HELLENIC REPUBLIC
UNIVERSITY OF CRETE

Distributed Computing

Graduate Course

Section 8: Concurrent Pools

Panagiota Fatourou
Department of Computer Science

Concurrent Pools

Definition

- A **pool** is an object that supports two atomic operations:
  - **set**(): puts an item in the pool
  - **get**(): removes and returns one of the items of the pool
- A pool may be bounded or unbounded.
  - **Bounded** Pool: holds a limited number of items. This number is called its capacity.
  - **Unbounded** Pool: can hold any number of items.

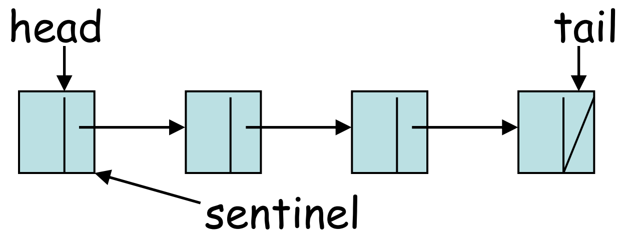
Applications

- producer-consumer type of applications
  - jobs to perform
  - keystrokes to interpret
  - purchase orders to execute
  - packets to decode, etc.

Concurrent Pools

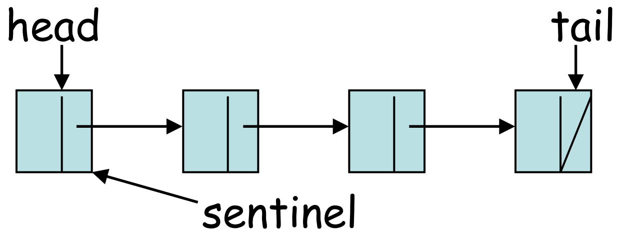
- Pool methods can be total, partial, synchronous.
  - **Total**: calls do not wait for certain conditions to become TRUE; so, get() (set()) returns an error code when the pool is empty (full)
  - **Partial**: calls may wait for conditions to hold.
  - **Synchronous**: waits for another method to overlap its call interval (i.e., threads rendezvous to exchange information).
- Fairness
  - FIFO -> queue
  - LIFO -> stack
  - Weaker properties

An Unbounded Total Queue



- A sequential queue of type T is an ordered sequence of items of type T.
- Methods it provides:
  - enq(T x): puts item x at one end of the queue, called the tail (enq() implements put());
  - deq() -> T: removes and returns the item at the other end of the queue, called the head (deq() implements get()).
- One of the nodes of the queue plays always the role of a sentinel node.
- A deq() method returns an error message if the queue is empty.
- We use two locks, EnqLock and DeqLock, to ensure that at most one enqueuer, and at most one dequeuer at a time can manipulate the nodes of the queue.

An Unbounded Total Queue



```
typedef struct node {
   T value;
   struct node *next;
} NODE;

shared NODE * Head, *Tail;
```

// initially, there is a sentinel node in the queue where
// Head and Tail point to.
// the sentinel is not always the same node
// initially, HeadL = TailL = FREE

```
void enq(T x) {
    NODE *n = new(NODE);
    n->value = x;
    n->next = NULL;
    lock(TailL);
    Tail->next = n;
    Tail = n;
    unlock(TailL);
}
```

```
T deq(void) {
    T result;
    lock(HeadL);
    if (Head->next == NULL)
        result = EMPTY_QUEUE;
    else {
        result = Head->next->value;
        Head = Head->next;
    }
    unlock(HeadL);
    return result;
}
```

An Unbounded Total Queue

- The algorithm cannot deadlock since each process acquires only one lock.
- An item is actually enqueued, when the enq() call sets the last node's next field to the new node, even before enq() resets Tail to point to the new node.
- The first and last nodes of the queue are not necessarily those pointed to by Head and Tail, respectively!
  - the actual first node is the successor of the node pointed to by Head.
  - the actual last node is the last item reachable from the Head.

An Unbounded Lock-Free Queue

```
typedef struct node {
      T value ;
    struct node *next ;
 } NODE;

typedef struct queue {
    NODE *Head ;
    NODE *Tail ;
} QUEUE ;

 void init(QUEUE *Q) {
     NODE *p = new(NODE) ;
                   // sentinel node
     p->next = NULL;
     Q->Head = Q->Tail = p ;
}
```

- The first node in the queue is a sentinel node, whose value is meaningless.
- Init() is called once before the beginning of the execution (e.g., by the system).
- Enq() is lazy: it takes steps in two distinct steps.
- Threads may need to help one another to ensure lock-freedom.

