Assignment-4:Implementing Obstruction Free and Wait Free MRMW Snapshot Algorithms

Submitted By: Kushagra Indurkhya
CS19B1017

Introduction

In this assignment I have implemented:

- Obstruction Free MRMW Snapshot algorithm:
  - The update() calls are wait-free, but scan() is not because any call can be repeatedly interrupted by update(), and may run forever without completion.
  - It is however obstruction-free since a snapshot() completes if it runs by itself for long enough.
- Wait-Free MRMW Snapshot algorithm:
  - To make the scan() method wait-free, each update() call helps a scan() it may interfere with, by taking a snapshot before modifying its register

Design

Obstruction Free Snapshot

For the implementation of an Obstruction free MRMW Snapshot as discussed in class.

I am using c++ std::atomic to make an MRMW array of pointers to objects of type reg_values which contain serial number sn,process_id corresponding to the last update, and the value.

```
class reg_values
{
public:
   T value;
   long sn;
   int pid;
   reg_values();
   /**
    * It initializes a reg_values object with the given value, serial number, and
process id.
    *
    * @param v The value to be registered.
    * @param s The serial number of the register.
    * @param p The process ID of the process that created the value.
    */
   reg_values(T v, long s, int p)
   {
       value = v;
       sn = s;
       pid = p;
   }
```

- REG[x].val contains the current value of component x.
- REG[x].(PID,sn) is the “identity” of v.
  - REG[x].pid is the index of the process that issued the corresponding update(x,v) operation

- REG[x].sn is the sequence number associated with this update when considering all updates issued by ppid.

In the update operation store operation on any element of this array, a new object swaps the previous object atomically

```
void update(int thread_id, int x, T value)
   {
       sn[thread_id]++;
       reg[x].store(new reg_values<T>(value, sn[thread_id], x));
       return;
   }
```

In the snapshot operation, we perform a collect and store the current state of registers in array aa and then keep on performing collects in array bb until we get the exactly states we return these values

Wait-Free Snapshot

For the implementation of wait-free implementation as discussed in class there are two arrays in each object:

operation update(x, v) **is**
    $sn_i \leftarrow sn_i + 1;$
    $REG[x] \leftarrow \langle v, i, sn_i \rangle;$
    $HELPSNAP[i] \leftarrow$ snapshot();
    return()
end operation.

operation snapshot() **is**
    $can\_help_i \leftarrow \emptyset;$
    for each $x \in \{1, \cdots, m\}$ **do** $aa[x] \leftarrow REG[x]$ **end for**;
    repeat forever
        for each $x \in \{1, \cdots, m\}$ **do** $bb[x] \leftarrow REG[x]$ **end for**;
        if $(\forall x \in \{1, \cdots, m\} : aa[x] = bb[x])$ **then** return$(bb[1..m].val)$ **end if**;
        for each $x \in \{1, \cdots, m\}$ **such that** $bb[x] \neq aa[x]$ **do**
            let $\langle -, w, - \rangle = bb[x];$
            if $(w \in can\_help_i)$ **then** return$(HELPSNAP[w])$
                else $can\_help_i \leftarrow can\_help_i \cup \{w\}$
            end if
        end for;
        $aa \leftarrow bb$
    end repeat
end operation.

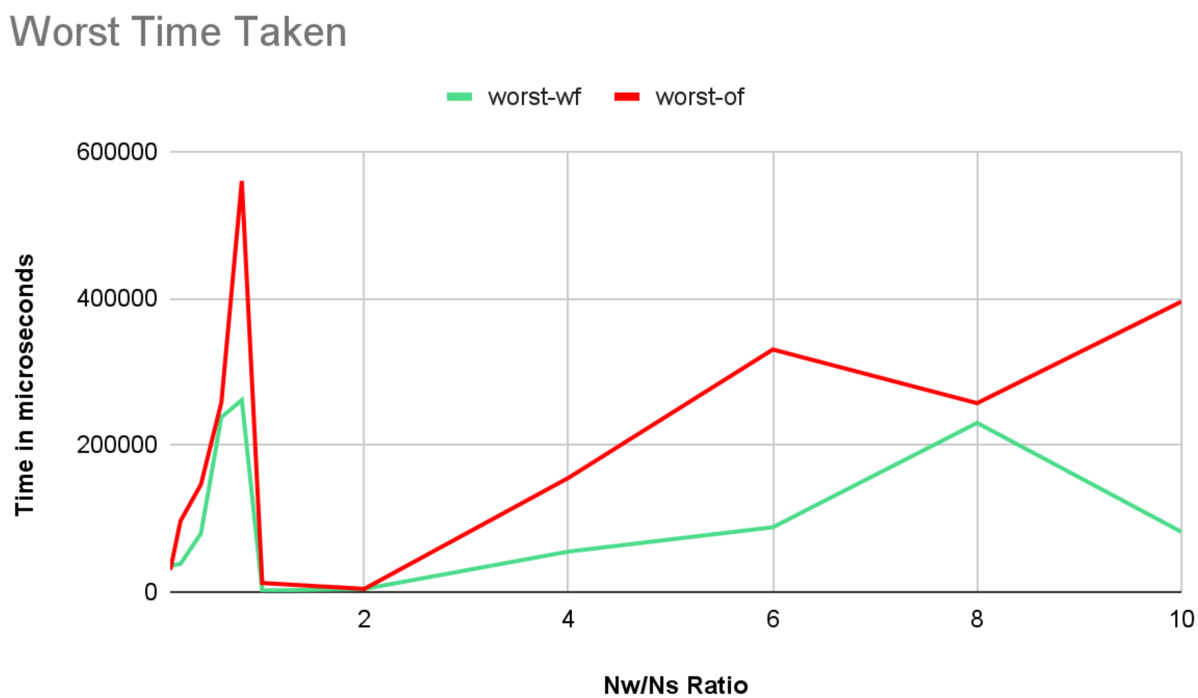
- The first array denoted REG[1..m], is made up of MWMR atomic registers each register has val,pid,sn.
- HELPSNAP[1..n], is made up of one SWMR atomic register per process.
  - HELPSNAP[i] is written only by pi and contains a snapshot value of REG[1..m] computed by pi during its last update() invocation.
  - This snapshot value is destined to help processes that issued snapshot() invocations concurrent with pi 's update.
  - More precisely, if during its invocation of snapshot() a process p j discovers that it can be helped by pi , it returns the value currently kept in HELPSNAP[i] as output of its own invocation of snapshot().
- In the update operation:

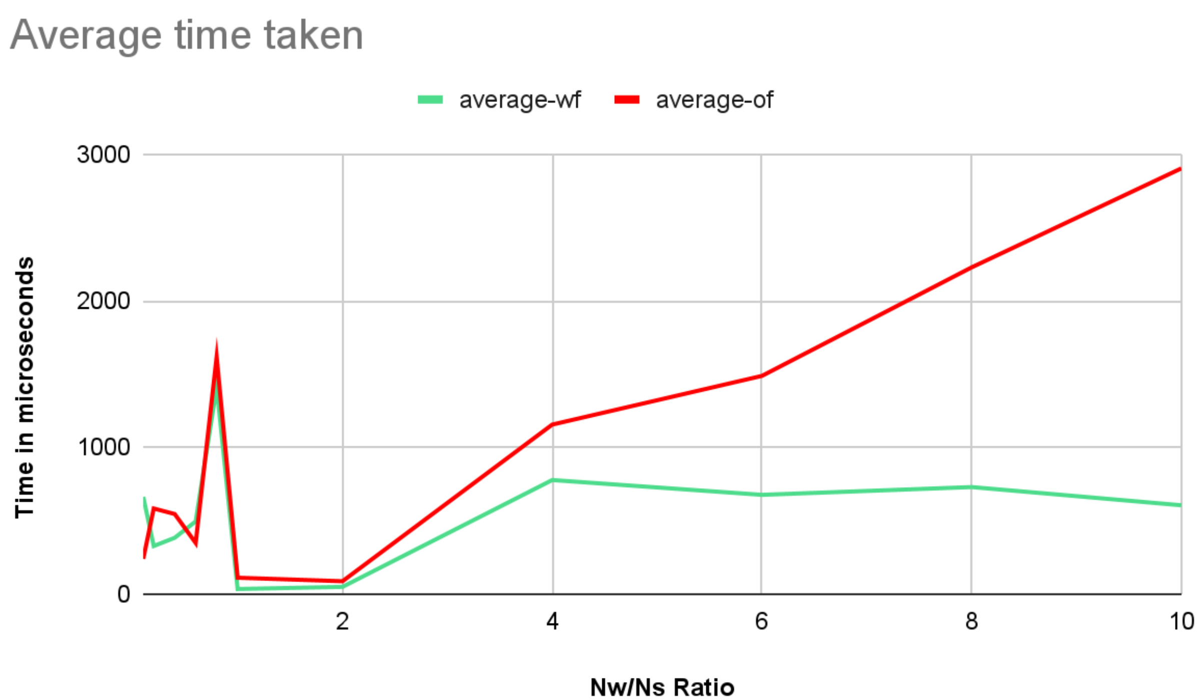
```
void update(int thread_id, int x, T value)
{
    sn[thread_id]++;
    reg[x].store(new reg_values<T>(value, sn[thread_id], x));
    HELPSNAP->at(thread_id) = snapshot(thread_id);
    return;
}
```

- In the snapshot operation:

```
       while (true)
       {
            for (int i = 0; i < m; i++)
            {
                reg_values<T> *temp = reg[i].load();
                bb[i] = new reg_values<T>(temp->get_value(), temp->get_sn(),
temp->get_pid());
            }
            bool flag = true;
            // Checking if the snapshot is consistent
            for (int i = 0; i < m; i++)
            {
                if (aa[i]->get_sn() != bb[i]->get_sn() || aa[i]->get_pid() !=
bb[i]->get_pid())
                {
                    flag = false;
                    break;
                }
            }
            if (flag)
            {
                for (int i = 0; i < m; i++)
                    result[i] = bb[i]->get_value();
                return result;
            }
            /* Checking if the snapshots are the same. */
            for (int i = 0; i < m; i++)
            {
                if (aa[i]->get_sn() != bb[i]->get_sn())
                {
                    /* Checking if the process is in the can_help set. If it is not, it
is added to the set. If it is, it
                    is returned. */
                    int w = bb[i]->get_pid();
                    if (can_help->find(w) == can_help->end())
                        can_help->insert(w);
                    else
                        return HELPSNAP->at(w);
                }
            }
            /* Copying the elements of bb into aa. */
            for (int i = 0; i < m; i++)
                aa[i] = bb[i];
       }
```

Result & Analysis





To compare the average and worst time taken by both the algorithms I kept (Nw,Ns) as (1000,100),(800,100)........(100,100).....(200,1000).......(100,1000) while keeping the other parameters constant.

In doing this experiment I observed the following:

- Wait-free performed better than Obstruction free on most occasions: Both The average and the worst time taken by wait-free were less than the obstruction-free which is what we expected as obstruction-free keeps on running while it receives no clean double collect while receiving no cooperation from update operation so it takes longer
- Anomaly when Ns>Nw : With fewer update threads disturbing the snapshot operation coincidentally there may be times when obstruction-free quickly gets clean double collects gaining an edge in performance over wait-free