Computing with anonymous processes

Prof R. Guerraoui
Distributed Programming 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)
```
The processes share an array of SWMR registers $\text{Reg}[1,..,n]$; the writer of register $\text{Reg}[i]$ is $\text{pi}$

```
inc():
temp := Reg[i].read() + 1;
Reg[i].write(temp);
return(ok)
```
Counter (atomic implementation)

```plaintext
read():
  sum := 0;
  for j = 1 to n do
    sum := sum + Reg[j].read();
  return(sum)
```
A **weak counter** has one operation **wInc()**

**wInc():**

\[ x := x + 1; \]

\[ \text{return}(x) \]

- Correctness: (a) if op1 precedes another op2, then op2 returns a value that is larger than op1; (b) the value returned does not exceed the number of invocations

- NB. Resembles a regular Fetch&Inc object
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

\[ \text{wInc():} \]
\[ i := 0; \]
\[ \text{while (Reg}[i].\text{read()} \neq 0) \text{ do} \]
\[ i := i + 1; \]
\[ \text{Reg}[i].\text{write}(1); \]
\[ \text{return}(i); \]
Weak Counter Execution

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

p1

p2

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

\[ \text{wInc():} \]
\[
\begin{align*}
i & := 0; \\
\text{while} \ (\text{Reg}[i].\text{read()} \neq 0) \ \text{do} \\
& \quad \text{if } L \text{ has been updated } n \text{ times then} \\
& \quad \quad \text{return the largest value seen in } L \\
& \quad \quad i := i + 1; \\
& \quad L.\text{write}(i); \\
& \quad \text{Reg}[i].\text{write}(1); \\
& \quad \text{return}(i);
\end{align*}
\]
Weak Counter
(wait-free implementation)

\textbf{wInc()}: \\
\hspace{1em} t := l := L.read(); i := k := 0; \\
\hspace{1em} \textbf{while} (\text{Reg}[i].read() \neq 0) \textbf{do} \\
\hspace{2em} i := i + 1; \\
\hspace{2em} \textbf{if} L.read() \neq l \textbf{then} \\
\hspace{3em} l := L.read(); t := \text{max}(t, l); k := k + 1; \\
\hspace{3em} \textbf{if} k = n \textbf{then} \text{return}(t); \\
L.write(i); \\
\text{Reg}[i].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**: $W_{\text{counter}}$, init to 0;

The processes share an array of **registers** $\text{Reg}[1,\ldots,N]$ 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

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

\texttt{scan()}: 
\texttt{ts := Wcounter.wInc();}
\texttt{while(true) do}
\texttt{\hspace{1em}If some Reg[j] contains a collect with a higher timestamp than ts, then return that collect}
\texttt{\hspace{1em}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)

\textbf{propose(v)}:
\begin{verbatim}
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
\end{verbatim}
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)

\[ q(1) \]

\[
\begin{align*}
\text{Reg}0(1) &= 0 \\
\text{Reg}1(1) &= 1 \\
\text{Reg}0(2) &= 0 \\
\text{Reg}1(2) &= 1 \\
\text{Reg}0(1) &= 1
\end{align*}
\]

\[ p(0) \]

\[
\begin{align*}
\text{Reg}1(1) &= 0 \\
\text{Reg}0(1) &= 1 \\
\text{Reg}1(2) &= 0 \\
\text{Reg}0(2) &= 1 \\
\text{Reg}0(1) &= 1
\end{align*}
\]
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