphores.
When using monitors the semaphore class is not needed. The functions are extended to be synchronized functions, and the thread calls "notifyAll" and "wait" is used.

5.1 Discussion

It is more convenient to use monitors for the barrier and the alley. Since the alley kind of uses groups and the barrier too. The monitor is more convenient having group whereas the semaphore is more oriented against only letting one thread pass at the time.
By using monitors the code is also much less complicated which will be shown during this section.

5.2 Implementation

The functions in the alley and in the barrier class are all synchronized. In addition, the sync function is more simple:

```
cars++;

if (barrierOn) {
        if (cars < threshold) {
                while (stop||!barrierOn) {
                        wait();
                }
        }else{
                stop=false;
                notifyAll();
        }
}

cars--;
if(cars==0){
        stop=true;
}
```

As seen the code is much shorter, and since the function is synchronized it is not needed to do anything specific when increasing or decreasing the cars counter. The problem with the monitor was that when the cars are released, it was hard to stop them when they arrived at the barrier again. This problem is fixed by introducing the boolean stop, so when go is set false, the cars are released. The stop is set false, when car number 9 or threshold comes along. But the stop is set to true, when the last car has been released.

6 Step 5

This step is about how to remove a car and restore it again, not necessarily immediately after. Subproblems of this task is, how to stop the car, how to remove the car, and how to restore the car. A even harder question is, what to do when removing a car inside the alley or waiting at the barrier. These questions are considered during this task.

6.1 Ideas & Design

To stop the car the interrupt function is used, but this means that every time a thread is sleeping or waiting and exception is thrown. Try catch is used to catch these exceptions, and then it has to be calculated what to do when the thread is catch at the specific point in the program. The interrupt stops the car class, but the cars do also have to be removed from the canvas else collisions will be shown. This is done by the CarDisplay clear function but again it is important to notice where in the code the thread is interrupted. Whenever the interrupt exception is catch in the alley or at the barrier, the car counters has to be decreased.

6.2 Implementation

It was noticed that sometimes the cars was not totally cleared from the canvas. This is because the while loop in the run function makes a half move and then another half move, which means that sometimes two half cars can be on two different carfields. It is not always enough to just remove the car from the current position. The CarDisplay has two clear functions. The one clearing a single position and the other clearing two positions. These two functions are used to solve this problem.

6.3 Extra G

7 Tests

The testing of the program has been made during the programming phase. But also some tests have been made in the CarTest-class to show examples of the program running.

8 Evaluation

The project has taught the practical use of semaphores and monitors in java compared to all the theory from the lectures. The use of semaphore should probably be used as the implementation to avoid collision. The semaphores does make a guarantee for the tiles being free, instead of checking the position of all the cars against the current cars position. The positions can be free, but when claiming the tile another faster car might have made the same claim. Unfortunately the problem was discovered very late, which is why another implementation has not been made.

The rest of the project has had small challenges, but the final product works well. The extras made in the project has mostly been parts, which was actually seen as basic functionality. Later what was seen as basic functionality was discovered as an extra assignment, which is the reason for additional extras.
The barrier works as suppose with editable threshold, even though the threshold is changed late in the process. The barrier is seen as working superbly. The same goes for the restore and remove processes, which works like a charm.

The remove can be performed in and out of the alley and also with a restore immediately following.

The overall project gives a good experience working with parallel processes. The result is very satisfying seen with from the group's own perspective.

9 Conclusion

10 Appendix

A Collision avoidance

The code for the cars to avoid collision can be seen below.

```
for (int i = 0; i < 9; i++) {
  if (i != no) {
    //Checking the position of the other cars
    //with the new position
    if (position[i].equals(newpos)) {
      free = false;
    }
    //checking the next position of the other
    //cars with the cars next position
    if (newpos.equals(cd.nextPos(i, position[i]))) {
      if(inAlley){
        if(no<5){
          //Make sure the cars in
          //the alley goes first
          if (no > i) {
            free = false;
          }
        }else{
          if (no < i) {
            free = false;
          }
        }
   }else{
      //If the cars are outside the alley the car
      // with highest number goes first
      if (no < i) {
        free = false;
      }
    }
  }
}
```

