Chalmers | Göteborgs Universitet

Principles of Concurrent Programming TDA384/DIT392

25 October 2023

Exam supervisor: N. Piterman (piterman@chalmers.se, 073 856 49 10)

(Exam set by N. Piterman based on the course given in Aug-Oct 2023)

Material permitted during the exam (hjälpmedel):
Two textbooks; four sheets of A4 paper with notes (single or double-sided); English dictionary (no smart phones allowed).

Grading: You can score a maximum of 70 points. Exam grades are: between 28–41 (3), between 42–55 (4), 56 or more (5).

Passing the course requires passing the exam and passing the labs. The overall grade for the course is determined as follows: between 40–59 (3), between 60-79 (4), 80 or more (5).

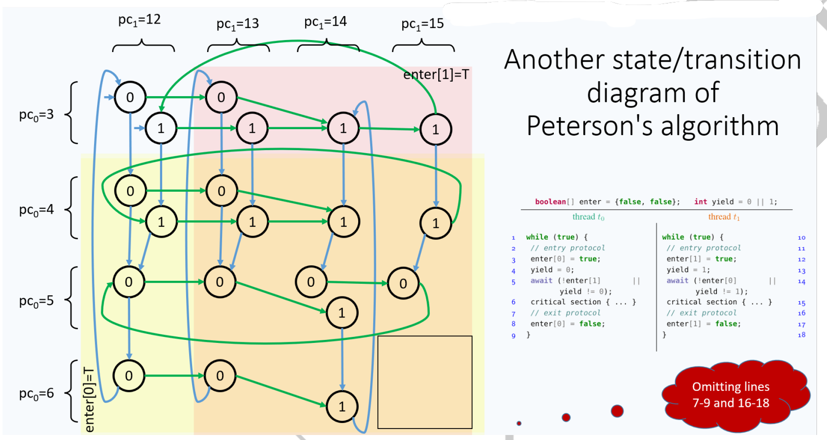
The exam **results** will be available in Ladok within 15 *working* days after the exam's date.

Instructions and rules:

- Please write your answers clearly and legibly: unnecessarily complicated solutions will lose points, and answers that cannot be read will receive no points!
- Justify your answers, and clearly state any assumptions that your solutions may depend on for correctness.
- Answer each question on a new page. Glance through the whole paper first; five questions, numbered Q1 through Q5. Do not spend more time on any question or part than justified by the points it carries.
- Be precise. In your answers, try to use the programming notation and syntax used in the questions. You can also use pseudo-code, *provided* the meaning is precise and clear. If need be, explain your notation.

Q1 (15p). In what follows you will get 5 assertions. For each assertion, you need to say whether it is correct (true) or not (false). You need to justify your answer in each case (an answer without a justification will not be granted full points).

1 The following diagram shows the state-transition diagraom of Peterson's algorithm. The diagram shows that Peterson's algorithm guarantees mutual exclusion. **(3p)**



Another state/transition diagram of Peterson's algorithm

```
 boolean[] enter = {false, false};   int yield = 0 || 1;
 ------------------------------------------------------------
            thread t0                        thread t1

1  while (true) {                  |  while (true) {                10
2   // entry protocol              |   // entry protocol            11
3   enter[0] = true;               |   enter[1] = true;             12
4   yield = 0;                     |   yield = 1;                   13
5   await (!enter[1]          ||   |   await (!enter[0]          || 14
          yield != 0);             |          yield != 1);
6   critical section { ... }       |   critical section { ... }     15
7   // exit protocol               |   // exit protocol             16
8   enter[0] = false;              |   enter[1] = false;            17
9  }                               |  }                             18
```

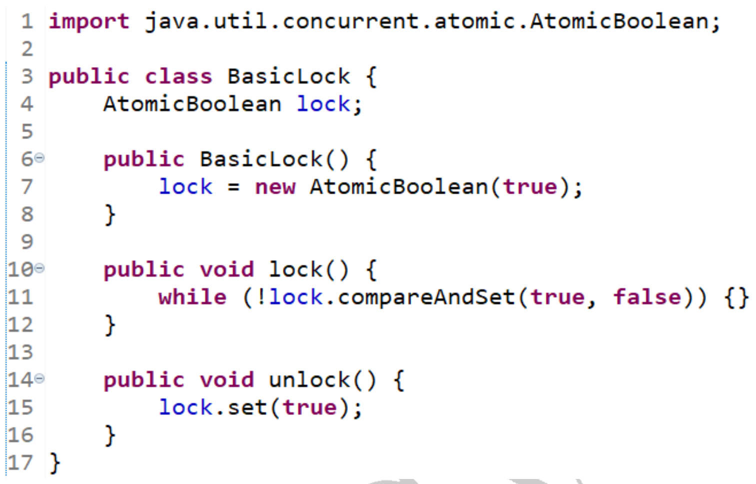
Omitting lines 7-9 and 16-18

Answer: True. States where both threads are in location 6 simultaneously are not reachable.

2 The diagram of Peterson's algorithm shows that it guarantees lack of starvation. **(3p)**

Answer: True. There is no loop when we restrict attention only to the states satisfying $pc_0 = 4$ *or* $pc_0 = 5$. *Similarly, there is no loop when we restrict attention only to the states satisfying* $pc_1 = 13$ *or* $pc_1 = 14$.