An Unbounded Lock-Free Queue

Short Description of enq():

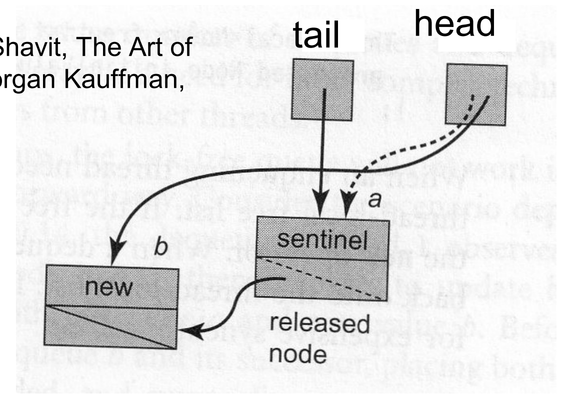
1. Create a new node and initialize it (lines 1-3)
2. Locate the last node in the queue (lines 5-6)
3. Call CAS() to append the new node (line 11)
4. Call CAS() to change the queue's Tail from the prior last node to the current last node (line 12).

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

1.    NODE *p = newcell(NODE) ;
2.    p->value = value ;
3.    p->next = NULL;

4.     while (TRUE) {                        // keep trying until enq() is done
5.         last = Q->Tail ;                               // read Tail
6.         next = last->next ;               // read next node of last
7.         if (last == Q->Tail) {            // are last and next consistent?
8.             if (next == NULL) {           // was Tail pointing to the last node?
9.                 if (CAS(last->next , next , p)   // try to link new node at the end of the list
10.                    break ;
               }
11.            else CAS(Q->Tail, last, next ) ; // tail was not pointing to last node; try to advance
         } // if
     } // while
12.   CAS(Q->Tail, last, p );        // equeue is done -> try to swing Tail to the inserted node
} // Enqueue
```

An Unbounded Lock-Free Queue

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

13.   while (TRUE)                         //  keep trying until deq() is done
14.      first = Q->Head;                  // read Head
15.      last = Q->Tail;                   // read Tail
16.      next = first->next;
17.      if (first == Q->Head) {           // are first and next still consistent?
18.         if (first == last) {           // is queue empty or Tail falling behind?
19.            if (next == NULL)           // is queue empty?
20.               return FALSE;
21.            CAS(Q->Tail, last, next);   // tail is falling behind -> try to advance it
            }
22.         else {                         // no need to deal with Tail
23.            *pvalue = next->value;      // read value to return
24.            if (CAS(Q->Head, first, next))   // try to swing Head to the next node
25.               break;
            } // else                      //   deq() is done, so exit loop
         }  // if
      }  // while
26.  return TRUE;                          // queue was not empty, so deq() suceeded
} // Dequeue
```

Figure 10.13: M. Herlihy and N. Shavit, The Art of Multiprocessor Programming, Morgan Kauffman, 2008



An Unbounded Lock-Free Queue

- An enq() is linearized at the point where the CAS of line 9 is successfully executed (by the initiator of enq() or by any of its helpers).
- A deq() is linearized as follows:
  - If it returns a value, it is linearized when it performs a successful CAS at line 24;
  - otherwise, it is linearized at line 16.
- **Lemma 1:** The linearization point of each operation (enq() or deq()) is within the execution interval of the operation.
- **Proof:** It follows by the definition of the linearization points.

An Unbounded Lock-Free Queue

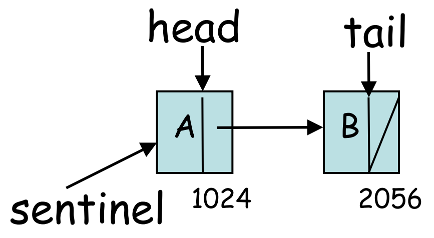
- **Lemma 2:** The following properties hold:
  1. The list is always connected.
  2. Nodes are only inserted after the last node in the list.
  3. Nodes are only deleted from the beginning of the list.
  4. Head always points to the first node in the list.
  5. Tail always point to a node in the linked list.
- **Proof:** By induction on the number of steps performed. Left as an exersice!

- **Lemma 3:** The sequential execution defined by the linearization points is legal.
- **Intuition for the Proof:** The linearization point of an operation reflects the point at which the operation takes effect.
- The queue variables always reflect the state of the queue; they never enter a transient state in which the state of the queue can be mistaken.
- **Theorem 4:** The above algorithm is a linearizable lock-free implementation of an unbounded queue.

