COL818 A1

Author: Yatharth Kumar

Entry Number: 2020CS10413

Files Submitted

1. `LFUniversal.hpp`: Implementation of lock-free version of universal consensus object.
2. `WFUniversal.hpp`: Implementation of wait-free version of universal consensus object.
3. `Node.hpp`: Node class used in consensus class.
4. `Consensus.hpp`: Implementation of Consensus class used in the Node objects.
5. `Consensus.cpp`: Testing of the consensus object defined in Consensus.hpp.
6. `run.sh`: Script to run the test cases on ConcurrentStack and ConcurrentQueue.
7. `Readme.pdf`: Readme in pdf
8. `ConcurrentQueue.cpp`: Implementation of concurrent queue using both LF and WF universal consensus object.
9. `ConcurrentStack.cpp`: Implementation of concurrent stack using both LF and WF universal consensus object.

How to run?

1. Use `sh run.sh <ConcurrentStack/ConcurrentQueue> <Number of Threads>`.
2. Check the results `:)`.

Report

Consensus

The Consensus object is templated implemented using CAS protocol. I have used the `asm` directive to implement it. This Consensus object is for `x86` systems.

```
template<typename T>
class Consensus {
  public:
    T* original{0};

    static int CAS(T **ptr, T* oldVal, T* newVal) {
      unsigned char ret;
      __asm__ __volatile__ (
              "  lock\n"
              "  cmpxchgq %[newval], %[mem]\n"
              "  sete %0\n"
              : "=q" (ret), [mem] "+m" (*ptr), "+a" (oldVal)
              : [newval]"r" (newVal)
              : "memory");
```

```
        return ret;
      }

    public:
      int winner{-1};
      int helpId{-1};
      T* decide(int id, int helpedId, T* prefer) {
        if (CAS(&original, 0, prefer)) {
          winner = id;
          helpId = helpedId;
          return prefer;
        }
        else {
          return original;
        }
      }
  };
```

Each `Consensus` object consists of a `CAS` function implemented using `cmpxchgq` instruction for `x86` which is used inside the `decide` function to decide the `winner` of consensus game. The `decide` function takes the pointers to objects instead of objects itself, and since pointers are 64-bit long, I have used the `cmpxchgq` here. The winner of consensus game are stored inside the `winner` attribute. I have also created a `helpId` which is used in case of `WFUniversal`, when the threads may help other threads, hence it can be used to store which thread was actually helped after the consensus game.

Node

Each `Node` object consists of an invocation defined by `storedFunction` which is of type `std::function<void()>` attribute, a `Consensus` object to decide the next pointer for the `Node`, a sequence number and pointer to next `Node` object. The `storedFunction` attribute can contain *any* invocation. I have used `std::bind` to store callabales inside `storedFunction`.

LFUniversal/WFUniversal

These are the main classes that implement the Universal Consensus algorithm. This class utilises the `Node` and `Consensus` object defined above. The classes contain the `apply` function which takes the `invocation` and `threadid` as input. Each thread uses this to call the desired function on the concurrent object.

Concurrent Stack and Queue

I have used the `LFUniversal` and `WFUniversal` objects to create a concurrent stack and queue. Both stack and queue support the `push` and `pop` operation. One notable thing is that I could use the `std::stack` and `std::queue` easily to create the concurrent stack and queue directly.

Verification

I have used `std::this_thread::sleep_for(std::chrono::milliseconds(rand()))` to create random winning order everytime for consensus games. Also, each thread uses `push` and `pop` randomly based on their thread id. To verify the correctness, after each thread is done with their jobs, I simulate the

`storedFunction`s defined in linked list of `Node`s, this gives me the final state of the concurrent object. Now to check whether this state is correct or not, I once again go through the linked list of `Node`s and simulate `push` and `pop` based on the `winner`/`helpId` stored in the `Node`s. Finally, I check if the two objects have same state or not.

References

The reference for `CAS` function implemented in `Node` class has been taken from: https://copyprogramming.com/howto/cmpxchg-example-for-64-bit-integer