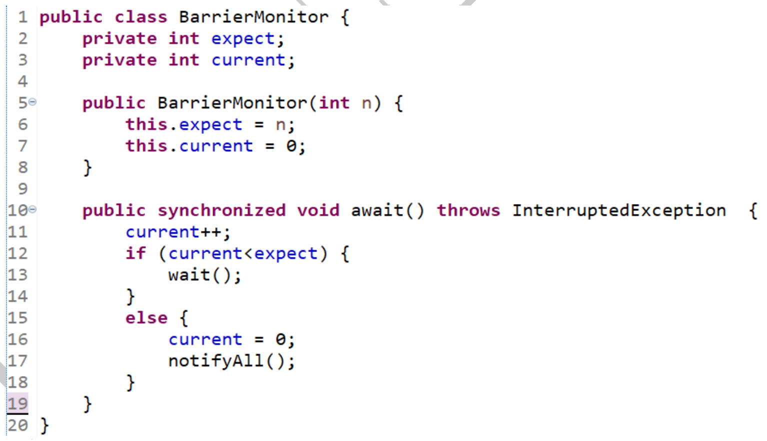
3 A *lock* is a data structure that guarantees (a) exactly one thread acquires the lock and (b) only the thread that acquired the lock can release the lock. Does the following Java code implements a lock? **(3p)**



```java
import java.util.concurrent.atomic.AtomicBoolean;

public class BasicLock {
    AtomicBoolean lock;

    public BasicLock() {
        lock = new AtomicBoolean(true);
    }

    public void lock() {
        while (!lock.compareAndSet(true, false)) {}
    }

    public void unlock() {
        lock.set(true);
    }
}
```

Answer: False. It guarantees (a) above but not (b).

4 The program below is an implementation of a barrier using the Java language based monitor. **(3p)**

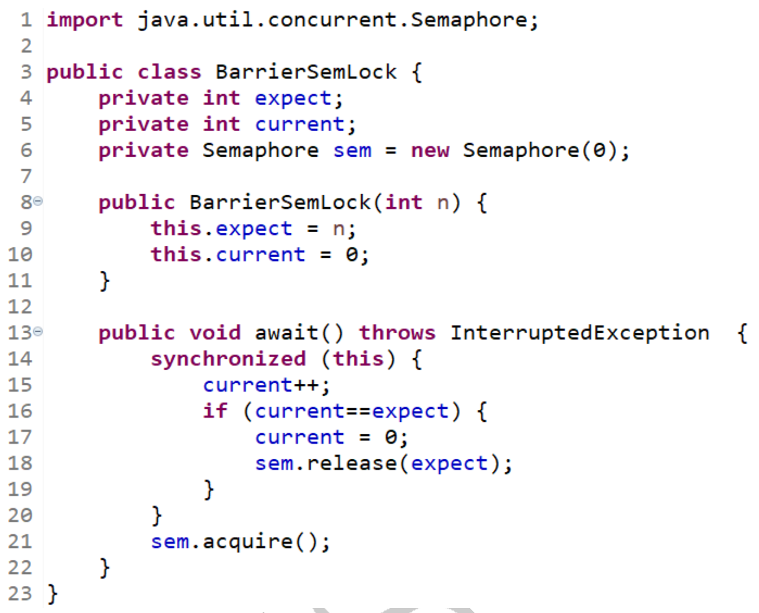


```java
public class BarrierMonitor {
    private int expect;
    private int current;

    public BarrierMonitor(int n) {
        this.expect = n;
        this.current = 0;
    }

    public synchronized void await() throws InterruptedException  {
        current++;
        if (current<expect) {
            wait();
        }
        else {
            current = 0;
            notifyAll();
        }
    }
}
```

Answer: True. The integrity of the number of current waiting threads is ensured by the await method being synchronized. The first expect−1 threads to arrive wait on the condition. The expect thread to arrive, resets the barrier and notifies all the others to continue.

This answer does not take into account spurious wakeups. Answers explaining that this does not work due to spuriour wakeups would also be fully accepted.

5 The program below is an implementation of a barrier using one Semaphore and one Lock. **(3p)**

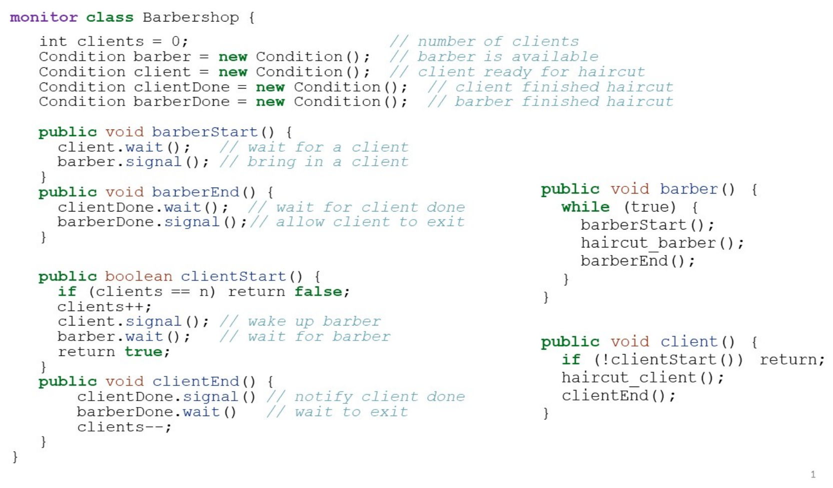


```java
import java.util.concurrent.Semaphore;

public class BarrierSemLock {
    private int expect;
    private int current;
    private Semaphore sem = new Semaphore(0);

    public BarrierSemLock(int n) {
        this.expect = n;
        this.current = 0;
    }

    public void await() throws InterruptedException  {
        synchronized (this) {
            current++;
            if (current==expect) {
                current = 0;
                sem.release(expect);
            }
        }
        sem.acquire();
    }
}
```

Answer: False. For example, one thread could go very fast through the barrier and pass multiple times.