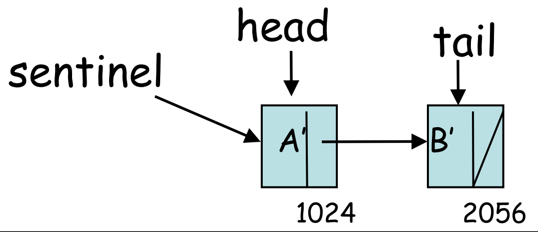
The ABA Problem

Thread 0	Thread 1

```
boolean deq(Queue *Q, int *pvalue) {
   Node *first, *last, *next;
   while (1) {
      first = Q->Head;
      last = Q->Tail;
      next = first->next;
      if (first == Q->Head) {
         if (first == last) {
               if (next == NULL)
                              return FALSE;
               CAS(Q->Tail, last, next);
         }
         else {

            *pvalue = next->value;  → B







               if (CAS(Q->Head, first, next)) break;
      }
      return TRUE;  → INCORRECT!
```



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



An Unbounded Lock-Free Queue Implementation

```
typedef struct
    pointer_with_counter {
     struct node *ptr;
     unsigned int count;
} PointerWithCounter;

struct node {
     T value;
     PointerWithCounter next;
} Node;
```

```
struct queue {
     PointerWithCounter Head;
     PointerWithCounter Tail;
} Queue;

void Initialize(Queue *Q) {
        Node  *p = new(Node);
        p->next.count = 0;
        p->next.prt = NULL;
        Q->Head = Q->Tail = <p,0>;
}
```

An Unbounded Lock-Free Queue Implementation

```
typedef struct
   pointer_with_counter {
              struct node *ptr;
              unsigned int count;
} PointerWithCounter;

struct node {
              int value;
              PointerWithCounter next;
} Node;
```

```
void Enqueue(Queue *Q, T value) {
5    Node *p = new(Node);         // allocate a new node
6    PointerWithCounter next, last;
7    p->value = value;
8    p->next = <NULL,0>;
9    while (1) {
10        last = Q->Tail; // read Tail.ptr and Tail.count together
12        next = last.ptr->next;    // read next.ptr and next.count
14        if (last == Q->Tail) {      // are last and next still consistent?
15             if (next.ptr == NULL) {             // is last pointing to the last node?
16                  if (CAS(last.ptr->next, next, <p,next.count+1>)
17                       break;                    // enqueue is done- > exit loop
             }
18             else CAS(Q->Tail, last, <next.ptr, last.count+1>);
                  // Tail was not pointing to the last node->try to advance
        } // if
    }  // while
19 CAS(Q->Tail, last, <p, last.count+1>);
     // enqueue is done -> swing Tail to the inserted node
}  // Enqueue
```

An Unbounded Lock-Free Queue Implementation

```
typedef struct
   pointer_with_counter {
            struct node *ptr;
            unsigned int count;
} PointerWithCounter;

struct node {
            int value;
            PointerWithCounter next;
} Node;
```

```
boolean Dequeue(Queue *Q, T *pvalue) {
   PointerWithCounter first, last, next;
20   while (1) {
21      first = Q->Head;      // read Head.ptr and Head.count
22      last = Q->Tail;       // read Tail.ptr and Tail.count
23      next = first.ptr->next;   // read next.ptr and next.count
24      if (first == Q->Head) {   // are first, last, next still consistent?
25         if (first.ptr == last.ptr) {     // is queue empty or Tail falling behind?
26            if (next.ptr == NULL) return FALSE;
27               CAS(Q->Tail, last, <next.ptr, last.count+1>);   // tail is falling behind;
                                                                  // TryAdvance
         }
28       else {
29          *pvalue = next.ptr->value;                          // read before CAS
30          if (CAS(Q->Head, first, <next.ptr, first.count+1>))  // swing Head to next node
31             break;
         } // else
      }  // if
   }  // while
// free (first.ptr) -> erroneous since a process may be poised to execute line 23
// this step would then cause a segmentation fault!
33   return TRUE;
} // Dequeue
```

A Synchronous Queue Implementation
Dual Data Structures

Major Ideas - Enqueue()

- If there are no dequeue() requests in the queue:
    1. Places a reservation struct in the queue, indicating that the enqueuer is waiting for a dequeuer with which to rendezvous
    2. Spins on a flag in the reservation struct
- Later, a dequeuer that discovers the reservation struct of the enqueuer fulfills the request by withdrawing the enqueuer's item and setting the flag field of the enqueuer's reservation struct.

Dequeue() works similarly.

A Synchronous Queue Implementation
Dual Data Structures

Properties

- Waiting threads can spin on a locally cached flag
- Reservations are queued in the order they arrive, ensuring that requests are fulfilled in the same order.
- The implementation ensures linearizability, since each partial method call can be ordered when it is fulfilled.

Dual Data Structure

- The methods take effect in two stages:
  - reservation, and
  - fulfillment.

