COL818 : Principles of Multiprocessing Systems

Name: Vatsal Jingar
Entry Number: 2020CS50449

Date: 19 Feb 2024
Assignment: 1

Universal Class Implementation and Correctness

1 Theory

We know that Universal Class can make the equivalent sequence of invocations as there could be in any lock free or wait free linearizable algorithm. In further sections, we will show that our implementation of Concurrent Stack based on Universal Class is linearizable. And hence we will see a global real time order of effects. We know that to see same order, the prefix of invocations should same for all the threads. Hence, we will show in last section that every thread will see same invocation. Here the notion of same would be in sense that the either one of them is prefix of another.

2 Implementation of Universal Class

We know that the class should be independent of any object class. Hence I used templates so that one can make Universal Class that could be used in any Object.

2.1 Invocation Class

```
    template <typename A, typename F>
class Invoke
{
public:
    A val;
    F func;
    Invoke() {}
    Invoke(A val, F func) : val(val), func(func) {}
};
```

The class depends of two data types : A and F. A will be type of args and F will be data type of F.
Invoke object can be applied on corresponding Sequential object. The method "apply" of Sequential Object can use Invoke object to know the function and corresponding arguments to be used.

2.2 CAS Implementation

I have used __asm.__volatile (assembly instructions) in x86_64 (local system architecture) to implement CAS. I have used lock cmpxchg x86 instruction for this.

```
    bool compare_and_set(volatile int *r, int expected, int new_value)
{
    unsigned char result;
    asm volatile(
        "lock cmpxchg %3, %1\n\t" // Compare and swap
        "sete %0"                 // Set result based on zero flag
        : "=q"(result), "+m"(*r)
        : "a"(expected), "r"(new_value)
        : "memory");
    return result;
}
```

2.3 Consensus

```
template <typename N>
class Consensus
{
public:
    volatile int FIRST;
    N* mp[n];
    Consensus()
    {
        FIRST = -1;
    }
    N* decide(N *node, int myid)
    {
        mp[myid] = node;
        bool success = compare_and_set(&this->FIRST, -1, myid);
        if (success)
        {
            return mp[myid];
        }
        else{
            return mp[this->FIRST];
        }
    }
};
```

I have used same algorithm for consensus discussed in class. The implementation of apply method is different in the way that input method will take one extra parameter, the value i which will be thread id. I have passed this thread id using the global loop where this threads would be running. In theory we can directly the process id and hash it to get the id in between 0 to n(number of threads) but this can lead to collisions of id. Hence breaking the correctness. Therefore, I passed this input from scope from where these threads are called.

2.4 Definition of Lock Free Universal

```
    template <typename O, typename A, typename F>
class LFuniversal
{
public:
    Node<A, F> *head[n];
    Node<A, F> *tail;
    LFuniversal();
    void apply(Invoke<A, F> *invoke, O *object, int i);
};
```

The definition of Waitfree contains one more array of Node pointers named announce. Templates shows that one can pass any function and arg. Note that O* object should be **Empty**.

3 Implementation of Concurrent Stack and Queues

For the case of Stack, we will first define a Sequential object over a regular stack class. This sequential class would be set of a regular class stack object and a general method apply. This apply method will be called by Universal Class apply method.

```
    template <typename A, typename F>
class SeqStack
{
public:
    stack<A> s;
    SeqStack() {}
    void apply(Invoke<A, F> *invoke)
    {
        invoke->func(s, invoke->val);
    }
};
```

One can now define some functionalities that could be called on base object, in this case, push and pop. Once we had defined the Sequential Object and functionalities that we want, we

can defined Concurrent Stack that uses empty sequential object and Universal class to fill that empty sequential object.

```
template <typename O, typename A, typename F>
class ConcurrentStack_WaitFree
{
public:
    WFuniversal<O, A, F> wuniversal;
    void push(Invoke<A, F> *invoke, O *seqstack, int i)
    {
        wuniversal.apply(invoke, seqstack, i);
    }
    void pop(Invoke<A, F> *invoke, O *seqstack, int i)
    {
        wuniversal.apply(invoke, seqstack, i);
    }
};
```

As we can observe we are providing two interfaces to interact with concurrent stack. But as our Universal class is abstract enough to handle all the functions, both have exact same code except the change in name.
To watch this concurrent stack in action, we will define a global concurrent stack object and per thread invocation objects. I had implemented the threads in OpenMP by using **pragma omp parallel for directive**.

I had implemented the **ConcurrentQueue** in similar way. The difference is that the base class object of sequential object is queue. We assumed that the queue provides same functionalities but with different semantics.

4 Verification of Universal Class, Concurrent Stack and Concurrent Queue

Here we will first prove that the implementations of Concurrent Stacks and Concurrent Queues are linearizable. Let's again look at the implementations in more algorithmic way.

```
ConcurrentStack<O,A,F>:
    UniversalObj;
    push(invocable, object, int i){
        newState = UniversalObj.apply(invocable, object, i);
        return newState
    }
    pop(invocable, object, int i){
        newState = UniversalObj.apply(invocable, object, i)
        return newState
    }
```

- Similar Algorithm is for Concurrent Queue except the base object and change is semantics of functionalities of push and pop.

