



HELLENIC REPUBLIC  
UNIVERSITY OF CRETE

# Distributed Computing

## Graduate Course

### Section 9: Safe Memory Reclamation

Panagiota Fatourou  
Department of Computer Science

# Hazard Pointers

## Main Ideas

- In the vast majority of algorithms for lock-free dynamic objects, a thread holds only a small number of pointers that may later be used without further validation for accessing the contents of dynamic nodes, or as targets of expected values of ABA-prone atomic comparison primitives.

## Methodology

- Associate a number of single-writer multi-reader shared pointers, called **hazard pointers**, to each process.
- The number of hazard pointers required for each process depends on its algorithm.
- For simplicity, assume that this number is a constant  $K$ , the same for all processes.
- The technique is split into two components:
  - the algorithm for processing retired nodes
  - the extensions to the implementation that should be made in order to ensure that the conditions to guarantee the safety of memory reclamation and ABA prevention are met.

# Hazard Pointers: Access Hazard - ABA hazard

shared head, tail;

*// initially, both point to a dummy node*

```
void enq(T value ) {  
    NODE *next , *last ;
```

```
1. NODE *p = newcell(NODE) ;  
2. p->value = value ;  
3. p->next = NULL;  
  
4. while (TRUE) {  
5.     last = tail ;  
6.     next = last->next ;  
7.     if (last != tail) continue;  
8.     if (next != NULL) {  
9.         CAS(tail, last, next);  
10.        continue;  
11.    }  
12.    if (CAS(last->next , NULL , p)  
        break ;  
13. } // while  
14. CAS(tail, last, p );  
} // Enqueue
```

## Identify hazards and hazardous references

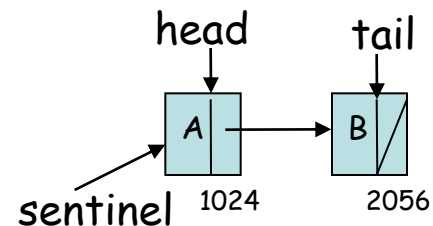
1. Lines 2 and 3 are safe (p points to some node after the execution of line 1)
2. the access on line 6 may be hazardous:
  - between the read of tail (line 5) and the execution of line 6, more nodes may have been inserted and all nodes, including the node pointed to by last, may have been removed (and become free).
3. Line 7: ABA hazard (the ABA problem could appear here)
4. Line 9: ABA hazard
5. Line 12 may be hazardous:
  - access hazard -> accessing the next field of last is not safe (since the node pointed to by last may have been released)
  - ABA hazard
6. Line 14: ABA hazard

# The ABA Problem

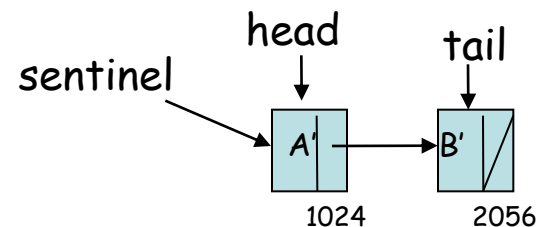
## Thread 0

```
boolean deq(int *pvalue) {
    Node *first, *last, *next;
    while (1) {
        first = head;
        last = tail;
        next = first->next;
        if (first == head) {
            if (first == last) {
                if (next == NULL)
                    return FALSE;
                CAS(tail, last, next);
            }
            else {
                *pvalue = next->value; → B
            }
        }
        if (CAS(head, first, next)) break;
    }
    return TRUE; → INCORRECT!
```

## Thread 1



```
deq()
enq(A')
deq()
enq(B')
```



# Hazard Pointers: Access Hazard - ABA Hazard

```
boolean deq(QUEUE *Q, T *pvalue) {  
    NODE *first, *last, *next;
```

```
1.  while (TRUE)  
2.    first = head;  
3.    last = tail;  
4.    next = first->next;  
5.    if (first != head) continue;  
6.    if (next == NULL)  
7.        return EMPTY;  
8.    if (first = last) {  
9.        CAS(tail, last, next);  
10.       continue;  
    }  
11.   *pvalue = next->value;  
12.   if (CAS(head, first, next))  
        break;  
    } // while  
13. return TRUE;  
} // Dequeue
```

## Identify hazards and hazardous references

1. the access on line 4 may be hazardous:
  - between the read of head (line 2) and the execution of line 4, the node pointed to by first may have been removed (and become free).
2. Line 5: ABA hazard
3. Line 9: ABA hazard
4. Line 11: Access hazard (node pointed to by next may have been de-allocated).
5. Line 12: ABA hazard

# Application of Hazard Pointers: Methodology

1. Examine the target algorithm as follows:
  - a. Identify hazards/hazardous references
  - b. For each distinct hazardous reference, determine the point where it is created and the last hazard that uses it. During this period, a hazard pointer needs to be dedicated to that reference.
  - c. Compare the periods for all hazardous references, and determine how many hazard pointers are required for each process.
2. For each hazardous reference, enhance the algorithm as follows:
  - a. write the address of the node that is the target of the reference to an available hazard pointer.
  - b. Validate that the node is safe.
    - if the validation succeeds, continue normally.
    - if not, skip the rest of the execution and follow the path of the target algorithm when a conflict is detected.
3. Before returning from an operation, release the hazard pointers that have been used during the execution of the operation.
4. At carefully chosen points of the execution of the operation, call `RetireNode()` to identify that a node has been deleted.