A Synchronous Queue Implementation

```
#define ITEM 0
#define RESERVATION 1

typedef struct node {
      boolean type;    // a struct may represent either an item waiting to be dequeued
                       // or a reservation waiting to be fulfilled
                       // at any point in time, all nodes of the queue have the same type
      T value;         // value of item
                       // it changes to NULL when the item is dequeued
                       // when the struct represents a dequeue reservation is NULL
                       // resets to the response when deq() occurs
       struct node *next;
} NODE;

NODE *Head, *Tail;

void InitializeSynchronousQueue(void) {  // executed only once at the beginning of the execution
     NODE * sentinel = newcell(NODE);
     sentinel->type = ITEM;
     sentinel->next = NULL;
     Head = Tail = sentinel;
}
```

A Synchronous Queue Implementation

```
void enq(T x) {
    boolean success;
    NODE *last, *first, *next;

    NODE *p = newcell(NODE);
    p->type = ITEM;  p->value = x; p->next = NULL;
    while (TRUE) {                                              // keep trying until enq() is done
        last = Tail; first = Head;                              // read Head and Tail
        if (last == first  ||  last->type == ITEM) {            // is the queue empty?
                                                                // or does it contain only items?
            next = last->next;                                  // read next node of last
            if (last == Tail) {                                 // is last and next still consistent?
                if (next != NULL)  CAS(Tail, last, next);       // try to advance Tail
                else if (CAS(last->next, next, p)) {            // try to insert the new item at the end
                        CAS(Tail, last, p);                     // try to swing Tail to the inserted node
                        while (p->value == x) noop;             // spin while no rendezvous occur
                        first = Head;                           // re-read Head
                        if (p == first->next) CAS(Head, first, p); // am I the first struct in the queue?
                        return;
                }
            }
        }
        else {                                  // the queue contains deq reservations to rendezvous
            next = first->next;                 // read next node of first
            if (last != Tail  ||  first != Head  ||  next == NULL)  // if last & first are not consistent any more
                    continue;                   // or if there are no items in the list start from scratch
            success = CAS(next->value, null, x);                // try to rendezvous
            CAS(Head, first, next);                             // try to advance Head
            if (success) return;                // if rendezvous successful return TRUE;
        }                                       // otherwise false.
    }
}
```

An Unbounded Lock-Free Stack

- A stack of type T is a collection of items of type T. The stack provides two operations push(x) and pop() satisfying the last-in-first-out property: the last item pushed is the first poped.

An Unbounded Lock-Free Stack

```
#define MIN 5
# define MAX 15

boolean TryPush(NODE *n) {
    NODE * oldTop = Top;
    n->next = oldTop;
    if (CAS(Top, oldTop, n))
            return TRUE;
    else return FALSE;
}

void push(DATA x) {
    NODE *n = newcell(NODE);
    n->value = x;
    while (TRUE) {
            if (tryPush(n)) return;
            else Backoff(MIN,MAX);
    }
}
```

```
NODE *TryPop(void) {
    NODE *oldTop = Top;
    NODE * newTop;
    if (oldTop == NULL)
            return EMPTY_STACK;
    newTop = oldTop->next;
    if (CAS(Top, oldTop, newTop))
            return oldTop;
    else return NULL;
}

T pop(DATA x) {
    NODE *rn;
    while (TRUE) {
        rn = TryPop();
        if (rn == EMPTY_STACK)
            return EMPTY_STACK;
        if (rn != NULL)
                return rn->value;
        else Backoff(MIN,MAX);
    }
}
```

An Unbounded Lock-Free Stack

- The linearization point of a push() operation is placed at the point that the successful CAS of TryPush() occurs for this push().
- The linearization point of a pop() operation that returns a value of type T is placed at the point that the successful CAS of TryPop() occurs for this Pop().
- If the Pop() returns EMPTY_CODE, its linearization point is placed at the point that it reads Top in TryPop().
- **Theorem**: The above algorithm is a linearizable, non-blocking implementation of an unbounded stack.

Elimination

- The stack implementation scales poorly:
  - operations can proceed only one after the other, ordered by successful CAS calls applied to the stack's top field.
  - Stack's top field is a source of contention.
    - exponential back-off significantly helps in solving this problem

Main Idea to make the stack parallel

- if a push() is immediately followed by a pop(), the two operations cancel out, and the stack's state does not change.
- Cause concurrent pairs of pushes and pops to cancel: threads calling push() exchange values with the threads calling pop(), without ever modifying the stack itself.
  - We then say that the two calls eliminate one another.

Elimination

- Threads eliminate one another through an EliminationArray in which threads pick random array entries to try to meet complementary calls.

