Computing with anonymous processes

Prof R. Guerraoui
Distributed Computing Laboratory
Counter (sequential spec)

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 $\text{pi}$

$\text{inc()}$

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

read():

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

\[
\begin{align*}
\text{wInc()} - 1 \\
\text{p1} \\
\text{wInc()} - 2 \\
\text{p2} \\
\text{wInc()} - 2 \\
\text{p3}
\end{align*}
\]
Weak Counter (lock-free implementation)

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

\textit{wInc()}: 

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

\[ \text{wInc()} - 1 \quad \text{wInc()} - 2 \quad \text{wInc()} - \]

p1

p2

p3
The processes also use a MWMR register \( L \)

\[
\text{\textbf{wInc():}}
\]

\[
i := 1;
\]

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

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

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

\[
i := i + 1;
\]

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

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

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

\( wInc() : \)

\[
t := l := L.\text{read}(); i := 1; k := 0;
\]

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

\[
i := i + 1;
\]

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

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

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

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

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

\[
\text{return}(i);
\]
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

\[ \text{update}(i,v): \]
\[ \text{ts} := \text{Wcounter}.\text{wInc}(); \]
\[ \text{Reg}[i].\text{write}(v,\text{ts},\text{self}.\text{scan}()); \]
\[ \text{return}(\text{ok}) \]
Snapshot implementation

\textbf{scan()}: 

\begin{itemize}
  \item \texttt{ts := Wcounter.wInc();}
  \item \texttt{while(true) do}
    \begin{itemize}
      \item If some \texttt{Reg[j]} contains a collect with a higher timestamp than \texttt{ts}, then return that collect
      \item If \texttt{n+1} sets of reads return identical results then return that one
    \end{itemize}
\end{itemize}
Consensus (obstruction-free)

We consider binary consensus

The processes share two infinite arrays of registers: Reg_0[i] and 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 Reg_\( v \)[i]
Consensus (obstruction-free)

`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 v:= 1-v;
  i := i+1;
end
Consensus (solo process)

\[ q(1) \]

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

\[ q(1) \]
- \( \text{Reg0(1)} = 0 \)
- \( \text{Reg1(1)} := 1 \)
- \( \text{Reg0(2)} = 0 \)
- \( \text{Reg1(2)} := 1 \)
- \( \text{Reg0(1)} = 1 \)

\[ p(0) \]
- \( \text{Reg1(1)} = 0 \)
- \( \text{Reg0(1)} := 1 \)
- \( \text{Reg1(2)} = 0 \)
- \( \text{Reg0(2)} := 1 \)
- \( \text{Reg0(1)} = 1 \)
Consensus (binary)

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