1 Elimination Stack

For simplicity, we assume a value $\perp$, which is different from Null and any other value the stack can store. Since Null is used as a legit return value, representing the value of Pop operation (when exchanging values using the elimination array), Null can not be used to represent an initialization value, different then any stack value. The same holds for a Node, since a Null node represent an empty stack, the value $\perp$ is used to distinguish between initialization value and empty stack.

For simplicity, we split the Recover routine into sub-routines, based on which operation (Push, Pop, Exchange) is pending, or needs to be recover. This can be concluded easily by the type of record stored in $Announce[pid]$ (ExInfo or OpInfo), thus there is no need to explicitly know where exactly in the code the crash took place. Also, the Recover routine returns Fail in case the last pending operation did not took affect (no linearization point), nor it will take in any future run. In such case, the user has the option to either re-invoke the operation, or to skip it, depends on the needs and circumstances of the specific use of the data structure.

The given implementation ignores the log of failures and successes of the exchange routine when recovering. That is, in case of a crash during an Exchange, a process is able to recover the Exchange routine, however, the log of successes and failures is not update, since it might be the process already updated it. In addition, in case of a Fail response, we do not know whether the time limit (timeout) was reached, or that the process simply crashed earlier in the routine without completing it. The given implementation can be expanded to also consider the log. Nonetheless, for ease of presentation we do not handle the log in case of a crash. Assuming crash events are rare, the log still gives a roughly good approximation to the number of failures and successes, thus our approach might be useful in practice.

1.1 A Lock-Free Exchanger

An exchanger object supports the Exchange procedure, which allows exactly two processes to exchange values. If process A calls the Exchange with argument a, and B calls the Exchange of the same object with argument b, then A's call will return value b and vice versa.

On the original algorithm [cite the book?!], processes race to win the exchanger using a **CAS** primitive. A process accessing the exchanger first reads its content, and act according to the state of it. The first process observe an Empty state, and tries to atomically writes its value and change the state to Waiting. In such case, it spins and wait for the second process to arrive. The second, observing the state is now Waiting, tries to write its value and change the state to Busy. This way, it informs the first one a successful collision took place. Once the first process notice the collision, it reads the other process value and release the exchanger by setting it back to Empty. In order to avoid an unbounded waiting, if a second process does not show up, the call eventually timeout, and the process release the exchanger and return.

Assume a process p successfuly capture the exchanger by setting its status to Waiting, followed by a crash. Now, some other process q complete the exchange by setting the exchanger to Busy. Upon recovery, p can conclude some exchange was completed, but it can not tell whether its value is part of the exchange, and thus it can not complete the operation. Moreover, p and q must agree, otherwise q will return p's value, and thus the operation of p must be linearized together with q operation.

In order to avoid the above problem, we take an approach resembling the BST implementation. Instead of writing a value to the exchanger, processes will use an info record, containing the relevant information for the exchange. This way, processes use the exchanger in order to exchange info records (more precisely, pointers to such records), and not values. To overcome the problematic scenario described earlier, if a process q observe the exchanger state is WAITING with some record *yourop*, it first update its own record *myop* it is about to try and collide with *yourop*, and only then performs the **CAS**. This way, if the collision is successful, the record *myop* which now stored in the exchanger implies which two records collide. Also, the fact that different processes uses different records guarantee that at most one record can collide with *yourop*.

Using records instead of values, when using wisely, allows us to farther improve the algorithm. First, there is no need to store the exchanger's state in it (by using 2 bits of it to mark the state), but we can rather have this info in the record. Second, if there is a BUSY record in the exchanger, it contains the info of the two colliding records. Therefore, a third process, trying to also use the exchanger, can help the processes to complete the collision, and then can try and set the exchanger back to EMPTY, so it can use it again. In the original implementation, a process observaing a BUSY exchanger, have to wait for the first process to read the value and release the exchanger. Therefore, if the first process crash after the collision, the exchanger will be hold by it forever. The helping mechanism avoids this scenario, making the exchange routine non-blocking.

Notice that no exchange record with EMPTY state is ever created, except for the $default$ record. Therefore, reading EMPTY state is equivalent to the exchanger storing a pointer to $default$. A process p creates a new record *myop* when accessing the exchanger, with a unique address. As long as p fails to perform a successful **CAS**, and thus fails to store *myop* in *slot*, it is allowed to try again. However, once a process performs a successful **CAS** and stores *myop* in *slot*, the only other **CAS** it is allowed to do are in order to try and store $defualt$ in *slot*. Thus, *myop* can be written exactly once to *slot*. It follows that a collision can occur between two processes exactly - once a WAITING record stored in *slot*, only a single **CAS** can replace it with a BUSY record. As the two records can not be written again to *slot*, no other process can collide with any of the records.