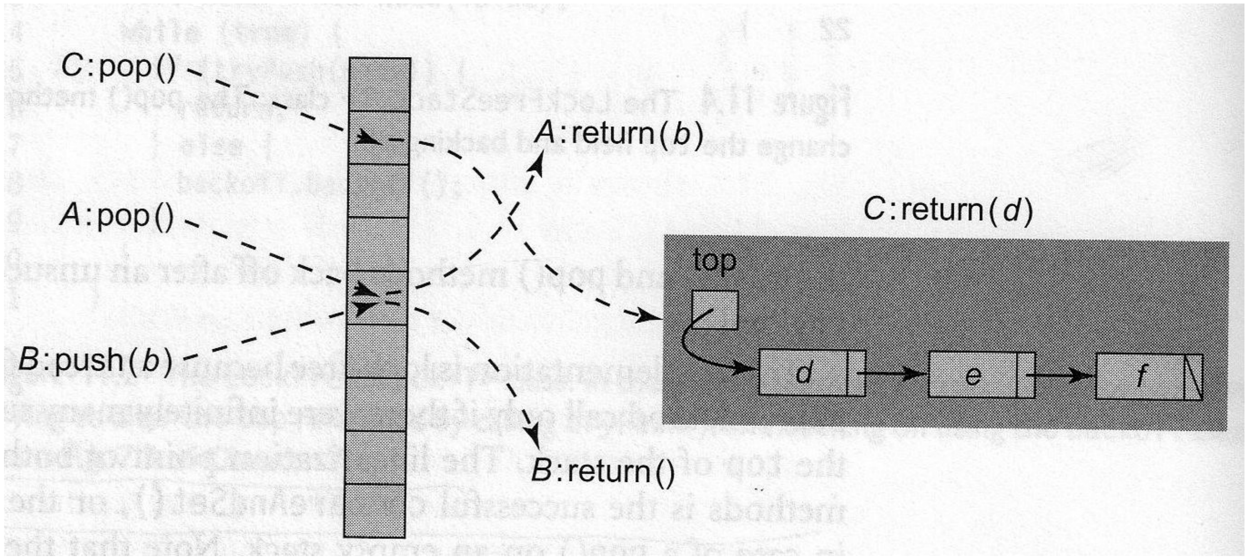


Figure 11.5: M. Herlihy and N. Shavit, The Art of Multiprocessor Programming, Morgan Kauffman, 2008

- The EliminationArray can be used as a backoff scheme on a shared lock-free stack.
  - First, each process accesses the lock-free stack.
  - If it fails to complete its operation, it attempts to eliminate it using the array instead of simply backing-off.
  - If it fails to eliminate, then it accesses the lock-free stack again.
- This structure is called Elimination-BackOff Stack.

The Elimination BackOff Stack

Main Ideas

- Allow threads with pushes and pops to coordinate and cancel out.
- Avoid a situation where an operation (push() or pop()) matches with more than one other operations.
- An exchanger is an object that allows exactly two threads to rendezvous and exchange values.
  - The first process that arrives writes its value and spins until a second process arrives.
  - The second process detects that the first is waiting, reads its value, and "signals" the exchange.
  - Both processes have now read the other's value, so they can return.
  - The call of the first process may timeout, allowing it to leave the exchanger.
- Processes should spin rather than block in the exchanger, since we expect them to wait only for very short durations.

A Lock-Free Exchanger

```
#define EMPTY  1
#define WAITING 2
#define BUSY 3

T exchange(shared <T,int> slot, T myitem, long timeout) {
    long timeBound = getnanos() + timeout;
1.   while (TRUE) {
2.        if (getnanos() > timeBound) return TIMEOUT;          // if it is time for timeout, leave the exchanger
3.        <youritem, state> = slot;
4.        switch(state) {
5.            case EMPTY:                    // try to place your item in the slot and set state to WAITING
6.                if (CAS(slot, <youritem, EMPTY>, <myitem, WAITING>)) {    // if this is done successfully
7.                        while (getnanos() < TimeBound) {                   // spin until it is time for timeout
8.                                <youritem,state> = slot;                   // read slot
9.                                if (state == BUSY) { slot = <null, EMPTY>;    // if the exchange is complete
10.                                       return youritem; }                // return the other process's item
                        }
                                                       // if no other thread shows up
11.                       if (CAS(slot, <myitem, WAITING>, <null, EMPY>)) {  // reset the state of slot to EMPTY
12.                               return TIMEOUT;         // if this is done successfully, leave the exchanger
13.                       } else {                        // some exchanger process must have shown up
14.                               <youritem, state> = slot; // complete the exchange, by reading slot,
15.                               slot = <null, EMPTY>;   // changing slot's state to EMPTY,
16.                               return youritem;        // and returning the item of the other process
                          }
                }
18.               break;
19.          case WAITING:                              // some thread is waiting and slot contains its item
20.               if (CAS(slot, <youritem, WAITING>, <myitem, BUSY>)) // replace the item with your own
21.                    return youritem;                       // and return the item of the other process
22.               break;
23.          case BUSY:                                 // two other threads are currently using the slot
24.               break;                                // the process must retry
       } // switch
    } // while
}
```

A Lock-Free Exchanger

- The exchanger implementation allows the inserted item to be null.

