Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Parallel Programming
Assignment 12: Consensus
Spring Semester 2020

Assigned on: **11.05.2020** Due by: **18.05.2020**

Overview

Distributed consensus is a fundamental problem in computer science. It is defined as follows: There are N agents, each agent provides an input, i.e., an integer number. At the end of the consensus protocol, all agents need to agree on a single value out of the inputs provided by any agent. Some of the agents may fail or progress arbitrarily slow, yet we still require the protocol to terminate in a finite number of steps (wait-free consensus).

Exercise 1 – Wait-free implies lock free

Explain why a valid wait-free consensus protocol cannot use locks.

Answer: In a wait-free algorithm, any thread needs to be able to make progress independently of other threads. An algorithm is wait-free if every operation has a bound on the number of steps the algorithm will take before the operation completes. Thus, we cannot use locks to implement a wait free algorithm (assuming locks are used to protect shared resources), as Thread A might obtain the lock, then become really slow. During this period all other threads that want to obtain the lock are blocked and cannot make any progress.

Execise 2 – Valence states

Assume N=2 and inputs are either 0 or 1 for each agent. Thus in the initial state of any consensus protocol we are in a bivalent state (bivalent = the output can be 0 or 1). However, at termination all agents have agreed on a single value, thus we are in a univalent state. Explain why there is a finite number of bivalent states in any wait-free consensus protocol.

Answer: Wait-freedom implies that the protocol finishes in a finite number of steps. Thus there can only be a finite number of states and no cycles.

Exercise 3 – Consensus among prisoners

Imagine there are 100 people in a prison. Each day the warden picks a prisoner (each prisoner with the probability 1/100). The prisoner is led to a room with a light that he can turn on or off. Initially the light is turned off. After the prisoner was in the room he can state ”by now every prisoner was in the room at least once". If this statement is made and it is true, all prisoners are released. If the statement is made and it is false, all prisoners are shot. Devise a strategy that the prisoners can follow to make sure they get released some day in the future with absolute certainty (no other communication is allowed).

Answer: The first prisoner selected is in charge of turning the light on whenever it is found off. Each other prisoner is to turn the light off the very first time they find it on, otherwise they are to leave it in the state they found it. When the first prisoner selected turns the light on for the 100th time, they can safely assume that all prisoners have been in the interrogation room.

1 Exercise 4 – Implementing two thread consensus

Assume you have a machine with atomic registers and an atomic test-and-set operation with the following semantics (X is initialized to 1):

```
int TAS() {
  res = X;
  if (res = 1) {
    X = 0
  }
  return(res)
}
```

Implement a two-process consensus protocol using TAS() and atomic registers.

Answer:

```
// use two atomic registers, one to store the value we propose, one holds
// the value the other thread proposed. For simplicity we access them like
// an array

int propose(int p) {
    // assume each thread has an id (thread_id) which is either zero or one
    R[thread_id] = p;
    val = TAS()
    if (val == 1) return p;
    else return R[(thread_id + 1) % 2];
}

Explanation: TAS returns one exactly once, to the first thread that calls it.
This thread is the "winner" and returns its own value, however, before a thread
calls TAS he stores its proposed value in a shared atomic register so the
other thread (once he knows he lost) can access and return it. The algorithm
is wait free since it does not contain loops or blocking operations.
```

Exercise 5 – Linearizability

Which of the following scenarios are lineraly consistent, assuming `s` is a stack? Either mark the point of linearization or explain why it is not linearly consistent.

a)
```
A : s.push(1)
B : s.push(2)
A : s.pop()
B : s.pop()
A : void
B : void
A : 2
B : 1
```

Answer: This is not a valid history. A and B did not finish the push operations before they start pop().

b)
```
A : s.push(1)
B : s.push(2)
A : void
B : void
A : s.pop()
A : 2
B : s.pop()
B : 1
```

Answer: Yes the linearization point of A : push(1) is before that of B : push(2)

c)
```
A : s.push(1)
B : s.push(2)
A : void
B : void
B : s.pop()
B : 1
A : s.pop()
A : 2
```

Answer: Yes the linearization point of A : push(1) is after that of B : push(2)