The EXCHANGE-RECOVER routine relies on the following argument. If a process p successfully wrote op_p to *slot* using the **CAS** in line 13, the only way to overwrite it by a different process q, is by a **CAS** in line 36 with a record op_q such that its state is BUSY, and $op_q.partner = op_p$. In addition, the only way to overwrite op_q is by a **CAS** replacing it with $default$, and this is done only after SWITCHPAIR(op_p, op_q) is completed, and thus both *result* fields are updated.

The correctness of the EXCHANGE-RECOVER routine is based on the above argument. There are few scenarios to consider. If p crash after a successful **CAS** in line 13, then op_p state is WAITING. Therefore, when reading *slot* in the EXCHANGE-RECOVER one of the following must hold. If *slot* contains op_p, then no process collide with p, and p continue to run as if the time limit has been reached. Otherwise, there was a collision. From the above argument, it must be that either op_q that collide with op_p is stored in *slot*, in this case $op_q.partner = op_p$, and p will try to complete the collision and release *slot*, or that op_q has been overwritten, and in this case the *result* field of op_p is updated. In both cases, p returns $op_p.result$. If p crash after a successful **CAS** in line 36, then op_p state is BUSY. It follows from the argument that the only way to overwrite op_p is only after completing the collision by SWITCHPAIR. Thus, either upon recovery p reads op_p from *slot*, and in this case it tries to complete the the operation, or that $op_p.result$ was already updated. In both cases, p returns it. If non of the above holds, then op_p was not involved in any collision, because either no successful **CAS** was done by p, or p reached the time limit while no process show up, and was able to set *slot* back to $defualt$. In any case, after the crash of p, op_p will never be written again to *slot*, nor any other op_q such that $op_q.partner = op_p$, as any such op_q tries to perform

CAS (op_p, op_q) that will fail. Also, as no process can collide with op_p, no SwitchPair with op_p as parameter is ever invoked, and in particular $op_p.result = \perp$ for the rest of the execution. This in turn implies that upon recovery p will return Fail, as required.

```
Type Node {
    T value
    int popby
    Node *next
}

Type PushInfo {                    ▷ subtype of Info
    Node *pushnd
}

Type PopInfo {                     ▷ subtype of Info
    Node *popnd
}

Type ExInfo {                      ▷ subtype of Info
    {Empty, Waiting, Busy} state
    T value, result
    ExInfo *partner, *slot
}
```

Figure 1: Type definition

ExInfo *default* - global static ExInfo object with state = EMPTY

Algorithm 1: T EXCHANGE (ExInfo **slot*, T *myitem*, long *timeout*)

```
long timeBound := getNanos() + timeout
ExInfo myop := new ExInfo(WAITING, myitem, ⊥, ⊥, slot)
Announce[pid] := myop
while true do
    if getNanos() > timeBound then
        myop.result := TIMEOUT                                   // time limit reached
        return TIMEOUT
    yourop := slot
    switch yourop.state do
        case EMPTY do
            myop.state := WAITING                                // attempt to replace default
            myop.partner := ⊥
            if slot.CAS(yourop, myop) then                       // try to collide
                while getNanos() < timeBound do
                    yourop := slot
                    if yourop ≠ myop then                        // a collision was done
                        if youop.parnter = myop then             // yourop collide with myop
                            SWITCHPAIR(myop, yourop)
                            slot.CAS(yourop, default)            // release slot
                        return myop.result
                end
                // time limit reached and no process collide with me
                if slot.CAS(myop, default) then                  // try to release slot
                    myop.result := TIMEOUT
                    return TIMEOUT
                else                                             // some process show up
                    yourop := slot
                    if yourop.partner = myop then
                        SWITCHPAIR(myop, yourop)                 // complete the collision
                        slot.CAS(yourop, default)                // release slot
                    return myop.result
            end
            break
        case WAITING do                                          // some process is waiting in slot
            myop.partner := yourop                               // attempt to replace yourop
            myop.state := BUSY
            if slot.CAS(yourop, myop) then                       // try to collide
                SWITCHPAIR(myop, yourop)                         // complete the collision
                slot.CAS(myop, default)                          // release slot
                return myop.result
            break
        case BUSY do                                             // a collision in progress
            SWITCHPAIR(yourop, yourop.parnter)                   // help to complete the collision
            slot.CAS(yourop, default)                            // release slot
            break
    end
end
```

Algorithm 2: void SwitchPair(ExInfo *first*, ExInfo *second*)

```
/* exchange the valus of the two operations                                  */
first.result := second.value
second.result := first.value
```

Algorithm 3: T Visit (T *value*, int *range*, long *duration*)

```
/* invoke Exchange on a random entery in the collision array                 */
int cell := randomNumber(range)
return Exchange(exchanger[cell], value, duration)
```