Linearization Points

- **For a successful exchange:** when the 2nd process to arrive changes the state from WAITING to BUSY.
  - At this point both exchange() calls overlap and the exchange is committed to be successful
- **For an unsuccessful exchange:**
  - If exchange() returns in line 2, then linearization point at return.
  - If it returns in line 12, then linearization point at the point that the CAS of line 11 is executed.
- The algorithm is lock-free because overlapping exchange() calls with sufficient time to exchange will fail only if other exchangers are repeatedly succeeding.

The Elimination Array

- The EliminationBackoffStack uses an elimination array, called exchanger[] with CAPACITY number of elements, where CAPACITY is some parameter to the algorithm.
- Each element of the elimination array is a shared variable storing a pair <T, state>, where state ∈ {EMPTY, WAITING, BUSY}.
- Each element of exchanger[] is initially <null, EMPTY>.

Code for Accessing the Elimination Array

```
T visit(T value, int range, long duration) {
    int el = randomnumber(range);
    return (exchange(exchanger[el], value, duration));
}
```

The EliminationBackOff Stack

```
void push(T x) {
    int range;
    long duration;

    NODE *nd = newcell(NODE);
    nd->value = x;

    while (TRUE) {
        if (tryPush(nd)) return;        // if you managed to push x successfully, return
        else {                          // try to use the elimination array, instead of backing-off
            range = CalculateRange();                    // choose the range parameter
            duration = CalculateDuration();              // choose the duration parameter
            otherValue = visit(x, range, duration);     // call visit with input value as argument
            if (otherValue == NULL) {    // check whether the value was exchanged with a pop() method
                RecordSuccess();         // if yes, record success
                return;
            }
            else if (otherValue == TIMEOUT)   // otherwise,
                RecordFailure();                    //  record failure
        }
    }
}
```

The Elimination BackOff Stack

```
T pop(void) {
    NODE *returnNode;
    while (TRUE) {
        returnNode = TryPop();                    // try to pop
        if (returnNode == EMPTY_STACK) // if stack is empty, return EMPTY_CODE
            return EMPTY_STACK;
        if (returnNode != NULL)                   // if tryPop() was successful, return the poped value
            return returnNode->value;
        range = CalculateRange();                 // choose the range parameter
        duration = CalculateDuration();           // choose the duration parameter
        otherValue = visit(NULL, range, duration); // call visit with input value as argument
        if (otherValue != NULL) {   // check whether the value was exchanged with a push() method
            RecordSuccess();        // if yes, record success
            return otherValue;
        }
        else if (otherValue == TIMEOUT)  // otherwise,
            RecordFailure();                      //  record failure
    }
}
```

The Elimination BackOff Stack

Range of the Elimination Array

- Smaller range -> greater chance of a successful collision when there are few threads
- Larger range -> lowers the chances of threads waiting on a busy Exchanger
- If few threads access the array, they should choose smaller ranges!
- As the number of threads increases, so should the range!

Dynamic Mechanism to control the range

- Record successful exchanges and timeout failures.
- Shrink the range as the number of failures increases and vice versa.

The Elimination BackOff Stack

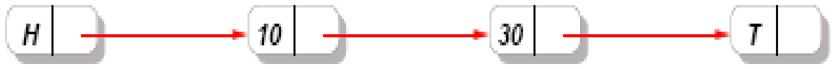
Linearizability

- Any successful push() and pop() that completes by accessing the lock-free stack can be linearized at the point of its LockFreeStack access.
- Any pair of eliminated push() and pop() can be linearized when they “collide”.
  - The operations completed through elimination do not affect the linearizability of those completed in the LockFreeStack, because they could have taken effect in any state of the LockFreeStack, and having taken effect, the state of the LockFreeStack would not have changed.

Performance

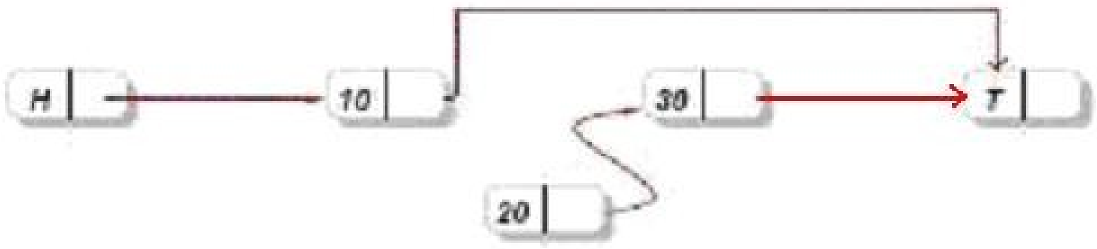
- Comparable to LockFreeStack at low loads. Why?
- What happens as the load increases?

