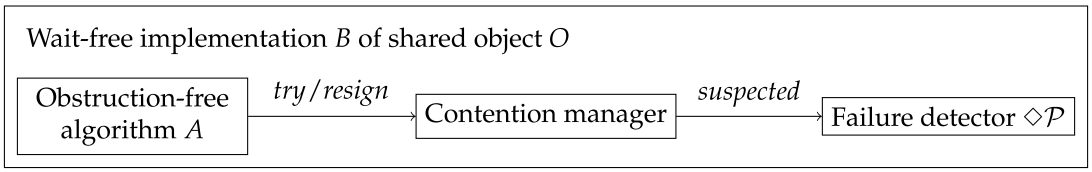
Concurrent Algorithms November 14, 2016

Exercise 6

Problem 1.

Let A be an *obstruction-free* algorithm implementing some shared object O with operations $op_1, \ldots, op_k$. The goal of the exercise is to transform algorithm A into a *wait-free* algorithm B that also implements shared object O (i.e., the operations $op_1, \ldots, op_k$). We will do it by implementing an abstraction called a *contention manager*, using an *eventually perfect* failure detector $\diamond\mathcal{P}$ and atomic registers.



A contention manager implements two operations: try_i and $resign_i$ (invoked by process p_i). These operations do not take any arguments and always return *ok*. A contention manager resolves contention, and thus guarantees wait-freedom, by delaying some processes that have invoked try_i. In other words, when a process p_i invokes try_i, a contention manager can decide when to return from the operation—it can delay the response of try_i for an arbitrarily long time.

We assume that algorithm A uses the interface of the contention manager, i.e., that it invokes try_i and $resign_i$. More precisely, every time an operation op_m, implemented by A, is executed by a process p_i, the following conditions are satisfied:

1. try_i is called always before the first step of the implementation of op_m is executed (i.e., just after op_m is invoked), and possibly many times while op_m is being executed, (You may stop the implementation of op_m at some point, call try_i, and later resume op_m at the same point.)

2. $resign_i$ is called *only* immediately after the last step of the implementation of op_m is executed (i.e., just before the result of op_m is returned),

3. If process p_i is correct but does not return from operation op_m (i.e., the implementation of the operation keeps executing), then p_i keeps calling try_i many times. (The number of times should be finite as the problem asks you for a wait-free algorithm. However, the number is unbounded as the failure detector introduced below only guarantees some property after some unknown time.)

Moreover, every time process p_i invokes try_i or $resign_i$, p_i waits until $try_i / resign_i$ returns before executing any further steps of algorithm A.

An eventually perfect failure detector $\diamond\mathcal{P}$ maintains, at every process p_i, a set $suspected_i$ of suspected processes. $\diamond\mathcal{P}$ guarantees that eventually, after some unknown time, the following conditions are satisfied:

1. Every correct process permanently suspects every crashed process,

2. No correct process is ever suspected by any correct process.

This means that $suspected_i$ can be arbitrary and different at every process for any *finite* period of time. However, eventually, at every correct process p_i, set $suspected_i$ will be permanently equal to the set of processes that have crashed.

Your task is to implement a contention manager C (i.e., the operations try_i and $resign_i$, for every process p_i) that converts obstruction-free algorithm A into wait-free algorithm B, and that uses only atomic registers and failure detector $\diamond\mathcal{P}$.