

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).

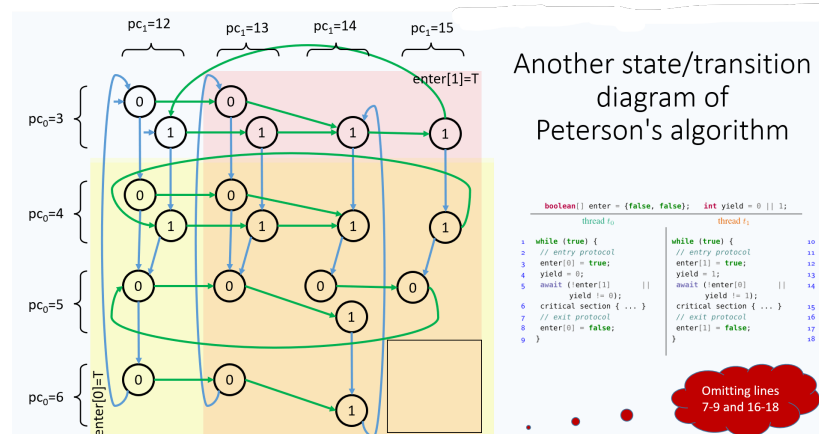
The exam **results** will be available in Ladok within 15 *working* days after the exam's date.

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 diagram of Peterson's algorithm. The diagram shows that Peterson's algorithm guarantees mutual exclusion. **(3p)**



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

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)**

```

1 import java.util.concurrent.atomic.AtomicBoolean;
2
3 public class BasicLock {
4     AtomicBoolean lock;
5
6     public BasicLock() {
7         lock = new AtomicBoolean(true);
8     }
9
10    public void lock() {
11        while (!lock.compareAndSet(true, false)) {}
12    }
13
14    public void unlock() {
15        lock.set(true);
16    }
17 }

```

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

```
1 public class BarrierMonitor {
2     private int expect;
3     private int current;
4
5     public BarrierMonitor(int n) {
6         this.expect = n;
7         this.current = 0;
8     }
9
10    public synchronized void await() throws InterruptedException {
11        current++;
12        if (current < expect) {
13            wait();
14        }
15        else {
16            current = 0;
17            notifyAll();
18        }
19    }
20 }
```

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

```
1 import java.util.concurrent.Semaphore;
2
3 public class BarrierSemLock {
4     private int expect;
5     private int current;
6     private Semaphore sem = new Semaphore(0);
7
8     public BarrierSemLock(int n) {
9         this.expect = n;
10        this.current = 0;
11    }
12
13    public void await() throws InterruptedException {
14        synchronized (this) {
15            current++;
16            if (current == expect) {
17                current = 0;
18                sem.release(expect);
19            }
20        }
21        sem.acquire();
22    }
23 }
```

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;
    Condition barber = new Condition(); // number of clients
    Condition client = new Condition(); // barber is available
    Condition clientDone = new Condition(); // client ready for haircut
    Condition barberDone = 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();
}

```

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.

(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.

(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();
}
```

3

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.

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.

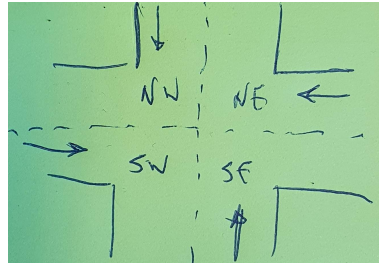
% get item from 'Buffer'; block if empty
get(Buffer) ->
  Ref = make-ref(),
  Buffer ! {get, self(), Ref},
  receive {item, Ref, Item} -> Item end.

% 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).

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?

(Part b). (3p)

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

(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.

(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).

(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?

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

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

```
26Ⓢ      public void run() {
27          if (high - low <= 1) return;
28          int mid = low + (high - low) / 2;
29          Thread l = new MergeSortParallel(arr,low,mid,space);
30          Thread r = new MergeSortParallel(arr,mid,high,space);
31
32          l.start();
33          r.start();
34
35          try {
36              l.join();
37              r.join();
38              merge(arr, low, mid, high, space);
39          } catch (InterruptedException e) {
40
41          }
42
43      }
```

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?

(Part b). (4p)

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

```
26Ⓢ      public void run() {
27          if (high - low <= 1) return;
28          int mid = low + (high - low) / 2;
29          Thread l = new MergeSortParallel(arr,low,mid,space);
30          Thread r = new MergeSortParallel(arr,mid,high,space);
31
32          l.start();
33          r.run();
34
35          try {
36              l.join();
37              merge(arr, low, mid, high, space);
38          } catch (InterruptedException e) {
39
40          }
41
42      }
```

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?

(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.