Algorithm 4: T Exchange-Recover ()

```
ExInfo *myop := Announce[pid]                     // read your last operation record
ExInfo *slot := myop.slot                           // and the slot on which it acts
if myop.state = Waiting then
    /* crash while trying to exchange defualt, or waiting for a collision       */
    yourop := slot
    if yourop = myop then                           // still waiting for a collision
        if slot.CAS(myop, default) then                        // try to release slot
            return Fail
        else                                                  // some process show up
            yourop := slot
            if yourop.partner = myop then
                SwitchPair(myop, yourop)                      // complete the collision
                slot.CAS(yourop, default)                               // release slot
            return myop.result
    else if yourop.partner = myop then                   // yourop collide with myop
        SwitchPair(myop, yourop)                              // complete the collision
        slot.CAS(yourop, default)                                       // release slot
        return myop.result
if myop.state = Busy then
    /* crash while trying to collide with myop.partner                          */
    yourop := slot
    if yourop = myop then                     // collide was successful and in progress
        SwitchPair(myop, myop.partner)                        // complete the collision
        slot.CAS(myop, default)                                         // release slot
        return myop.result
if myop.result ≠ ⊥ then
    return myop.result                           // collide was successfuly completed
else
    return Fail
```

Figure 2: Elimination Array routines

Algorithm 5: boolean TryPush (Node *nd)

```
/* attempt to perform Push to the central stack                  */
Node *oldtop := Top
nd.next := oldtop
if Top.CAS(oldtop, nd) then          // try to declare nd as the new Head
    nd.popby.CAS(⊥, Null)            // announce nd is in the stack
    return true
return false
```

Algorithm 6: boolean Push (T myitem)

```
Node *nd = new Node (myitem)
nd.popby := ⊥
PushInfo data := new PushInfo (nd)
while true do
    Announce[pid] := data                    // declare - trying to push node nd
    if TryPush(nd) then                      // if central stack Push is successful
        return true
    range := CalculateRange()                // get parameters for collision array
    duration := CalculateDuration()
    othervalue := Visit(myitem, range, duration)          // try to collide
    if othervalue = Null then         // successfuly collide with Pop operation
        RecordSuccess ()
        return true
    else if othervalue = Timeout then                     // failed to collide
        RecordFailure ()
end
```

Algorithm 7: boolean Push-Roceover ()

```
Node *nd := Announce[pid].pushnd
if nd.popby ≠ ⊥ then                     // nd was announced to be in the stack
    return true
if Search(nd) || nd.popby ≠ ⊥ then // nd in the stack, or was announced as such
    nd.popby.CAS(⊥, Null)                // announce nd is in the stack
    return true
return Fail
```

Algorithm 8: boolean Search (Node *nd)

```
/* search for node nd in the stack                               */
Node *iter := Top
while iter ≠ ⊥ do
    if iter = nd then
        return true
    iter := iter.next
end
return false
```

Figure 3: Push routine

Algorithm 9: T TryPop()

```
Node *oldtop := Top
Node *newtop
Announce[pid].popnd := oldnop                  // declare - trying to pop node oldtop
if oldtop = ⊥ then                                             // stack is empty
│ return Empty
newtop := oldtop.next
oldtop.popby.CAS(⊥, Null)                          // announce oldtop is in the stack
if Top.CAS(oldtop, newtop) then  // try to pop oldtop by changing Top to newtop
│ if newtop.popby.CAS(Null, pid) then   // try to announce yourself as winner
│ │ return oldtop.value
else
│ return ⊥
```

Algorithm 10: T Pop ()

```
Node *result
PopInfo data := new PopInfo (Top)
while true do
│ Announce[pid] := data                        // declare - trying to perform Pop
│ result := TryPop()                          // attempt to pop from central stack
│ if result ≠ ⊥ then                        // if central stach Pop is successful
│ │ return result
│ range := CalculateRange()                  // get parameters for collision array
│ duration := CalculateDuration()
│ othervalue := Visit(Null, range, duration)                     // try to collide
│ if othervalue = Timeout then                                // failed to collide
│ │ RecordFailure ()
│ else if othervalue ≠ Null then   // successfuly collide with Push operation
│ │ RecordSuccess ()
│ │ return othervalue
end
```

Algorithm 11: T Pop-Recover()

```
Node *nd := Announce[pid].popnd               // crash while trying to pop node nd
if nd = ⊥ then                                         // pop from an empty stack
│ return Empty
if Search(nd) then                             // nd was not removed from the stack
│ return Fail
nd.popby.CAS(Null, pid)        // nd was removed.  Try to complete the operation
if nd.popby = pid then                  // you are the process to win the pop of nd
│ return nd.value
return Fail
```

Figure 4: Pop routine