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?

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) ...

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.

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.

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.

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.