Q2 (18p). A barbershop consists of a waiting room with $n-1$ chairs and the barber chair. Overall, n clients can be in the shop. If there are no customers to be served, the barber sleeps. If a customer opens the doors and sees that there are n clients inside, they leave (even if one client is on the way out). If the barber is busy but chairs are available, the customer sits in a free chair. If the barber is asleep, the customer wakes up the barber. It is OK for the customer to be slow (and pick up their coat) on the way out. We supply three programs that attempt to solve the barbershop problem. One based on monitors and two based on semaphores. You have to analyze these programs and decide whether and how they solve the problem. For every solution: (1) Does the suggested program solve the barbershop problem? (2) Does it promise deadlock freedom? (2) Does it promise lack of starvation? If the solution works, explain why. If the solution does not work, give examples that show this.

(Part a). (6p)



```
monitor class Barbershop {
   int clients = 0;                       // number of clients
   Condition barber = new Condition();   // barber is available
   Condition client = new Condition();   // client ready for haircut
   Condition clientDone = new Condition();  // client finished haircut
   Condition barberDone = new Condition();  // barber finished haircut

   public void barberStart() {
     client.wait();   // wait for a client
     barber.signal(); // bring in a client
   }
   public void barberEnd() {
     clientDone.wait();  // wait for client done
     barberDone.signal();// allow client to exit
   }

   public boolean clientStart() {
     if (clients == n) return false;
     clients++;
     client.signal(); // wake up barber
     barber.wait();   // wait for barber
     return true;
   }
   public void clientEnd() {
       clientDone.signal() // notify client done
       barberDone.wait()   // wait to exit
       clients--;
   }
}
```

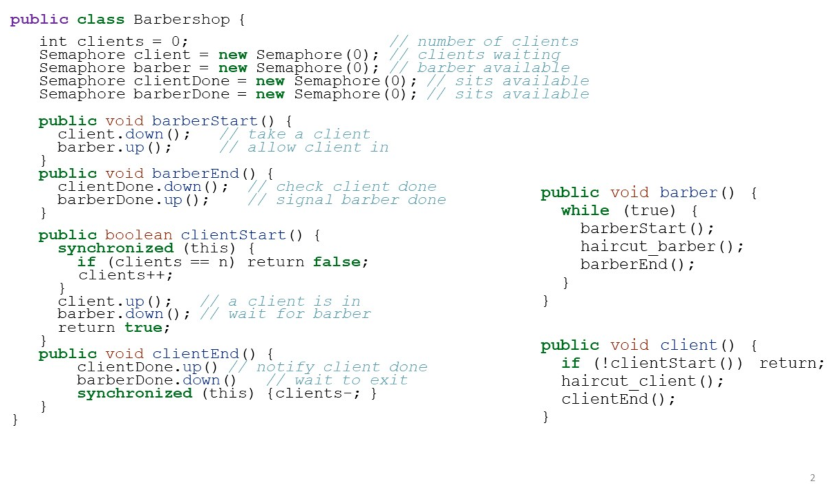
```
public void barber() {
  while (true) {
    barberStart();
    haircut_barber();
    barberEnd();
  }
}

public void client() {
  if (!clientStart()) return;
  haircut_client();
  clientEnd();
}
```

1

Check the monitor based solution above. The code run by the barber and by clients is given on the right hand side. They both call functions of the same monitor instance.

Answer: The implementation does not *solve the barbershop problem. For example, the barber starts, waits for the first client. The fisrt client comes and signals to the barber and waits for the barber. The barber wakes up (either because the signalling policy is signal and wait or because there is no one else around) and signals back to the client. The client comes in (either because the signalling policy is signal and wait or because there is no one else around). Now a second client comes, signals to the barber and waits for the barber. The barber and the first* *client finish, the first client goes out, the barber waits for a client. But the signal that the client sent when they entered has already been sent. This is a deadlock.*

(Part b). (6p)



```
public class Barbershop {
   int clients = 0;                                 // number of clients
   Semaphore client = new Semaphore(0); // clients waiting
   Semaphore barber = new Semaphore(0); // barber available
   Semaphore clientDone = new Semaphore(0); // sits available
   Semaphore barberDone = new Semaphore(0); // sits available

   public void barberStart() {
     client.down();    // take a client
     barber.up();      // allow client in
   }
   public void barberEnd() {
     clientDone.down();  // check client done
     barberDone.up();    // signal barber done
   }

   public boolean clientStart() {
     synchronized (this) {
       if (clients == n) return false;
       clients++;
     }
     client.up();    // a client is in
     barber.down(); // wait for barber
     return true;
   }
   public void clientEnd() {
       clientDone.up() // notify client done
       barberDone.down()    // wait to exit
       synchronized (this) {clients-; }
   }
}
```

