Assignment 5: Monitors

ETH Zurich

1 Queues

1.1 Background

There is a given generic queue class called *Queue*<*T*>, where T is the generic type of the queue elements. You only know that *Queue*<*T*> follows FIFO rules on a single-threaded execution, and offers the methods *enqueue*, *dequeue*, and *size*. No assumptions can be made about the thread safety of these operations.

1.2 Tasks

1. Implement a bounded concurrent queue using *Queue* in Java or a suitable pseudocode. The following operations must be implemented:

   - *void enqueue*(*T v*), which enqueues the value *v* on the queue.
   - *T dequeue*(), which dequeues a value and returns it to the caller.
   - A constructor which takes the queue bound (> 0) as an argument.

   You are to implement this using a signal-and-continue monitor and two condition variables, one condition variable for "not empty" and one for "not full". These condition variables only provide two operations: *signal* and *wait*. Recall that *signal* only awakens a single thread.

2. Imagine in the previous situation that a single condition variable is used for both the "not empty" and "not full" conditions. With a single condition variable, can you guarantee that a waiting enqueue (dequeue) operation is only awakened when the queue is not full (empty)?

   If yes, how? If not, what problem does this pose when only *signal* is available?

1.3 Solution

1.3.1 Task 1

Solution in a sort of pseudo-Java (which allows ConditionVariables to be attached to the surrounding monitor instead of a particular Lock).

```
class ConcQueue<T> {
Queue<T> q;

ConditionVariable not_empty;
ConditionVariable not_full;
int bound;

ConcQueue(int bound)
  {
    this.q         = new Queue<T>();
    this.not_empty = new ConditionVariable();
    this.not_full  = new ConditionVariable();
    this.bound     = bound;
  }

synchronized
enqueue(T v)
  {
    while (q.size() == bound)
      not_full.wait();

    q.enqueue(v);

    not_empty.signal();
  }

synchronized
T dequeue()
  {
    T result;

    while (q.size() == 0)
      not_empty.wait();

    result = q.dequeue();

    not_full.signal();

    return result;
  }
}
```

1.3.2 Task 2

No, a single condition variable cannot distinguish between different semantic conditions.

This poses the problem that a signal could be "lost" by it waking up a thread that didn't require that condition to be true instead of a thread that did require that condition to be true.

2 Signal and continue vs. signal and wait

2.1 Background

Listing 1 shows a monitor class that defines three parts of a job.

Listing 1: three part job class with signal and wait

```
monitor class THREE_PART_JOB

feature
  first_part_done : CONDITION_VARIABLE

   do_first_and_third_part
    do
       first_part
       first_part_done . signal    -- ''Signal and Wait'' signaling  discipline
       third_part
    end

  do_second_part
    do
       first_part_done . wait
      second_part
    end
end
```

The condition variable *first_part_done* is used to ensure that the first and the third part are executed by one thread t_1 and that the second part is executed by another thread t_2 in between the first and the third part. This is the correctness specification.

2.2 Task

1. Assume that the condition variable implements the "Signal and Wait" discipline. Is the code correct? If the code is correct, justify why it works. If the code is not correct, show a sequence of actions that illustrates the problem.
2. Assume now that the condition variable implements the "Signal and Continue" discipline instead. Is the code correct? If the code is correct, justify why it works. If the code is not correct, show a sequence of actions that illustrates the problem.
3. If the program is not correct with the "Signal and Continue" discipline, rewrite the program so that it is correct. To do this, use the "Signal and Continue" condition variables.

2.3 Solution

1. The code is not correct. It works if t_2 gets the monitor first. If t_1 gets the monitor first, then t_1 proceeds without synchronization. Once t_2 gets the monitor, it blocks and ends up in a deadlock.

2. The code is not correct. If t_1 gets the monitor first, then t_1 proceeds without synchronization. Once t_2 gets the monitor, it blocks and ends up in a deadlock. If t_2 gets the monitor first, then t_2 blocks and lets t_1 proceeds without synchronization; only after t_1 is done will t_2 continue.

3. The following code reproduces the correct behavior with the "Signal and Continue" signaling discipline:

Listing 2: three part job class with signal and continue

```
monitor class THREE_PART_JOB

feature
    first_part_done : CONDITION_VARIABLE
   monitor_returned: CONDITION_VARIABLE
    entered_first : BOOLEAN  -- Initially set to 'False'

    do_first_and_third_part
      do
          first_part
          first_part_done . signal    -- ''Signal and Continue'' signaling discipline
          entered_first  := True
         monitor_returned.wait
          third_part
      end

   do_second_part
      do
         if not entered_first then
            first_part_done . wait
         end
         second_part
         monitor_returned.signal    -- ''Signal and Continue'' signaling discipline
      end
end
```

3 Deadlocks

3.1 Background

We have a monitor class:

```
monitor class BAZ
  c: CONDITION_VARIABLE
  e: CONDITION_VARIABLE
  i: INTEGER
  x: INTEGER

  foo
    do
       if i < 5 then
         c.wait
       end
```

```
      x := i * 2
      e.signal
      x := 10 - x
    end

  bar
    do
      i := 5
      c.signal
      i := i + 1
      e.wait
      x := 8 - x
    end
end
```

3.2 Task

For the class given above, a thread will run *foo* and another will run *bar*. These executions occur concurrently. Assume that i is initialized to 0.

Consider the execution with both the signal-and-wait, and signal-and-continue signaling disciplines. For each signaling discipline, state and justify:

- whether the program will deadlock.
- if the program does not deadlock, what the value of x is at termination.

3.3 Solution

For signal-and-wait:

- *foo* is called first: deadlock.
- *bar* is called first: result is 14.

For signal-and-continue:

- *foo* is called first: result is 10.
- *bar* is called first: result is 10.