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)
The processes share an array of SWMR registers Reg[1,..,n]; the writer of register Reg[i] is pi

\textit{inc()}: 
\begin{itemize}
\item temp := Reg[i].read() + 1;
\item Reg[i].write(temp);
\item return(ok)
\end{itemize}
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

\[ p_1 \quad \text{wInc()} - 1 \]

\[ p_2 \quad \text{wInc()} - 2 \]

\[ p_3 \quad \text{wInc()} - 2 \]
Weak Counter
(lock-free implementation)

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

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

\[ \text{p1} \quad \text{wInc()} - 1 \quad \text{wInc()} - 2 \quad \text{wInc()} - \]

\[ \text{p2} \]

\[ \text{p3} \quad \text{wInc()} - \]
Weak Counter (wait-free implementation)

The processes also use a MWMR register $L$

$\textbf{wInc():}$

1. $i := 0$
2. while $(\text{Reg}[i].\text{read()} \neq 0)$ do
   1. if $L$ has been updated $n$ times then
      1. return the largest value seen in $L$
   2. $i := i + 1$
3. $L.\text{write}(i)$
4. $\text{Reg}[i].\text{write}(1)$
5. return($i$)
Weak Counter
(wait-free implementation)

$\textit{wInc}()$

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

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

$i := i + 1;$

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

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

if $k = n$ then return($t$);

$L.\text{write}(i);$ 

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

return($i$);
A snapshot has operations \texttt{update()} and \texttt{scan()} and maintains an array \( x \) of size \( n \)

\texttt{scan():}
\begin{itemize}
  \item return\((x)\)
\end{itemize}

NB. No component is devoted to a process

\texttt{update\((i,v)\):}
\begin{itemize}
  \item \( x[i] := v; \)
  \item return\((\text{ok})\)
\end{itemize}
Key idea for atomicity & wait-freedom

- The processes share a **Weak Counter**: $\text{Wcounter}$, init to 0;
- The processes share an array of **registers** $\text{Reg}[1,..,N]$ 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 $ts$

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

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

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

$\texttt{return(ok)}$
Snapshot implementation

\textit{scan()}: 
\begin{itemize}
  \item ts := Wcounter.wInc();
  \item while(true) do
    \begin{itemize}
      \item If some Reg\([j]\) contains a collect with a higher timestamp than ts, then return that collect
      \item If 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: $\text{Reg}_0[i]$ 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):
\begin{itemize}
\item while (true) do
\item if $\text{Reg}_{1-v}[i] = 0$ then
\item $\text{Reg}_v[i] := 1$
\item if $i > 1$ and $\text{Reg}_{1-v}[i-1] = 0$ then
\item return (v)
\item else $v := 1-v$
\item \text{end}
\end{itemize}
Consensus (solo process)

\[
\begin{align*}
q(1) \\
\text{Reg0}(1) &= 0 \\
\text{Reg1}(1) &= 1 \\
\text{Reg0}(2) &= 0 \\
\text{Reg1}(2) &= 1 \\
\text{Reg0}(1) &= 0
\end{align*}
\]
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):**

```
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
```