MODULE *lock_free_stack_ABA*

For *ABA* problem, see: https://*en.wikipedia.org*/wiki/*ABA_problem*

NOTE: The label "*pop2a*" in *popStack*() below "causes" the bug to be exposed. Try deleting it. This would have the effect of atomically executing:
CAS(*HEAD*, *local_myhead*, *address*[*local_myhead*].*next*);
because it hides the internal assembly code that first computes the register value, *reg_next*, and then executes *CAS*().

Note that the field "*onStack*" is present only for debugging, and for assertions. Don't hesitate to add extra fields and variables to ease the job of model checking.

The bug is the *ABA* problem. When illustrating this bug, the errors in the "Model Checking Results" tab will have too many frames to easily read. You can then remove many of the unnecessary labels in other routines to easily see the cause of the bug: the *ABA* problem.

```
  Always include these PlusCal modules for basic data types.
EXTENDS Naturals, Sequences, TLC, FiniteSets

  These constants will have to be assigned a value within the generated modeule.
CONSTANTS MAX_ITER, MAX_STACK_SIZE

  NUM_THREADS is a PRE-DEFINED constant. It cannot be changed later.
NUM_THREADS ≜ 2

  null is a unique value (pre-defined constant)
NULL ≜ CHOOSE n : n ∉ 1 . . MAX_STACK_SIZE

--algorithm LockFreeStack{
variables retVal = [thread ∈ 1 . . NUM_THREADS ↦ NULL],
          address = [addr ∈ 1 . . MAX_STACK_SIZE ↦
                        [next ↦ NULL, onStack ↦ FALSE, data ↦ 0]],
          initialized = FALSE ;
          HEAD ;

  CAS: compare-and-swap
  CAS must be a macro. This reflects that it is an assembly instruction
    that modifies its arguments. It must not use the call-by-value of a procedure.
  CAS is atomic. So, there are no intermediate labels.
macro CAS( x, y, z )
{
  if ( x = y ) {
    x := z ;   swap y for z as value of x
    retVal[self] := TRUE ;
   } else {
     retVal[self] := FALSE ;
   } ;
 }

procedure $pushStack$( $elt$ )
  variable $local\_myhead$ ;
{
  $push1$: $address[elt].next := HEAD$ ;
  $push2$: assert $address[elt].onStack = \text{FALSE}$ ;
  $push3$: $address[elt].onStack := \text{TRUE}$ ;
  $tryAgainPush$: $local\_myhead := HEAD$ ;
        $push5a$:  $address[elt].next := local\_myhead$ ;
        $push5b$:  $CAS(HEAD, local\_myhead, elt)$ ;
        $push5c$:  if ( $\neg retVal[self]$ ) {
        $push5d$:    goto $tryAgainPush$ ;
                   } ;
  $endPush$: return ;
 }

procedure $popStack$(  )
  variable $local\_myhead$, $reg\_next$, $elt$ ;
{
  $tryAgainPop$:
     $local\_myhead := HEAD$ ;
  $pop1$: if ( $local\_myhead = 0$ ) {   // If I believe $HEAD = 0$, return now.
  $pop1a$:  $retVal[self] := NULL$ ;
  $pop1b$:  return ;
          } ;

  $pop2$:  $reg\_next :=  address[local\_myhead].next$ ;
  $pop2a$: $CAS(HEAD, local\_myhead, reg\_next)$ ;
  $pop2b$: if ( $\neg retVal[self]$ ) {
  $pop2c$:   goto $tryAgainPop$ ;
           } ;
  $pop3$: $elt := local\_myhead$ ;
  $pop4$: assert $address[elt].onStack = \text{TRUE}$ ;
  $pop5$: $address[elt].onStack := \text{FALSE}$ ;
  $endPop$: $retVal[self] := elt$ ;
  return ;
 }

process ( $thread \in 1 \mathrel{..} NUM\_THREADS$ )
  variable $my\_set = \{\}$, $myelt$,
           $init\_thread$, $iterations = MAX\_ITER$ ;
{
  $init1$: $init\_thread := \text{CHOOSE } thr \in 1 \mathrel{..} NUM\_THREADS : \text{TRUE}$ ;
  $init2$: if ( $self = init\_thread$ ) {   // This thread will initialize global data
  init4:   $HEAD := 0$;
  init5:   while ( $HEAD < MAX\_STACK\_SIZE$ ) {
  init6:     $address[HEAD + 1] := [next \mapsto HEAD,\ onStack \mapsto \text{TRUE},\ data \mapsto 1]$;
  init7:     $HEAD := HEAD + 1$;
           } ;
  init8:   $initialized := \text{TRUE}$;   //init_thread will set this global var
         } ;
  init9: await $initialized$;   // all threads will wait for this

  th1: while ( $iterations > 0$ ) {
  th2:   either { if ( $my\_set \neq \{\}$ ) {
                      // Set myelt at random (non-deterministically)
  th2a:              with ( $tmp \in my\_set$ ) { $myelt := tmp$ } ;
  th2b:               $my\_set := my\_set \setminus \{myelt\}$;
  th2c:               call $pushStack(myelt)$;
                  } }
         or {
  th3a:        call $popStack()$;
  th3b:        if ( $retVal[self] \neq NULL$ ) {
  th3c:          $my\_set := my\_set \cup \{retVal[self]\}$  Add the popped elt to my_set
               } } ;
  th4: $iterations := iterations - 1$;
   } ;   end while
} ;   end process

}   \* end algorithm
```

* Modification History
* Last modified *Thu Apr* 11 13:35:54 *EDT* 2019 by gene
* Last modified Sun *Oct* 22 06:26:25 *EDT* 2017 by celestekostopulos-cooperman
* Created *Fri Oct* 20 22:08:53 *EDT* 2017 by celestekostopulos-cooperman