Computing with anonymous processes

Prof R. Guerraoui
Distributed Programming Laboratory
A **counter** has two operations **inc()** and **read()** and maintains an integer $x$ *init to 0*

**read():**
- return($x$)

**inc():**
- $x := x + 1$;
- return(ok)
Counter (atomic implementation)

The processes share an array of SWMR registers $\text{Reg}[1,..,n]$; the writer of register $\text{Reg}[i]$ is $\pi_i$

$\text{inc}()$

```plaintext
temp := $\text{Reg}[i].\text{read()} + 1;
\text{Reg}[i].\text{write}(\text{temp});
\text{return}(\text{ok})
```
Counter (atomic implementation)

`read()`:

```plaintext
sum := 0;
for j = 1 to n do
    sum := sum + Reg[j].read();
return(sum)
```
Weak Counter

A *weak counter* has one operation $wInc()$

**$wInc()$:**

- $x := x + 1;$
- return($x$)

- Correctness: if an operation precedes another, then the second returns a value that is larger than the first one
Weak counter execution

\[ wInc() - 1 \]

\[ wInc() - 2 \]

\[ wInc() - 2 \]
Weak Counter
(lock-free implementation)

The processes share an (infinite) array of
MWMR registers Reg[1,..,n,..,], init to 0

\textbf{wInc():}

\begin{itemize}
  \item i := 0;
  \item while (Reg[i].read() \neq 0) do
    \begin{itemize}
      \item i := i + 1;
      \item Reg[i].write(1);
    \end{itemize}
  \item return(i);
\end{itemize}
Weak counter execution

\[ wInc() - 1 \quad wInc() - 2 \quad wInc() - \]

\[ p1 \quad p2 \quad p3 \]
Weak Counter
(wait-free implementation)

The processes also use a MWMR register \( L \)

\( wInc() \):

\[ i := 0; \]

\[ \text{while} \ (\text{Reg}[i].\text{read()} \neq 0) \ \text{do} \]

\[ \quad \text{if} \ L \ \text{has been updated} \ n \ \text{times} \ \text{then} \]

\[ \quad \text{return the largest value seen in} \ L \]

\[ \quad i := i + 1; \]

\[ L.\text{write}(i); \]

\[ \text{Reg}[i].\text{write}(1); \]

\[ \text{return}(i); \]
Weak Counter
(wait-free implementation)

\( \text{wInc}() : \)

\[ t := l := L.\text{read}(); \ i := k := 0; \]

\[ \text{while} \ (\text{Reg}[i].\text{read}() \neq 0) \ \text{do} \]

\[ i := i + 1; \]

\[ \text{if} \ L.\text{read}() \neq l \ \text{then} \]

\[ l := L.\text{read}(); \ t := \text{max}(t,l); \ k := k + 1; \]

\[ \text{if} \ k = n \ \text{then return}(t); \]

\[ \text{L.write}(i); \]

\[ \text{Reg}[i].\text{write}(1); \]

\[ \text{return}(i); \]
Snapshot (sequential spec)

- A **snapshot** has operations **update()** and **scan()** and maintains an array \( x \) of size \( n \)
  
  - **scan():**
    - return(\( x \))
  
  - NB. No component is devoted to a process
  
  - **update\( (i,v) \):**
    - \( x[i] := v; \)
    - return(ok)
Key idea for atomicity & wait-freedom

The processes share a **Weak Counter**. 
Wcounter, init to 0;

The processes share an array of **registers** 
Reg[1,..,N] that contains each:

- a value,
- a timestamp, and
- a copy of the entire array of values
Key idea for atomicity & wait-freedom (cont’d)

To *scan*, a process keeps collecting and returns a collect if it did not change, or some collect returned by a concurrent *scan*

• Timestamps are used to check if a scan has been taken in the meantime

• To *update*, a process *scans* and writes the value, the new timestamp and the result of the scan
Snapshot implementation

Every process keeps a local timestamp $ts$

$\text{update}(i,v)$:

$ts := \text{Wcounter}.\text{wInc}();$

$\text{Reg}[i].\text{write}(v,ts,\text{self}.\text{scan}());$

return(ok)
Snapshot implementation

```plaintext
scan():
  ts := Wcounter.wInc();
  while(true) do
    If some Reg[j] contains a collect with a higher timestamp than ts, then return that collect
    If n+1 sets of reads return identical results then return that one
```
Consensus (obstruction-free)

- We consider binary consensus

- The processes share two infinite arrays of registers: $\text{Reg}_0[i]$ and $\text{Reg}_1[i]$

- Every process holds an integer $i$ init to 1

- Idea: to impose a value $v$, a process needs to be fast enough to fill in registers $\text{Reg}_v[i]$
Consensus (obstruction-free)

\textit{propose}(v):

while (true) do
  if \( \text{Reg}^{1-v}[i] = 0 \) then
    \( \text{Reg}_v[i] := 1; \)
    \( i := i + 1; \)
  else if \( i > 1 \) and \( \text{Reg}^{1-v}[i-1] = 0 \) then
    return (v);
  else
    \( v := 1 - v; \)
  end
Consensus (solo process)

\[ q(1) \]

- \( \text{Reg0}(1) = 0 \)
- \( \text{Reg1}(1) := 1 \)
- \( \text{Reg0}(2) = 0 \)
- \( \text{Reg1}(2) := 1 \)
- \( \text{Reg0}(1) = 0 \)
Consensus (lock-step)

<table>
<thead>
<tr>
<th>$q(1)$</th>
<th>$p(0)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reg0(1) := 0</td>
<td>Reg1(1) := 0</td>
</tr>
<tr>
<td>Reg1(1) := 1</td>
<td>Reg0(1) := 1</td>
</tr>
<tr>
<td>Reg0(2) := 0</td>
<td>Reg1(2) := 0</td>
</tr>
<tr>
<td>Reg1(2) := 1</td>
<td>Reg0(2) := 1</td>
</tr>
<tr>
<td>Reg0(1) := 1</td>
<td>Reg0(1) := 1</td>
</tr>
</tbody>
</table>
Consensus (binary)

propose(v):
while(true) do
If Reg_{1-v}[i] = 0 then
    Reg_v[i] := 1;
    if i > 1 and Reg_{1-v}[i-1] = 0 then
        return(v);
    else if Reg_v[i] = 0 then v := 1-v;
    if v = 1 then wait(2i)
    i := i+1;
end