Claim : Universal Class implemented provides the valid winner for any game and hence correctly appends the invocation node.
This holds as discussed in Class.
Informally, the consensus game is defined over cas which makes it generic object to be used any number of threads. As only one can win the consensus game, we can argue that it can make a valid sequential history for any linearizable history H.
Claim 1 : *The object of ConcurrentStack class is linearizable.*

- *Proof :* First of all proving Claim 1 will also prove that Class *ConcurrentQueue* is linearizable.
- Consider an arbitrary object of Class ConcurrentStack be *S*.
- We will prove the linearizability for *S* using **Induction**.
  We will prove that Universal Object executes the invocation according to a legal sequential history P for any History H (invocation history on S).
  This is equivalent to proving that S is linearizable according to definition of Linearizability.
  As proving the above statement will prove that for any history of invocations on S are executed in strict real time order.
- **Base Case** : The base case would be when only one method is invocated which is linearizable trivially.
- **Induction Hypothesis** : Let the History H be an arbitrary history of execution of invocations on object *S*. Assume that the history is sequential till k invocations and P' is corresponding legal sequential history and U executes this P' to create the current state of object.
- 1. **Induction Step** : We will show that Universal Object U executes invocations according to valid sequential history P" of H till k+1 invocations.

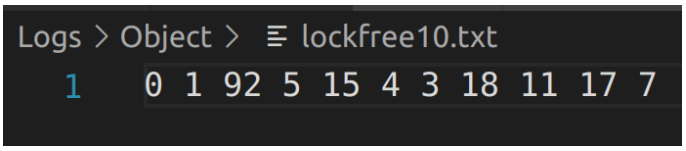
2. If the k^{th} invocation is overlapping with $k + 1^{th}$ invocation, Universal object will add it after the k^{th} invocation. If k^{th} invocation is overlapped and U had chosen k, this means thread corresponding to k^{th} invocation won the consensus game. As Linearizability allows arbitrary order for overlapping histories, inserting $(k + 1)^{th}$ invocation after k^{th} in P'. Hence P"(P' + $(k+1)th$) is also a valid sequential history for History H till k+1. And as U executes P" for History H till k+1, S is linearizable.
3. If the $(k)^{th}$ invocation is completed and after that any new $(k+1)^{th}$ invocation comes it should be appended after k^{th} invocation according to definition of Linearizability. We can observe that for our Universal Object U also, $(k+1)th$ invocation would be appended after k^{th} invocation. This is because completing of k^{th} invocation means that it is already appended in linked list of Universal object U. And as $(k + 1)^{th}$ invocation is started after ending of k^{th}, It must see node corresponding to k^{th} before itself. Hence U will execute a valid sequential history till k+1 invocations.
4. **In general,** If we use Lock free Universal Object than the thread can starve. Even if it starves it is linearizable as the next call would be overlapping hence any order would work.

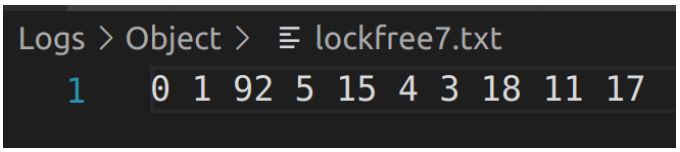
Claim 2 : If *Concurrent Object* uses **Lock Free** Universal Object, the Concurrent Object would be lock free.
Proof : As one thread would always succeed in consensus game by Lock free property of Universal object, one can say that atleast one thread invocations would be appended. If one thread succeed other one of the thread from remaining threads can again succeed in consensus game using the Lock free property of Universal Object. In this way, we can say that system in whole will succeeds in finite steps. Therefore the Concurrent Object S will be **Lock Free**.

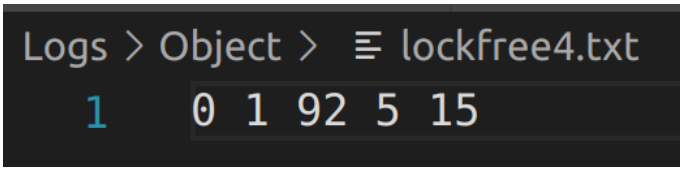
Claim 3 : If *Concurrent Object* uses **Wait Free** Universal Object, the Concurrent Object S would be Wait free.
Proof : As one thread can succeed in Consensus game independent of other threads, we can say that it will complete adding all its invocations in finite number of steps. Hence S is Wait Free. The proof holds due the property that winning consensus game implies adding invocation node at head.

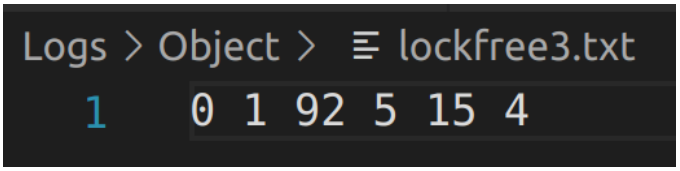
5 Verification using Real Example :

Although in previous section, I had given a rigorous proof that why the Concurrent Stack class would be linearizable. And Linearizability implies a **Global Common order** of effects. This is an example of order of executions seen by some threads for case of **Stack**. To visualize, I have only executed push calls. Once can comment line define Log to watch the execution results in Logs directory.









These are the invocations for Push method on 100 threads on Lock Free Concurrent Stack.