Computing with anonymous processes
Lecture Notes

1 Introduction

In the previous lectures, we assume that each process has an *id* and other processes know their *id*. In this lecture, we want to see how we could implement different types of objects without knowing their *id*s.

2 Counter

2.1 Read counter

A *read counter* is an object with two operations *inc()* and *read()* and it maintains an integer x which is initialized to 0.

```
read():
   return(x)

inc():
   x := x+1;
   return(ok);
```

At first, the question is how we could implement this counter when we know the number of the processes and each process knows its *id*.

Solution. The processes share an array of *SWMR base registers* (Reg[1,...,n] : n is the number of processes) and the writer of register $Reg[i]$ is process p_i. The *inc*() and *read*() operations are implemented as follows:

For *inc*() method, each process p_i reads from its own register ($Reg[i]$), increment it by one and then write to that register ($Reg[i]$).

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

For *read*() method, each process p_i reads from all the registers, and return the sum of the all read values.

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

The *read counter* **could not be implemented** when each process does not know its own *id* and also the total number of the processes. In the following section, we will see how we could implement the weaker types of counter in this situation.

2.2 Weak counter

A *weak counter* is an object with only one operation *wInc()* and it maintains an integer x which is initialized to 0.

```
wInc():
   x := x + 1;
   return(x);
```

The question is how we could implement this counter when each process does not know its own *id* and also the total number of the processes.

Solution. The processes share an infinite array of *MWMR* registers (Reg[1,...,n,...]) which is initialized to 0. For implementing $wInc()$ method, each process starts from register $Reg[0]$, reads the value of registers until it reaches the fist register with value 0. Then it writes '1' in that register and returns the *index* of that register. (Here i is the *index* in the set of registers and it is not the *pid*).

```
wInc():
   i := 0;
   while (Reg[i].read()≠0) do
     i := i + 1;
   Reg[i].write(1);
   return(i);
```

The problem is that the implemented $wInc()$ is not *wait-free*. Suppose a fast process calls the $wInc()$ sequentially, and concurrent with that, a slow process calls $wInc()$ method only once, but each time that the slow process reads the next register to find the first empty one, the fast process fills it sooner. So, the slow register will never reach the empty register.

To solve the problem, each process must know the total number of processes in the system (=n). The processes also share a *MWMR* register L which contains the last return-value of the $wInc()$.

```
wInc():
   t := l := L.read(); i := 0;
   while (Reg[i].read()≠0) do
     if L.read()≠l then
       l := L.read(); t := max(t,l); k := k+1;
       if k=n then return(t);
     i := i +1;
   L.write(i);
   Reg[i].write(1);
   return(i);
```

In this method, each process searches for the first '0' in the set of registers until it sees n updates in L register. If during this time, it could not find an empty register, it will return the largest value seen in L. The reason is that at least of one of the processes calls $wInc()$ after this process and it gets the answer. The *wait-free* implementation of weak counter is as follows.

So, in this section it was shown that the *weak counter* could be implemented when the processes do not know their *id*, but they know the total number of the processes in the system.

3 Snapshot

In this section we show how to implement *snapshot* object when processes do not know their id and also the total number of the processes in the system.

A *snapshot* has two operations *update*() and *scan*() and maintains an array of registers x of size n (Here i is the *index* in the set of registers and it is not the *pid*).

```
scan():
   return(x);

update(i,v):
     x[i] := v;
     return(ok);
```

For implementing *atomic* and *wait-free* snapshot, the processes share a weak counter *wCounter* which is initialized to 0. The processes also share an array of $MWMR$ registers $Reg[1, ..., N]$ that contains each a value, a timestamp and a copy of the entire array of values. The *update* and *scan* operations are implemented as follows.

3.1 Update operation

For updating, a process scans and writes the triple (value, new timestamp, result of scan) in the specified register.

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

3.2 Scan operation

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

```
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
```

4 Consensus

In this section we show how to implement *binary obstruction-free consensus* when processes do not know their id and also the total number of the processes in the system.

Solution. For implementing *binary consensus*, processes share two infinite array of registers ($Reg_0[i]$ *and* $Reg_1[i]$). Each process also holds an integer i which is initialized to 1. The idea is that for imposing a value v, a process needs to be fast enough to fill in registers $Reg_v[i]$.

```
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 := v-1;
     i = i+1;
   end
```

As soon as the process finds two sequential $(1 - v)$, it returns v as the decision value, otherwise it changes the v value to $v - 1$, and tries again. The problem is that the implemented algorithm is not *wait free*. Assume we have two processes p_1 and p_2 and process p_1 wants to propose 0 and process p_2 wants to propose 1 and these

two processes fill in $Reg_0[i]$ and $Reg_1[i]$ arrays sequentially. So, one of the arrays starts from 0 and fills with the sequence of (0,1) values and another array starts from 1 and fills with the sequence of (1,0) values. So, we could not find two sequential 0 in these two arrays; so the algorithm is not *wait free*. For solving the problem, we must stop one of the processes for a period of time such that the other process could find two sequential 0s. The *wait free* algorithm is as follows.

```
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 := v-1;
     if v=1 then wait(2i);
     i = i+1;
   end
```

The idea is that the process that was interested to fill the register i of ($Reg_0[]$) array and it failed, changes the v to '1' and in the next attempt will be interested to fill the ($Reg_1[]$) array, but before starting the next attempt, it must wait for $2i$ period of time. This implies that the other process could eventually find two sequential '0' in ($Reg_1[]$) array and so it could decide value '0'. So, the implemented algorithm is *wait free*.