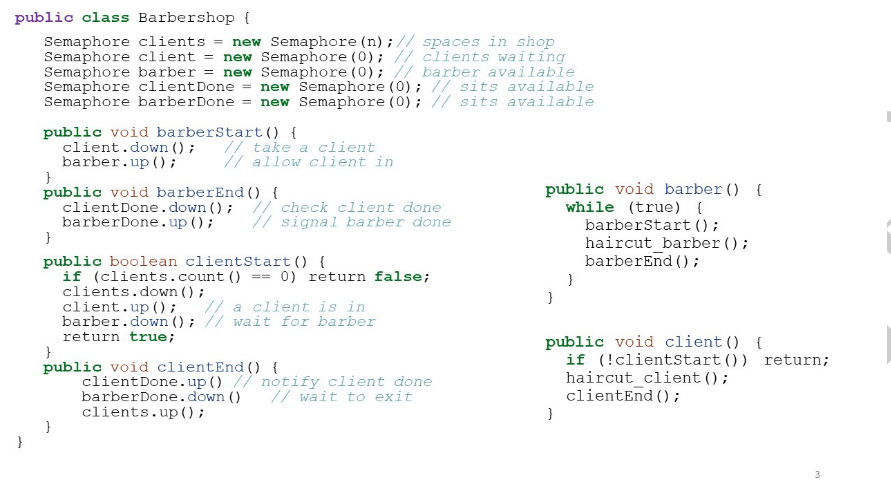
```
public void barber() {
  while (true) {
    barberStart();
    haircut_barber();
    barberEnd();
  }
}


public void client() {
  if (!clientStart()) return;
  haircut_client();
  clientEnd();
}
```

2

Check the semaphore based solution above. The code run by the barber and by clients is given on the right hand side. They both call functions of the same barbershop instance.

Answer: This implementation solves the barbershop problem. The client semaphore holds the number of clients in the shop minus the one on the barber sit. The barber semaphore is a binary semaphore signaling when the barber is free. The barberDone and clientDone semaphores are binary semaphores behaving like a barrier at the exit from the shop. The correctness of the client integer variables is maintained by synchronizing the access to it. When there are no clients, the barber waits on the client semaphore. When a client enters the shop they increase the client semaphore (waking up the barber if they are asleep). Once the barber picked a ticket from the client semaphore, they signal their availability by increasing the barber semaphore. One of the clients waiting on barber will pick up the barber. They both go for the haircut. On leaving, the client says that they are done, the barber picks it up and confirms that they are done as well and then the client can leave, reducing the number of customers upon going out. If the `barber` *semaphore is fair, then the solution offers lack of starvation.*

(Part c). (6p)



```
public class Barbershop {

   Semaphore clients = new Semaphore(n);// spaces in shop
   Semaphore client = new Semaphore(0); // clients waiting
   Semaphore barber = new Semaphore(0); // barber available
   Semaphore clientDone = new Semaphore(0); // sits available
   Semaphore barberDone = new Semaphore(0); // sits available

   public void barberStart() {
     client.down();    // take a client
     barber.up();      // allow client in
   }
   public void barberEnd() {
     clientDone.down();  // check client done
     barberDone.up();    // signal barber done
   }

   public boolean clientStart() {
     if (clients.count() == 0) return false;
     clients.down();
     client.up();   // a client is in
     barber.down(); // wait for barber
     return true;
   }
   public void clientEnd() {
       clientDone.up() // notify client done
       barberDone.down()   // wait to exit
       clients.up();
   }
}
```

```
public void barber() {
  while (true) {
    barberStart();
    haircut_barber();
    barberEnd();
  }
}

public void client() {
  if (!clientStart()) return;
  haircut_client();
  clientEnd();
}
```

Check the semaphore based solution above. The code run by the barber and by clients is given on the right hand side. They both call functions of the same instance of the barbershop class.

Answer: The implementation does not *solve the barbershop problem. There could be multiple clients that check if* `clients.count()==0` *at the same time. All of them proceeding to the* `clients.down()`*, effectively waiting inside the shop while no sits are available.*

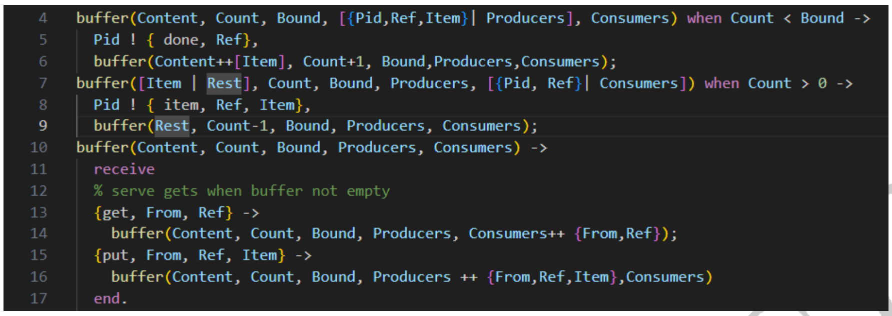
Q3 (10p). The following code shows the Erlang solution of the producer-consumer problem that was shown in class. It includes the server loop (`buffer`) and the two service functions for producers (`put`) and for consumers (`get`).

```
buffer(Content, Count, Bound) ->
  receive
    % serve gets when buffer not empty
    {get, From, Ref} when Count > 0 ->
      [First|Rest] = Content,     % match first item
      From ! {item, Ref, First}, % send it out
      buffer(Rest, Count-1, Bound);     % remove it from buffer
    % serve puts when buffer not full
    {put, From, Ref, Item} when Count < Bound ->
      From ! {done, Ref},              % send ack
      buffer(Content ++ [Item], Count+1, Bound) % add item to end
  end.
```

```
% get item from 'Buffer'; block if empty
get(Buffer) ->
   Ref = make_ref(),
   Buffer ! {get, self(), Ref},
   receive {item, Ref, Item} -> Item end.


% put 'Item' in 'Buffer'; block if full
put(Buffer, Item) ->
   Ref = make_ref(),
   Buffer ! {put, self(), Ref, Item},
   receive {done, Ref} -> done end.
```

