A Solution for the Exercise 5

Michał Kapałka

EPFL, LPD

STiDC’06, 12.XII 2006

The aim of the exercise

We have:

- An obstruction-free algorithm *A* – implementation of a shared object *O*
- Failure detector $\Diamond\mathcal{P}$

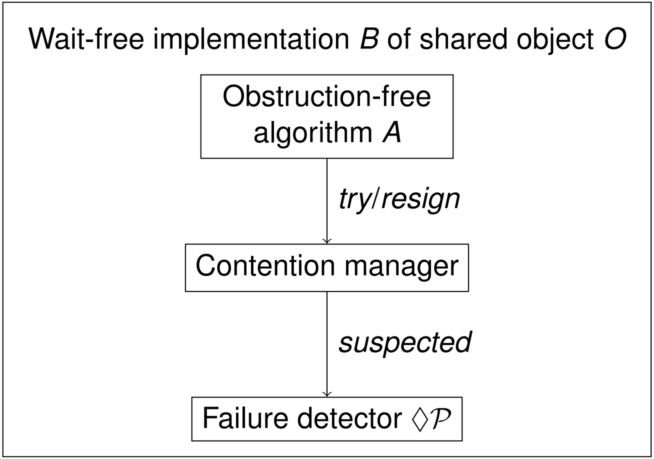
We want:

- A wait-free implementation of shared object *O*

We need:

- A contention manager that transforms any obstruction-free algorithm into a wait-free one

The big picture



Wait-free implementation *B* of shared object *O*

Obstruction-free algorithm *A*

try/*resign*

Contention manager

suspected

Failure detector $\Diamond\mathcal{P}$

Obstruction-freedom

- Obstruction-freedom $=$ progress only in the absence of contention $\Rightarrow$conditional progress
- Weaker than wait-freedom

Why obstruction-freedom?

- Easier to implement and optimize than wait-freedom – separation of concerns:
  - Obstruction-free algorithm $=$ safety $+$ weak liveness
  - Contention manager $=$ stronger liveness
- Contention managers can provide wait-freedom or can use simple heuristics (e.g., exponential back-off)
- Contention managers can be tuned to particular system / application and combined – safety always preserved

Assumptions

Algorithm A must communicate with a contention manager ⇒calls *try* and *resign*:

- try_i is called always before an operation starts, and possibly many times within the operation,
- $resign_i$ is called *only* immediately before the operation returns,
- If a process p_i is correct but never returns from an operation then p_i calls try_i infinitely many times.

Failure detector $\Diamond\mathcal{P}$

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.

A first approach

```
uses: $T[1,\ldots,n]$—array of single-bit registers
initially: $T[1,\ldots,n] \leftarrow false$

upon $try_i$ do
    $T[i] \leftarrow true$
    repeat
        $leader_i$ = the non-suspected process with $T[leader_i] = true$
        with the lowest process id
    until $leader_i = p_i$

upon $resign_i$ do
    $T[i] \leftarrow false$
```

Not wait-free – possible starvation (in fact: lock-free)

A first approach

```
uses: $T[1, \ldots, n]$—array of single-bit registers
initially: $T[1, \ldots, n] \leftarrow false$

upon $try_i$ do
    $T[i] \leftarrow true$
    repeat
        $leader_i$ = the non-suspected process with $T[leader_i] = true$
        with the lowest process id
    until $leader_i = p_i$

upon $resign_i$ do
    $T[i] \leftarrow false$
```

Not wait-free – possible starvation (in fact: lock-free)

A wait-free contention manager

```
uses: $T[1,\ldots,N]$—array of registers
initially: $T[1,\ldots,N] \leftarrow \bot$

upon $try_i$ do
    if $T[i] = \bot$ then $T[i] \leftarrow$ GetTimestamp()
    repeat
        $sact_i \leftarrow \{\, p_j \mid T[j] \neq \bot \wedge p_j \notin \Diamond\mathcal{P}.suspected_i \,\}$
        $leader_i \leftarrow$ the process in $sact_i$ with the lowest
        timestamp $T[leader_i]$
    until $leader_i = p_i$

upon $resign_i$ do
    $T[i] \leftarrow \bot$
```

Properties of `GetTimestamp()`

- Timestamps have to be unique
- They should also be increasing, but atomicity not necessary
- A solution (implemented with registers): combine a value returned by a weak counter with process id