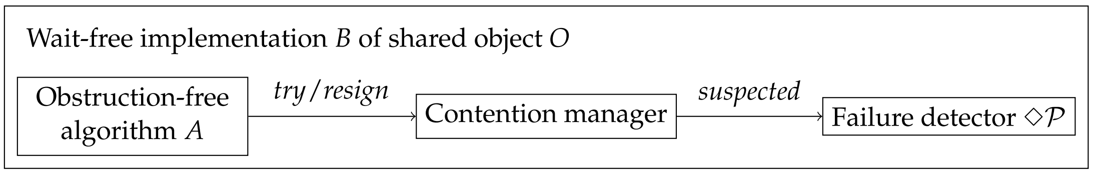
Concurrent Algorithms November 14, 2017

Solutions to 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.


Wait-free implementation B of shared object O
Obstruction-free algorithm A
$try/resign$
Contention manager
$suspected$
Failure detector $\diamond\mathcal{P}$

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,
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 never returns from operation op_m (i.e., the implementation of the operation is executed infinitely long), then p_i calls try_i infinitely many times.

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 $\diamond\mathcal{P}$.

Solution

The following algorithm implements a contention manager that transforms any obstruction-free algorithm into a wait-free one:

```
uses: $T[1,\dots,N]$—array of registers, $Executing[1,\dots,N]$—atomic wait-free snapshot object
initially: $T[1,\dots,N] \leftarrow \perp$, $Executing[1,\dots,N] \leftarrow \perp$

upon $try_i$ do
    if $T[i] = \perp$ then $T[i] \leftarrow$ GetTimestamp()

    repeat
        $sact_i \leftarrow \{ p_j \mid T[j] \neq \perp \wedge p_j \notin \Diamond\mathcal{P}.suspected_i \}$
        $Executing.update(i, \perp)$
        $leader_i \leftarrow$ the process in $sact_i$ with the lowest timestamp $T[leader_i]$
        if $leader_i = i$ then $Executing.update(i, i)$
    until $Executing.scan()$ contains only i and $\perp$

upon $resign_i$ do
    $T[i] \leftarrow \perp$
    $Executing.update(i, \perp)$
```

The algorithm uses a procedure `GetTimestamp()` that generates *unique* timestamps. We assume that if a process gets a timestamp t from `GetTimestamp()`, then no process can get a timestamp lower than t infinitely many times. Thus, we can easily implement `GetTimestamp()` using only registers (or even without using any shared objects). For example, we can use the output of a counter (see the lecture notes on how to implement a counter from registers) combined with a process id (to ensure that timestamps are unique). The algorithm also uses a wait-free, atomic snapshot object to store the process that should be executing next (or is currently executing) in order to avoid two processes executing concurrently.