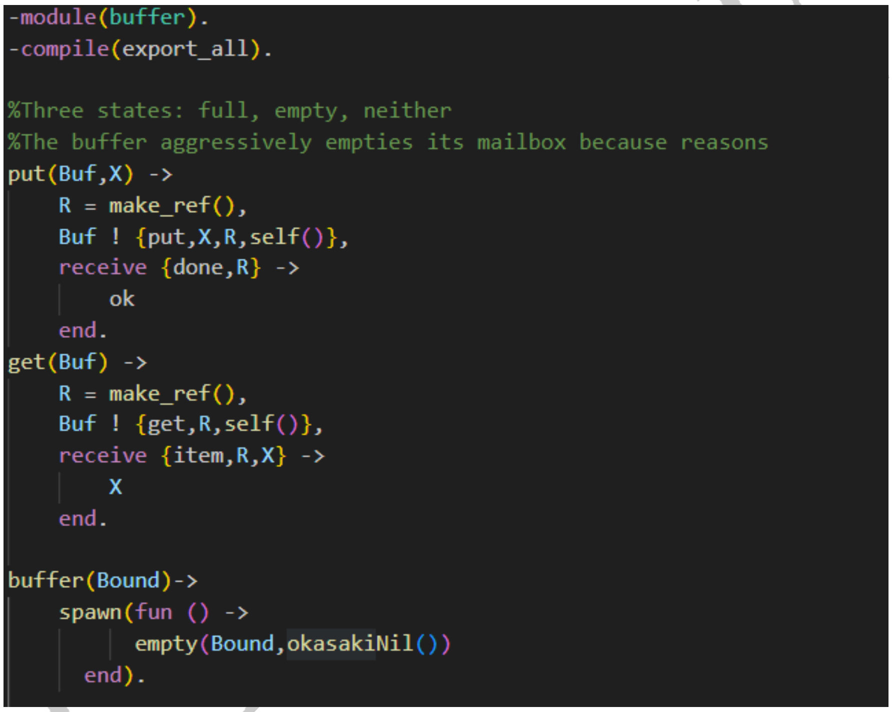
This implementation strongly relies on the mailbox of the process running the server in order. Change the server loop so that it does not rely on the mailbox for managing the queue of those waiting to produce or to consume. Namely, add extra memory to the state of the server so that it manages the queue of those that are waiting to produce or to consume (**2pt**). Change the server loop so that those waiting in your new queues are served in FIFO order (**4pt** producers; **4pt** consumers).

Answer: Here is a first simple solution:

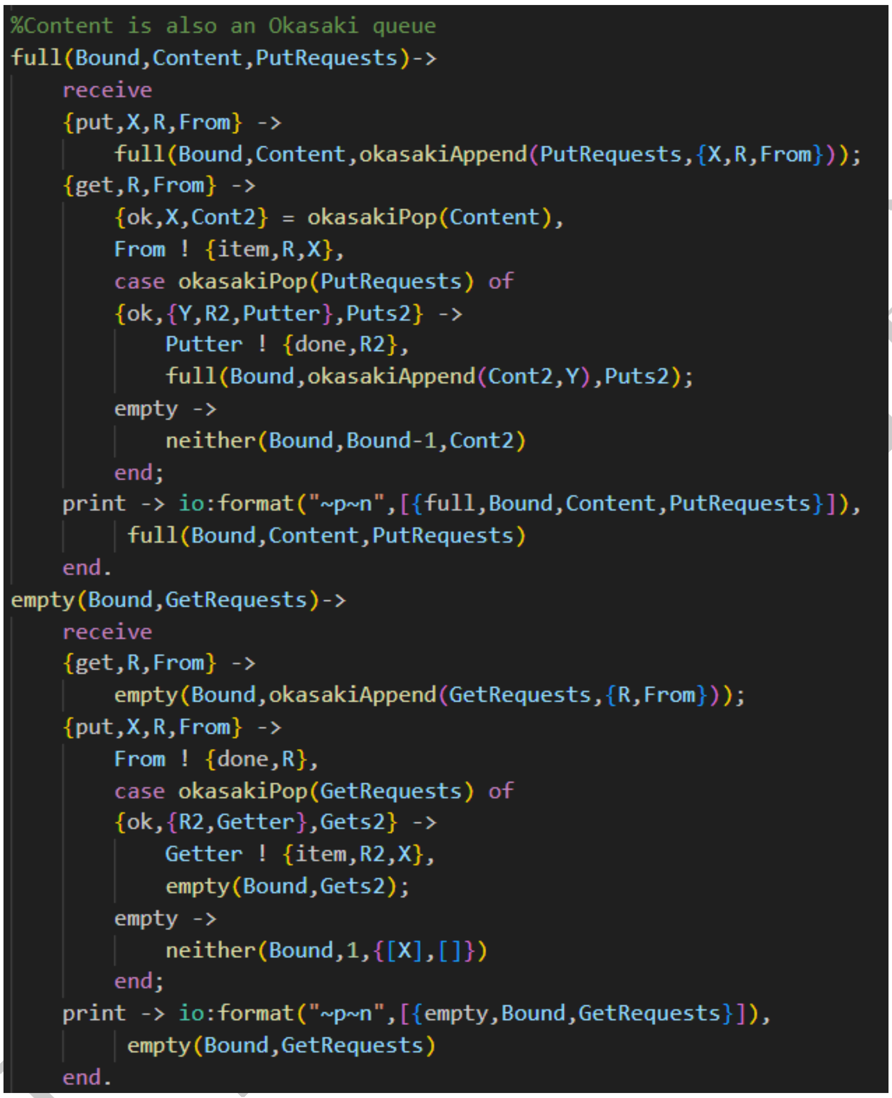


```erlang
buffer(Content, Count, Bound, [{Pid,Ref,Item}| Producers], Consumers) when Count < Bound ->
  Pid ! { done, Ref},
  buffer(Content++[Item], Count+1, Bound,Producers,Consumers);
buffer([Item | Rest], Count, Bound, Producers, [{Pid, Ref}| Consumers]) when Count > 0 ->
  Pid ! { item, Ref, Item},
  buffer(Rest, Count-1, Bound, Producers, Consumers);
buffer(Content, Count, Bound, Producers, Consumers) ->
  receive
  % serve gets when buffer not empty
  {get, From, Ref} ->
    buffer(Content, Count, Bound, Producers, Consumers++ {From,Ref});
  {put, From, Ref, Item} ->
    buffer(Content, Count, Bound, Producers ++ {From,Ref,Item},Consumers)
  end.
```

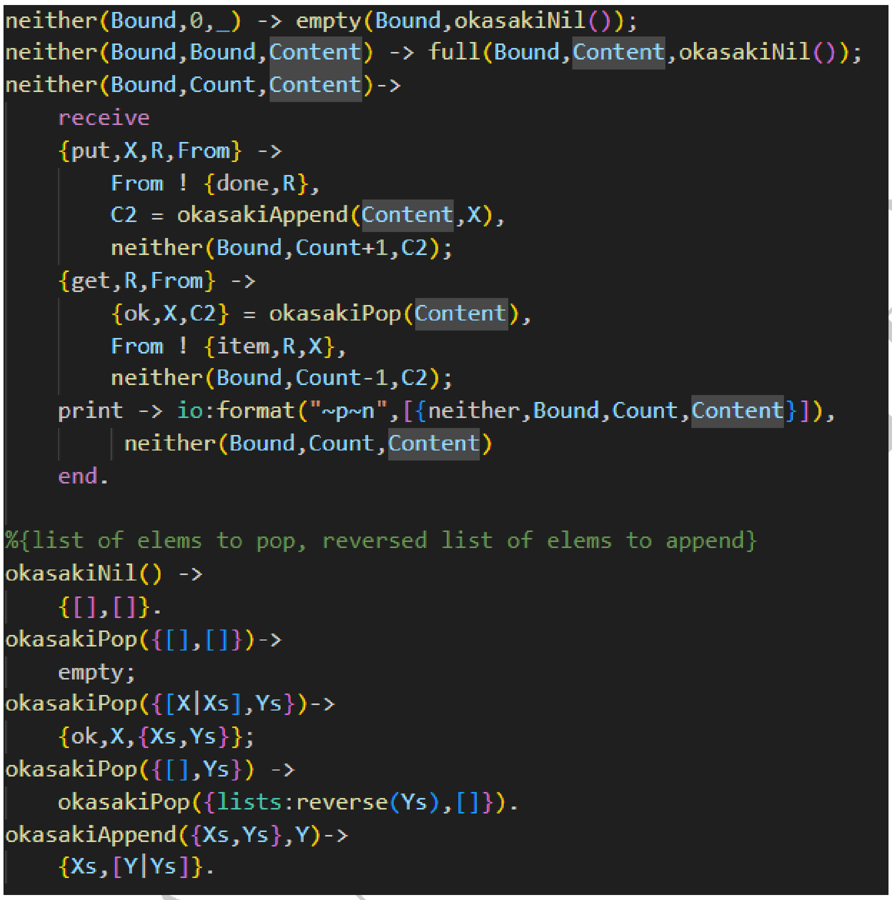
And a solution using an Okasaki queue that is more efficient:



```erlang
-module(buffer).
-compile(export_all).

%Three states: full, empty, neither
%The buffer aggressively empties its mailbox because reasons
put(Buf,X) ->
    R = make_ref(),
    Buf ! {put,X,R,self()},
    receive {done,R} ->
        ok
    end.
get(Buf) ->
    R = make_ref(),
    Buf ! {get,R,self()},
    receive {item,R,X} ->
        X
    end.

buffer(Bound)->
    spawn(fun () ->
        empty(Bound,okasakiNil())
      end).
```