Shared Lists

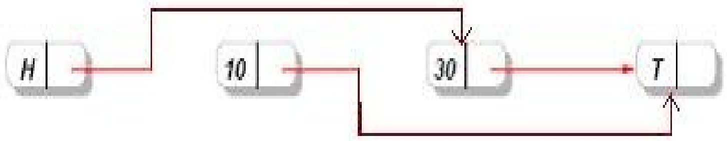


A simple sorted linked-list

Synchronization Problems when Accessing Shared Lists



Insertion of node with key 20 and concurrent deletion of node with key 30 in the list



Concurrent deletion of nodes with keys 10 and 30 from the list

Linked Lists

Coarse-Grained Synchronization

- Take a sequential implementation of the operation, add a lock, and ensure that each operation starts by acquiring the lock and ends by releasing it.

Fine-grain Synchronization

- Split the object into several independently synchronized components and ensure that operations interfere only when trying to access the same component at the same time.

Optimistic Synchronization

- Search without acquiring any locks. If the operation finds the sought-after component, it locks that component, and then validates (i.e., it checks whether the component is still reachable from the beginning of the list, and that it has not changed).
  - it is good only if validation succeeds more often than it fails.

Lazy Synchronization

- "Postpone the hard work for later": removing an element of a data structure can be split in two phases:
  - logically remove the element by marking it
  - physically remove the element by unlinking it from the rest of the data structure

Nonblocking Synchronization

- Eliminate locks entirely, relying on built-in atomic operations such as CAS or LL/SC for synchronization.

Linked Lists

- A set supports the following three operations:
  - **insert**(int key, T x): insert x of type T to the set; returns TRUE if x is inserted and FALSE if x is already in the set
  - **delete**(int key): delete key from the set; returns TRUE if key is deleted, and FALSE if key is not in the set
  - **search**(int key): search for key and return TRUE if it is in the set and FALSE otherwise.
- An operation is called successful if it returns TRUE and unsuccessful if it returns FALSE.

Linked Lists

- We implement a set as a linked list of nodes.
- The list contains regular nodes and two sentinel nodes, called **head** and **tail**, that point to the first and last element of the list, respectively.
- Sentinel nodes are never inserted or deleted; the key of head is MININT and the key of tail is MAXINT.
- The list is sorted in key order.
- Each process p uses two pointers, $curr_p$ and $pred_p$ to traverse the list; $curr_p$ points to the current node accessed in the list and $pred_p$ to its previous node.

```
typedef struct node {
            T value;
            int key;
            NODE *next;
} NODE;
shared Lock lc;                 // initially, free
shared NODE *head, *tail;
// initially, head points to a dummy node with key
// equal  to MININT and tail points to a dummy
// node with key  equal to MAXINT


boolean search(int key) {
    NODE *curr; boolean result;

    lock(lc);
    curr = head;
    while (curr->key < key)
            curr = curr->next;
    if (key == curr->key) result = TRUE;
    else result = FALSE;
    unlock(lc);
    return result;
}
```

Linked Lists – Coarse Grained Synchronization

```
boolean insert(int key, T x) {
// code for process p
    Node *pred, *curr;
    boolean result;

    lock(lc);
    pred = head;
    curr = pred->next;
    while (curr->key < key) {
            pred = curr;
            curr = curr->next;
    }
    if (key == curr->key) result = FALSE;
    else {
        NODE *node = newcell(NODE);
        node->next = curr;
        node->value = x; node->key = key;
        pred->next = node;
        result = TRUE;
    }
    unlock(lc);
    return result;
}
```

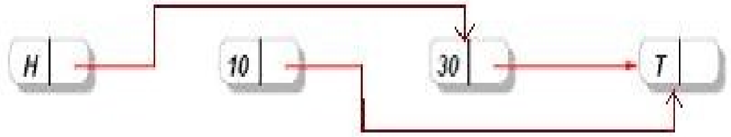
```
boolean delete(int key) {
    // code for process p
    Node *pred, *curr;
    boolean result;

    lock(lc);
     pred = head;
     curr = pred->next;
     while (curr->key < key) {
             pred = curr;
             curr = curr->next;
     }
     if (key == curr->key) {
             pred->next = curr->next;
             result = TRUE;
     }
     else result = FALSE;
     unlock(lc);
     return result;
}
```

Linked Lists – Coarse Grained Synchronization

- The algorithm is obviously correct.
  - The linearization point for each operation is placed at the point that the operation acquired the lock.
- The implementation satisfies the same progress condition as the lock implementation it employs.
- If contention is low, the performance of the algorithm is ok.
- If contention is high, the algorithm performs poorly since parallelism is restricted.
  - processes are delayed waiting for one another.

