

The Weakest Failure Detectors to Boost Obstruction-Freedom

Rachid Guerraoui¹ **Michał Kapalka¹** Petr Kouznetsov²

¹EPFL, Switzerland

²MPI-SWS, Germany

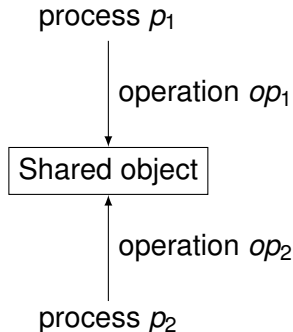
DISC 2006, 20.IX 2006

Problems with Concurrent Programming

- Multi-processor/-core \Rightarrow synchronization techniques essential
- **Ideal** implementations of shared objects:

Linearizable (atomic) + Wait-free or Non-blocking (lock-free)
--

- Wait-free = progress for everyone
- Non-blocking = progress for someone



Problems with Wait-Freedom/Non-blockingness

But wait-free/non-blocking + linearizable algorithms:

- Difficult to design
- Difficult to optimize for average case = **low contention**

Solution: Separation of Concerns

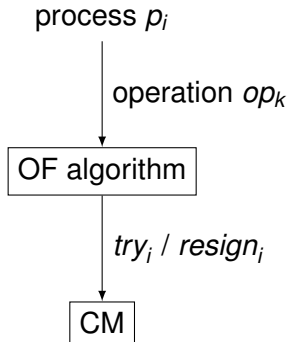
Two **independent** modules:

- 1 **Obstruction-free** (OF) algorithm \Rightarrow **safety** + minimal liveness
 - Must **always** return correct results (linearizability)
 - Obstruction-freedom: progress guaranteed only when no contention
- 2 **Contention manager** (CM) \Rightarrow boosts **liveness**
 - CM has limited power \Rightarrow safety always preserved, even when CM behaves badly.

The idea adopted by OF software transactional memory

Contention Manager

- OF communicates with CM **only** via calls *try* and *resign* (no parameters, return OK)
- But CM cannot mess up with safety
 ⇒ CM can only **delay a process** that calls *try* to help other processes
- *try* = when operation starts or when contention
- *resign* = when operation completes



Providing Wait-Freedom

- Our focus: CM that guarantees wait-freedom or non-blockingness
- How? By allowing each (some) process to run its operation in isolation **sufficiently long**
- How long is sufficiently long? Asynchronous system \Rightarrow no upper bound \Rightarrow until the operation is **completed**, or the process **crashes**

Wait-Free CM – an Example

Process p_1	Process p_2
starts op_1 runs alone completes op_1	starts op_1 suspended
starts op_2 runs alone completes op_2 ...	continues contention suspended continues ...

- p_1 has to be blocked **indefinitely** long (p_2 may be very slow).
- But if p_2 **crashes**, p_1 cannot remain blocked forever!
- CM needs **some** information about failures.

Wait-Free CM – an Example

Process p_1	Process p_2
starts op_1 runs alone completes op_1	starts op_1 suspended
starts op_2 runs alone completes op_2	continues contention suspended
...	continues
... blocked by CM	... runs alone completes op_1

- p_1 has to be blocked **indefinitely** long (p_2 may be very slow).
- But if p_2 **crashes**, p_1 cannot remain blocked forever!
- CM needs **some** information about failures.

Wait-Free CM – an Example

Process p_1	Process p_2
	starts op_1
starts op_1	suspended
runs alone	
completes op_1	continues
	contention
	suspended
starts op_2	
runs alone	
completes op_2	continues
...	...
blocked by CM	runs alone
continues	CRASHES

- p_1 has to be blocked **indefinitely** long (p_2 may be very slow).
- But if p_2 **crashes**, p_1 cannot remain blocked forever!
- CM needs **some** information about failures.

The Question

Question

What is the minimal amount of information about failures needed to guarantee wait-freedom using a CM?

Answer

*Information about failures has to be **eventually accurate** ($\diamond\mathcal{P}$).*

The Question

Question

What is the minimal amount of information about failures needed to guarantee wait-freedom using a CM?

Answer

*Information about failures has to be **eventually accurate** ($\diamond\mathcal{P}$).*

Sufficiency Part

Basic idea: make processes execute operations **one by one**

⇒ no contention

initially: $T[1, \dots, n] \leftarrow \perp$

upon try_i **do**

[**if** $T[i] = \perp$ **then** $T[i] \leftarrow GetTimestamp()$
 repeat
 | $leader_i \leftarrow$ the **non-crashed** process
 | that announced the lowest non- \perp ts in T
 until $leader_i = i$

upon $resign_i$ **do**

└ $T[i] \leftarrow \perp$

Necessity Part

The main idea:

- We have an algorithm C implementing a CM that guarantees wait-freedom.
- For every pair of processes p_i and p_j (p_i never crashes) we want that:
 - 1 If p_j **crashes**, then p_i eventually permanently suspects p_j ,
 - 2 If p_j **never crashes**, then p_i eventually stops suspecting p_j forever.
- We make p_i and p_j invoke *try* and *resign* on $C \Rightarrow$ simulate an execution of an OF algorithm

Necessity Part (2)

Process p_i	Process p_j
try_i	try_j
suspect p_j	inc R_j
wait for inc R_j	
stop suspecting p_j	try_j
$resign_i$	inc R_j
	...
try_i	
suspect p_j	
...	

Necessity Part (2)

Process p_i

try_i
 suspect p_j
 wait for inc R_j
 stop suspecting p_j
 $resign_i$

try_i
 suspect p_j
 wait for inc R_j

Process p_j

try_j
 inc R_j

try_j
 CRASHES

If p_j **crashes**: p_i suspects p_j and waits for R_j forever

Necessity Part (2)

Process p_i	Process p_j
try_i	try_j
suspect p_j	inc R_j
wait for inc R_j	
stop suspecting p_j	try_j
$resign_i$	inc R_j
	...
try_i blocked by CM	

If p_j never crashes:

- CM must eventually make p_j perform steps alone
 \Rightarrow block p_i until p_j resigns,
- But p_j never resigns $\Rightarrow p_i$ blocked forever, not suspecting p_i ,
- A **subtlety**: OF is violated then, but if CM releases p_i ∞ many times, OF holds and CM violates wait-freedom.

Contribution

Main results:

- 1 $\diamond\mathcal{P}$ is the weakest failure detector to implement a **wait-free** contention manager
- 2 Ω^* is the weakest failure detector to implement a **non-blocking** contention manager ($\Omega \prec \Omega^* \prec \diamond\mathcal{P}$)

But also:

- 1 Separation of concerns has a cost
- 2 Prove that wait-freedom is more difficult than non-blockingness
- 3 Give a precise model of interaction between obstruction-free algorithm and contention manager

Related Work

- We do not consider **overhead** of CM
- Some discussion + wait-free CM algorithm: Fich et al. (DISC'05)
- More about overhead: see our companion paper (EPFL technical report)