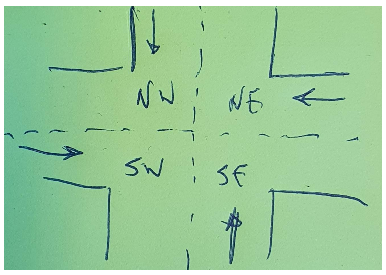


```erlang
%Content is also an Okasaki queue
full(Bound,Content,PutRequests)->
    receive
    {put,X,R,From} ->
        full(Bound,Content,okasakiAppend(PutRequests,{X,R,From}));
    {get,R,From} ->
        {ok,X,Cont2} = okasakiPop(Content),
        From ! {item,R,X},
        case okasakiPop(PutRequests) of
        {ok,{Y,R2,Putter},Puts2} ->
            Putter ! {done,R2},
            full(Bound,okasakiAppend(Cont2,Y),Puts2);
        empty ->
            neither(Bound,Bound-1,Cont2)
        end;
    print -> io:format("~p~n",[{full,Bound,Content,PutRequests}]),
         full(Bound,Content,PutRequests)
    end.
empty(Bound,GetRequests)->
    receive
    {get,R,From} ->
        empty(Bound,okasakiAppend(GetRequests,{R,From}));
    {put,X,R,From} ->
        From ! {done,R},
        case okasakiPop(GetRequests) of
        {ok,{R2,Getter},Gets2} ->
            Getter ! {item,R2,X},
            empty(Bound,Gets2);
        empty ->
            neither(Bound,1,{[X],[]})
        end;
    print -> io:format("~p~n",[{empty,Bound,GetRequests}]),
         empty(Bound,GetRequests)
    end.
```



```erlang
neither(Bound,0,_) -> empty(Bound,okasakiNil());
neither(Bound,Bound,Content) -> full(Bound,Content,okasakiNil());
neither(Bound,Count,Content)->
    receive
    {put,X,R,From} ->
        From ! {done,R},
        C2 = okasakiAppend(Content,X),
        neither(Bound,Count+1,C2);
    {get,R,From} ->
        {ok,X,C2} = okasakiPop(Content),
        From ! {item,R,X},
        neither(Bound,Count-1,C2);
    print -> io:format("~p~n",[{neither,Bound,Count,Content}]),
         neither(Bound,Count,Content)
    end.

%{list of elems to pop, reversed list of elems to append}
okasakiNil() ->
    {[],[]}.
okasakiPop({[],[]})->
    empty;
okasakiPop({[X|Xs],Ys})->
    {ok,X,{Xs,Ys}};
okasakiPop({[],Ys}) ->
    okasakiPop({lists:reverse(Ys),[]}).
okasakiAppend({Xs,Ys},Y)->
    {Xs,[Y|Ys]}.
```

Q4 (15p). You are trying to design a junction to be used by automated vehicles (in Sweden, so driving on the rhs). The junction is partitioned to four areas and a lock is associated with each of these areas:



Here is an initial implementation:

```
public class Junction {
  private Lock ne = new ReentrantLock();
  private Lock se = new ReentrantLock();
  private Lock nw = new ReentrantLock();
  private Lock sw = new ReentrantLock();

  void approachFromNW() { nw.lock(); sw.lock(); }
  void leaveFromSW() { nw.unlock(); sw.unlock(); }

  void approachFromSE() { se.lock(); ne.lock(); }
  void leaveFromNE() { se.unlock(); ne.unlock(); }

  void approachFromEN() { ne.lock(); nw.lock(); }
  void leaveFromWN() { ne.unlock(); nw.unlock(); }

  void approachFromWS() { sw.lock(); se.lock(); }
  void leaveFromES() { sw.unlock(); se.unlock(); }

  ...
}
```

(Part a). (3p)

Suppose that vehicles approaching the junction call the approach function and upon leaving the junction call the appropriate leave function. Does this ensure lack of collisions between cars?

Answer: Yes. In order to cross a location that is in the path of another vehicle, a vehicle would have to lock the appropriate lock first.

(Part b). (3p)

Does the program ensure that all cars wanting to enter the junction will eventually do so?

Answer: No. If four vehicles approach the junction at the same time from the four directions and acquire one lock each, the system can deadlock.

(Part c). (3p) Based on your insights from (a) and (b), change the program so that rather than continuing straight cars would turn left. For example, on approach from south east a car would pass through the SE, then NE, then NW of the junction. Your solution should ensure lack of collisions and that every car that wants to enter the junction would do so eventually.

Answer: Fix an order on the locks (NE,NW,SE,SW) and all approaches would acquire the locks according to this order. For example:

```
void approachFromSE() { ne.lock(); nw.lock(); se.lock(); }
void leaveFromNE() { ne.unlock(); nw.unlock(); se.unlock(); }
```

(Part d). (3p) How would your answer change if entrances from the east and west are blocked so that no cars arrive from the east and the west. Furthermore, cars from the south are now allowed to either continue north or turn left (exiting from west; doing the trajectory SE, NE, NW). Cars from the north continue, as before, south. You may assume that cars coming from the same direction do not collide with one another (even if they have to stop).