B Alley

```
class Alley {

  Semaphore u = new Semaphore(4);
  Semaphore d = new Semaphore(1);
  Semaphore b = new Semaphore(1);
  Semaphore a = new Semaphore(1);
  boolean trafficUp;

  public void enter(int no) throws InterruptedException {
```

```
    if (no < 5) {
      if (trafficUp) {
        if (Integer.parseInt(u.toString()) == 4) {
      try {
        d.P();
      } catch (InterruptedException e) {
        print(no, " has been removed at d.P");
        throw new InterruptedException();
      }
      trafficUp = true;
        }
    try {
      u.P();
    } catch (InterruptedException e) {
      print(no, " has been removed at u.P");
      u.V();
      throw new InterruptedException();
    }

    } else {
      try {
        b.P();
      } catch (InterruptedException e) {
        print(no, " has been removed at b.P");
        throw new InterruptedException();
      }

        if (trafficUp) {
          b.V();
        } else {
          try {
            d.P();
          } catch (InterruptedException e) {
            print(no, " has been removed at d.P");
            b.V();
            throw new InterruptedException();
          }
          b.V();
        }
        try {
          u.P();
        } catch (InterruptedException e) {
          print(no, " has been removed at u.P");
          u.V();
          throw new InterruptedException();
        }
          trafficUp = true;
      }
  } else {
    if (trafficUp) {
```

```
        try {
          d.P();
        } catch (InterruptedException e) {
          print(no, " has been removed at d.P");
          throw new InterruptedException();
        }
        try {
          u.P();
        } catch (InterruptedException e) {
          print(no, " has been removed at u.P");
          u.V();
          throw new InterruptedException();
        }
          trafficUp = false;
      } else {
        if (Integer.parseInt(u.toString()) == 4) {
          try {
            d.P();
          } catch (InterruptedException e) {
            print(no, " has been removed at d.P");
            throw new InterruptedException();
          }
          trafficUp = false;
          }
          try {
            u.P();
          } catch (InterruptedException e) {
            print(no, " has been removed at u.P");
            u.V();
            throw new InterruptedException();
          }
        }
      }
              print(no, " ends entering");
        }

    public void leave(int no)
            throws InterruptedException {

          u.V();

      try{ a.P(); }catch(InterruptedException e){
        throw new InterruptedException();
      }

      if (Integer.parseInt(u.toString()) == 4) {
        if(Integer.parseInt(d.toString())==0){
                  d.V();
        }
      }
```

```
        a.V();
            print(no, " ends leaving");
          }
}
```

C jSpin

The code from jSpin to run the analysis:

```
define N                    8       /* no. of processes */

short c = 0;
short u = 4;
short d = 1;
short b = 1;
bool trafficUp = false;

/* Declare and instantiate N Counter processes */


inline v(s){
        s++;
}

inline p(s){
        atomic{
                s > 0 -> s--;
        }
}

active [N] proctype Alley ()
{
        c=(c+1)%N;
entry:
        if :: c<4 ->
                if :: trafficUp == true ->
                        if :: u == 4 -> p(d);
                                trafficUp = true;
                      :: else -> skip;
                        fi;
                p(u);
                :: trafficUp == false -> p(b);
                        if :: trafficUp == true ->
                                v(b);
                        :: trafficUp == false ->
                                p(d);
                                v(b);
                        fi;
                        p(u);
                        trafficUp=true;
                :: else -> skip;
```

```
                fi;
        :: c>=4 ->
                if :: trafficUp == true ->
                        p(d);
                        p(u);
                        trafficUp = false;
                :: trafficUp ==false; ->
                        if :: u == 4 ->
                                p(d);
                                trafficUp=false;
                        :: else -> skip;
                        fi;
                        p(u);
                fi;
        :: else -> skip;
        fi;

leave:
        v(u);
        atomic{
        if :: u==4 ->
                if :: d== 0 -> v(d);
                :: else -> skip;
                fi;
        :: else -> skip;
        fi;
        }
}

/* Invariant check */
active proctype Check ()
{
        (0 <= c && c <= 7) -> assert(true);
        (0 <= u && u <= 4) -> assert(true);
        (0 <= d && d <= 1) -> assert(true);
        (0 <= b && b <= 1) -> assert(true);
}
```

The analysis run with 5 processes:

```
(Spin Version 6.1.0 -- 4 May 2011)
        + Partial Order Reduction
Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +
State-vector 60 byte, depth reached 72, ... errors: 0 ...
 36981340 states, stored
 57403474 states, matched
 94384814 transitions (= stored+matched)
```

```
 10018950 atomic steps
hash conflicts:  35003377 (resolved)
Stats on memory usage (in Megabytes):
 3103.598       equivalent memory usage for states
 (stored*(State-vector + overhead))
 2015.764       actual memory usage for states (compression: 64.95%)
                state-vector as stored = 29 byte + 28 byte overhead
  256.000       memory used for hash table (-w25)
    0.107       memory used for DFS stack (-m2000)
 2271.064       total actual memory usage
unreached in proctype Alley
        (0 of 91 states)
unreached in proctype Check
        (0 of 9 states)
pan: elapsed time 145 seconds
pan: rate 255784.62 states/second
```

The analysis run with 6 process:

```
pan: resizing hashtable to -w29.. pan: out of memory
hint: to reduce memory, recompile with
  -DCOLLAPSE # good, fast compression, or
  -DMA=68    # better/slower compression, or
  -DHC # hash-compaction, approximation
  -DBITSTATE # supertrace, approximation
(Spin Version 6.1.0 -- 4 May 2011)
Warning: Search not completed
        + Partial Order Reduction
Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +
State-vector 68 byte, depth reached 84, ... errors: 0 ...
 2.69e+08 states, stored
4.5920728e+08 states, matched
7.2820728e+08 transitions (= stored+matched)
 79882859 atomic steps
hash conflicts: 2.7405665e+08 (resolved)
Stats on memory usage (in Megabytes):
24627.686       equivalent memory usage for
states (stored*(State-vector + overhead))
11490.506       actual memory usage for states (compression: 46.66%)
                state-vector as stored = 17 byte + 28 byte overhead
 4096.000       memory used for hash table (-w29)
    0.107       memory used for DFS stack (-m2000)
    4.537       memory lost to fragmentation
15582.076       total actual memory usage
pan: elapsed time 1.16e+03 seconds
pan: rate 231499.41 states/second
```