Assignment 4: 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 Master 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 A barrier with a monitor

2.1 Background

A barrier is a form of synchronization that determines a point in the execution of a program which all threads in a group have to reach before any of them may move on.

2.2 Task

1. Develop a monitor class that implements a barrier for n threads using the signal-and-continue signaling discipline. Your monitor class should have a feature *join*; to join a barrier a thread can call *barrier.join*. You may assume that the threads are numbered from 1 to n and that the identifier of the current thread can be queried with *current_thread .id*.
2. What difference does it make if your solution uses the signal-and-wait signaling discipline? Give two execution sequences, one for each signaling discipline, to show the difference. In particular, mention when threads acquire/release the lock on the monitor and mention when the threads enter/leave the queues of the condition variables and the monitor. You can assume that there are only three threads, i.e. $n = 3$.

   For example, an execution sequence could look like this:

   (a) Thread 1: Acquire the lock on the monitor.
   (b) Thread 1: Release the lock on the monitor.
   (c) Thread 1: Continue with the post-synchronization workload.
   (d) ...

2.3 Solution

```
monitor class BARRIER

create
  make

feature -- Initialization
  make (a_number_of_threads)
      -- Create the barrier with 'a_number_of_threads'.
    do
      number_of_threads := a_number_of_threads
      occupancy := 0
      create is_full.make (1, number_of_threads)
    end

feature -- Basic operations
  join
      -- Join the barrier.
    do
      if occupancy + 1 < number_of_threads then
```

```
          occupancy := occupancy + 1
          is_full [current_thread.id].wait
        else
          across is_full as l_cursor loop
            if not l_cursor.item.is_empty then
              l_cursor.item.signal
            end
          end
        end
      end

feature {NONE} -- Implementation
  number_of_threads: INTEGER -- The number of threads.
  occupancy: INTEGER -- How many threads have joined the barrier?
  is_full : ARRAY [CONDITION_VARIABLE] -- An array with one condition variable for
        each thread. Each joining thread that does not complete the barrier uses its condition
        variable to wait for the barrier to become full. The last thread uses the condition
        variables to signal the other threads that the barrier is now full.
end
```

The signal-and-continue discipline permits the following execution:

1. Thread 1: Add thread 1 to the queue of the monitor. Acquire the lock on the monitor. Remove thread 1 from the queue of the monitor.
2. Thread 1: Set occupancy to 1.
3. Thread 1: Release the lock on the monitor. Add thread 1 to the queue of its condition variable.
4. Thread 2: Add thread 2 to the queue of the monitor. Acquire the lock on the monitor. Remove thread 2 from the queue of the monitor.
5. Thread 2: Set occupancy to 2.
6. Thread 2: Release the lock on the monitor. Add thread 2 to the queue of its condition variable.
7. Thread 3: Add thread 3 to the queue of the monitor. Acquire the lock on the monitor. Remove thread 3 from the queue of the monitor. All threads have reached the barrier.
8. Thread 3: Remove thread 1 from the queue of its condition variable. Add thread 1 to the queue of the monitor.
9. Thread 3: Remove thread 2 from the queue of its condition variable. Add thread 2 to the queue of the monitor.
10. Thread 3: Release the lock on the monitor.
11. Thread 3: Continue with the post-synchronization workload.
12. Thread 1: Acquire the lock on the monitor. Remove thread 1 from the queue of the monitor.
13. Thread 1: Release the lock on the monitor.
14. Thread 1: Continue with the post-synchronization workload.

15. Thread 2: Acquire the lock on the monitor. Remove thread 2 from the queue of the monitor.
16. Thread 2: Release the lock on the monitor.
17. Thread 2: Continue with the post-synchronization workload.

The signal-and-wait discipline permits the following execution:

1. Thread 1: Add thread 1 to the queue of the monitor. Acquire the lock on the monitor. Remove thread 1 from the queue of the monitor.
2. Thread 1: Set occupancy to 1.
3. Thread 1: Release the lock on the monitor. Add thread 1 to the queue of its condition variable.
4. Thread 2: Add thread 2 to the queue of the monitor. Acquire the lock on the monitor. Remove thread 2 from the queue of the monitor.
5. Thread 2: Set occupancy to 2.
6. Thread 2: Release the lock on the monitor. Add thread 2 to the queue of its condition variable.
7. Thread 3: Add thread 3 to the queue of the monitor. Acquire the lock on the monitor. Remove thread 3 from the queue of the monitor. All threads have reached the barrier.
8. Thread 3: Add thread 3 to the queue of the monitor. Remove thread 1 from the queue of its condition variable. Transfer the lock on the monitor to thread 1.
9. Thread 1: Release the lock on the monitor.
10. Thread 1: Continue with the post-synchronization workload.
11. Thread 3: Acquire the lock on the monitor.
12. Thread 3: Add thread 3 to the queue of the monitor. Remove thread 2 from the queue of its condition variable. Transfer the lock on the monitor to thread 2.
13. Thread 2: Release the lock on the monitor.
14. Thread 2: Continue with the post-synchronization workload.
15. Thread 3: Acquire the lock on the monitor.
16. Thread 3: Release the lock on the monitor.
17. Thread 3: Continue with the post-synchronization workload.

3 Signal and continue vs. signal and wait

3.1 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.

3.2 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.

3.3 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
```

4 Smoke Signals

4.1 Background

This task, originally proposed by Patil [2], was a response to Dijkstra's semaphores.

4.2 Task

There is a table with 3 smokers, and a dealer. The smokers continually smoke and make cigarettes. Each smoker also has an infinite amount of one type of supply (papers, tobacco, matches) to make a cigarette. The smokers cannot accumulate supplies that are not their own. They smoke a single cigarette, then try to acquire the required supplies to make a new one, ad infinitum.

The dealer is responsible for non-deterministically selecting two smokers, taking one of each of their supplies, and placing them on the table. He then notifies the third smoker that he/she may take these supplies and make another cigarette when he is finished his current cigarette (if he has one). When the dealer sees the table is again empty, he will repeat the action of placing supplies on the table.

Try to formulate the dealer and each smoker as a separate process. You may use either monitors or semaphores to solve the problem.

4.3 Solution

We present a solution using semaphores. There are 4 semaphores: one for the table, initialized to 1, and one for each smoker placed in an array of semaphores. All smoker semaphores are initialized to 0.

The dealer process continually performs the following actions:

```
table.down
select the smoker, k, to not take supplies from
smoker[k].up
```

Each smoker process, i, continually performs the following actions:

```
smoker[i].down
sleep (make_cigarette_time)
table.up
sleep (smoke_time)
```

5 Unisex bathroom

5.1 Background

This task has been adapted from *Foundations of Multithreaded, Parallel, and Distributed Programming* [1]. In an office there is a unisex bathroom with n toilets. The bathroom is open to both men and women, but it cannot be used by men and women at the same time.

5.2 Task

1. Develop a Java program that simulates the above scenario using Java built-in monitors. Your solution should be deadlock and starvation free.
2. Justify why your solution is starvation free.

5.3 Solution

The program and the justifications can be found in the source that comes with this solution.

References

[1] Gregory R. Andrews. Foundations of Multithreaded, Parallel, and Distributed Programming. Addison Wesley, 1999.

[2] Sahus Patil. Limitations and capabilities of Dijkstra's semaphore primitives for coordination among processes. MIT, 1971.