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 \( \pi_i \)

\textit{inc()}:

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

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

Weak Counter

A weak counter has one operation $wInc()$

$wInc()$:  
\[ x := x + 1; \]
\[ \text{return}(x) \]

- 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 $\text{Reg}[1,..,n,..]$, init to 0

$\text{wInc}()$:

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

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

p1

wInc() -

p2

wInc() -

p3
Weak Counter
(wait-free implementation)

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

\[ wInc() : \]

\[ i := 0; \]

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

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

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

\[ i := i + 1; \]

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

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

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

\texttt{wInc()}: \\
\texttt{t := l := L.read(); i := k := 0;} \\
\texttt{while (Reg[i].read() \neq 0) do} \\
\texttt{i := i + 1;} \\
\texttt{if L.read() \neq l then} \\
\texttt{l := L.read(); t := max(t, l); k := k + 1;} \\
\texttt{if k = n then return(t);} \\
\texttt{L.write(i);} \\
\texttt{Reg[i].write(1);} \\
\texttt{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 \textit{scan}, a process keeps collecting and returns a collect if it did not change, or some collect returned by a concurrent \textit{scan}.

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

- To \textit{update}, a process \textit{scans} and writes the value, the new timestamp and the result of the scan.
Snapshot implementation

Every process keeps a local timestamp ts

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

\textit{scan():}

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

\texttt{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):
\begin{itemize}
  \item while(true) do
    \begin{itemize}
      \item if Reg_{1-v}[i] = 0 then
      \item Reg_v[i] := 1;
      \item if i > 1 and Reg_{1-v}[i-1] = 0 then
      \item return(v);
      \item else v := 1-v;
      \item i := i+1;
      \end{itemize}
  \end{itemize}
end
Consensus (solo process)

\[ q(1) \]

\begin{align*}
\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)

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

\[\textit{propose}(v):\]

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