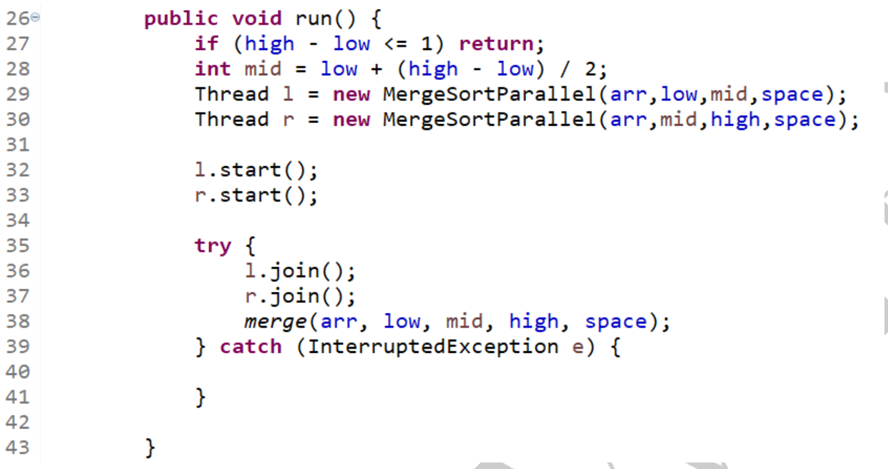
Answer: It is enough to use only the NW lock for cars coming from the north and cars coming from the south and turning left. Cars coming from the south and continuing straight do not have to use locking.

(Part e). (3p) Consider the case that multiple cars are allowed to be in the same region of the junction. That is, the junction is actually much larger with multiple lanes going in every direction but still only partitioned to four regions. Now multiple cars would be coming from south and turning left simultaneously. What coordination problem would that be similar to and why?

Answer: Readers-writers. Readers and writer access would still be mutually exclusive but it would be allowed for multiple writers to be in simultaneously.

Q5 (12p). (Part a). (4p)

Here is the first parallel implementation of merge sort shown in class:



```java
        public void run() {
            if (high - low <= 1) return;
            int mid = low + (high - low) / 2;
            Thread l = new MergeSortParallel(arr,low,mid,space);
            Thread r = new MergeSortParallel(arr,mid,high,space);

            l.start();
            r.start();

            try {
                l.join();
                r.join();
                merge(arr, low, mid, high, space);
            } catch (InterruptedException e) {

            }

        }
```

How many threads would be involved in the sorting of an array with 512 entries? What is the maximal number of threads that could be running merge simultaneously?

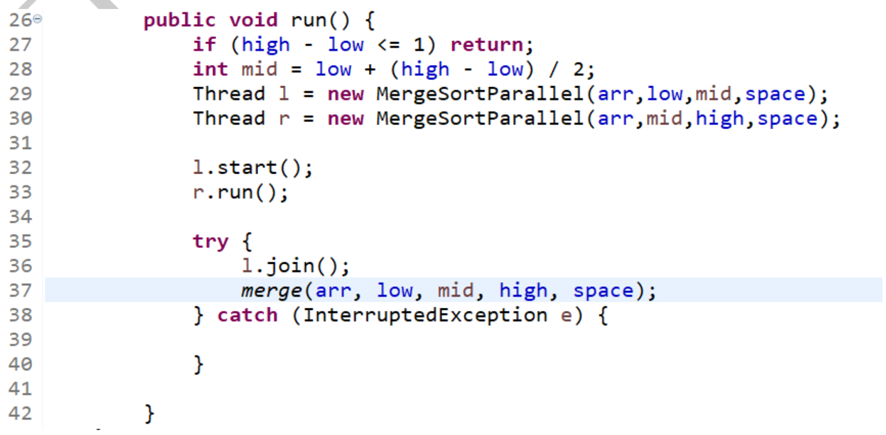
Answer: There will be 1023 threads involved.

$$\sum_{i=0}^{9} 2^i = 2^{10} - 1 = 1023$$

The maximal number of threads that could be running merge simultaneously is 256. This happens in the case that 256 threads handling arrays of length 2 merge the results of the two threads spawned by them.

(Part b). (4p)

We noted that the spawning of threads can be improved by allowing the spawning thread to do some actions:



```java
        public void run() {
            if (high - low <= 1) return;
            int mid = low + (high - low) / 2;
            Thread l = new MergeSortParallel(arr,low,mid,space);
            Thread r = new MergeSortParallel(arr,mid,high,space);

            l.start();
            r.run();

            try {
                l.join();
                merge(arr, low, mid, high, space);
            } catch (InterruptedException e) {

            }

        }
```

How many threads would be involved in the sorting of an array with 512 entries? What is the maximal number of threads that could be running merge simultaneously?

Answer: There will be 512 threads involved. Every thread now handles a unique section of the array of length 1. The maximal number of threads that could be running merge simultaneously does not change.

(Part c). (4p) Discuss the advantages and disadvantages of running the above parallel version of merge sort on a machine with 8 cores vs using a ForkJoinPool or ExecutorService for this task.

Answer: Just using threads is simple to implement. The main optimization that is requird for sorting such a small array is probably just to allow a larger part of the sort to be done sequentially.

On the other hand, threads will be competing with each other for resources adding time would be spent on swapping in and out of memory.

Using a pool, swapping of threads will not happen. It still uses quite complex concepts for doing something quite simple.