# Hazard Pointers Applications

```
void enq(T value ) { // code for process p;  
    NODE *next , *last ;
```

```
1.  NODE *p = newcell(NODE) ;  
2.  p->value = value ;  
3.  p->next = NULL;  
  
4.  while (TRUE) {  
5.      last = tail ;  
        *hp[i][0] = last;  
        if (last != tail) continue;  
6.      next = last->next ;  
7.      if (last != tail) continue;  
8.      if (next != NULL) {  
9.          CAS(tail, last, next);  
10.         continue;  
11.     }  
12.     if (CAS(last->next , NULL , p)  
        break ;  
  
13. } // while  
14. CAS(tail, last, p);  
15. *hp[i][0] = NULL;  
} // Enqueue
```

```
boolean deq(QUEUE *Q, T *pvalue) {
```

```
    // code for process p;
```

```
    NODE *first, *last, *next;
```

```
1.  while (TRUE)  
2.      first = head;  
        *hp[i][0] = first;  
        if (head != first) continue;  
3.      last = tail;  
4.      next = first->next;  
        *hp[i][1] = next;  
5.      if (first != head) continue;  
6.      if (next == NULL) {  
7.          *hp[i][0] = *hp[i][1] = NULL;  
            return EMPTY;  
        }  
8.      if (first = last) {  
9.          CAS(tail, last, next); continue;  
        }  
11.     *pvalue = next->value;  
12.     if (CAS(head, first, next)) {  
        RetireNode(first);  
        break;  
        }  
    } // while  
    *hp[i][0] = *hp[i][1] = NULL;  
13. return TRUE;  
} // Dequeue
```

# Hazard Pointers

- Each process  $p_i$  uses two persistent local variables,  $rlist_i$ , and  $rcount_i$ . These are used to maintain a private list of retired nodes.
- RetireNode() by process  $p_i$ 
  - Insert the retired node into the list of retired nodes of process  $p_i$
  - Whenever the size of the list of retired nodes reaches a threshold  $R$ , call find-hazard() to get a list of the nodes that are currently hazardous.
- Scan() by process  $p_i$ 
  - Scan the HP array for non-null values. Each non-null value is inserted in a local list, called plist.
  - Check each node in  $rlist$ , whether it is included in the plist.
    - If no, the node can be de-allocated (no hazard pointer points to it)
    - If yes, the node is returned in the  $rlist$  to be checked again next time.



# Hazard Pointers

## Shared and private structures used by the algorithm

```
NODE **Hp[n][K];
// initialize each element of HP to some
// pointer address using newcell() (the
// first K such pointers are the hazard
// pointers of p0, etc.) Each of these
// memory cells contain NULL, initially.
```

```
// persistent private variables
int rcounti = 0;
List rlisti; // list of retired nodes
```

```
void RetireNode(NODE *node) {
// code for process pi
  push(rlisti, node);
  rcounti++;
  if (rcounti >= R) Scan();
}
```

```
void Scan(void) { // code for process pi
// Stage 1: Scan HP list and insert
// non-null values in plist
  init(plist); // plist is a temporary list
  for (i = 0; i < n; i++) {
    for (j = 0; j < K; j++) {
      hptr = *HP[i][j];
      if (hptr != NULL)
        insert(plist, hptr);
    }
  }
  // 2nd stage: search plist
  move all items of rlisti to tmlist;
  rcounti = 0;
  node = pop(tmlist);
  while (node != NULL) {
    if (lookup(plist, node)) {
      push(rlisti, node);
      rcounti++;
    } else PrepareForReuse(node);
  }
  node = pop(tmlist);
}
```

p <sub>0</sub>	2024
	2058
p <sub>1</sub>	3056
	3060
p <sub>2</sub>	1128
	1132
3060	NULL
3056	NULL
2052	NULL
2024	NULL
1132	NULL
1128	NULL

# Bibliography

These slides are based on material that appears in the following paper:

- Maged M. Michael, "Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects", IEEE Trans. Parallel Distrib. Syst. 15(6): 491-504 (2004)

# End of Section



Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

# Financing

- The present educational material has been developed as part of the educational work of the instructor.
- The project "Open Academic Courses of the University of Crete" has only financed the reform of the educational material.
- The project is implemented under the operational program "Education and Lifelong Learning" and funded by the European Union (European Social Fund) and National Resources



**Notes**

# Licensing Note

- The current material is available under the Creative Commons Attribution-NonCommercial-NoDerivs 4.0[1] International license or later International Edition. The individual works of third parties are excluded, e.g. photographs, diagrams etc. They are contained therein and covered under their conditions of use in the section «Use of Third Parties Work Note».



[1] <http://creativecommons.org/licenses/by-nc-nd/4.0/>

- As Non-Commercial is defined the use that:
  - Does not involve direct or indirect financial benefits from the use of the work for the distributor of the work and the license holder
  - Does not include financial transaction as a condition for the use or access to the work
  - Does not confer to the distributor and license holder of the work indirect financial benefit (e.g. advertisements) from the viewing of the work on website
- The copyright holder may give to the license holder a separate license to use the work for commercial use, if requested.

# Reference Note

Copyright University of Crete , Panagiota Fatourou 2015. Panagiota Fatourou. «Distributed Computing. Section 9: Safe Memory Reclamation». Edition: 1.0. Heraklion 2015. Available at: <https://opencourses.uoc.gr/courses/course/view.php?id=359>.

# Preservation Notices

Any reproduction or adaptation of the material should include:

- the Reference Note
- the Licensing Note
- the declaration of Notices Preservation
- the Use of Third Parties Work Note (if is available)

together with the accompanied URLs.