Linked Lists – Fine Grained Synchronization

- Each node is associated with its own lock.
- As a process traverses the list, it locks each node when it first visits it, and then at some later point it may release the lock.
- Locks are acquired as follows:
  - While holding the lock of the node pointed to by $pred_p$, acquire the lock of the node pointed to by $curr_p$, and then release the lock of the node pointed to by $pred_p$.
  - This is called hand-over-hand locking or lock coupling.
- It is not safe to unlock $pred_p$ before acquiring the lock to $curr_p$ since $curr_p$ may have been deleted until its lock is acquired.
- To avoid deadlocks, the locks should be acquired in the same order by each process.



Why delete() must acquire two locks.
Why hand-over-hand locking is required.

```
typedef struct node {
      T value; int key; Lock lock; NODE *next;
} NODE;

boolean search(int key) {
    NODE *curr, *pred; boolean result;

    lock(head->lock);
    pred = head;
    curr = pred->next;
    lock(curr->lock);
    while (curr->key < key) {
             unlock(pred->lock);
             pred = curr;
             curr = curr->next;
             lock(curr->lock);
    }
    if (key == curr->key) result = TRUE;
    else result = FALSE;
    unlock(pred->lock); unlock(curr->lock);
    return result;
}
```

Linked Lists – Fine Grained Synchronization

```
boolean insert(int key, T x) { // code for process p
    Node *pred, *curr; boolean result;

    lock(head->lock);
    pred = head;
    curr = pred->next;
    lock(curr->lock);
    while (curr->key < key) {
            unlock(pred->lock);
            pred = curr;
            curr = curr->next;
            lock(curr->lock);
    }
    if (key == curr->key) result = FALSE;
    else {
        NODE *node = newcell(NODE);
        node->next = curr;
        node->value = x; node->key = key;
        pred->next = node;
        result = TRUE;
    }
    unlock(pred->lock);
    unlock(curr->lock);
    return result;
}
```

```
boolean delete(int key) {
    // code for process p
    Node *pred, *curr;
    boolean result;

    lock(head->lock);
    pred = head;
    curr = pred->next;
    lock(curr->lock);
    while (curr->key < key) {
            unlock(pred->lock);
            pred = curr;
            curr = curr->next;
            lock(curr->lock);
    }
    if (key == curr->key) {
            pred->next = curr->next;
            result = TRUE;
    }
    else result = FALSE;
    unlock(pred->lock);
    unlock(curr->lock);
    return result;
}
```

Linked Lists – Fine Grained Synchronization

Linearizability

- A successful insert(k, x) is linearized when the node with the next higher key is locked.
- An unsuccessful insert(k, x) is linearized when the node with key k is locked.
- Similar rules apply for delete.
- Linearization points for search -> when the node with key k (if any) or with the next higher key (if not) is locked.

Progress

- The algorithm is starvation-free, assuming that all individual locks are starvation-free.
  - Deadlock is not possible.
  - If a process p attempts to lock head, it eventually succeeds.
  - Eventually, all locks held by other processes will be released and p will manage to lock $pred_p$ and $curr_p$.

Linked Lists – Optimistic Synchronization

Main Ideas

- Search without acquiring locks
- Lock the nodes found
- Confirm that the locked nodes are correct.
  - Use some form of validation.
  - Guarantee freedom from interference.

```
typedef struct node {
          T d;
          int key;
          Lock lock;
          NODE *next;
} NODE;
shared NODE *head, *tail;
```

```
boolean validate(NODE *pred, NODE *curr) {
   NODE *tmp = head;

   while (tmp->key <= pred->key) {
     if (tmp == pred)  {
        if (pred->next == curr) return TRUE;
        else return FALSE;
     }
     tmp = tmp->next;
   }
   return FALSE;
}
```

```
boolean search(int key) {
   NODE *curr; boolean result;

   while (TRUE) {
     pred = head; curr = pred->next;
     while (curr->key < key) {
          pred = curr; curr = curr->next;
     }
     lock(pred->lock); lock(curr->lock);
     if (validate(pred, curr) == TRUE) {
        if (key == curr->key) result = TRUE;
        else result = FALSE;
        return_flag = 1;
     }
     unlock(pred->lock); unlock(curr->lock);
     if (return_flag) return result;
   }
}
```

Linked Lists – Optimistic Synchronization

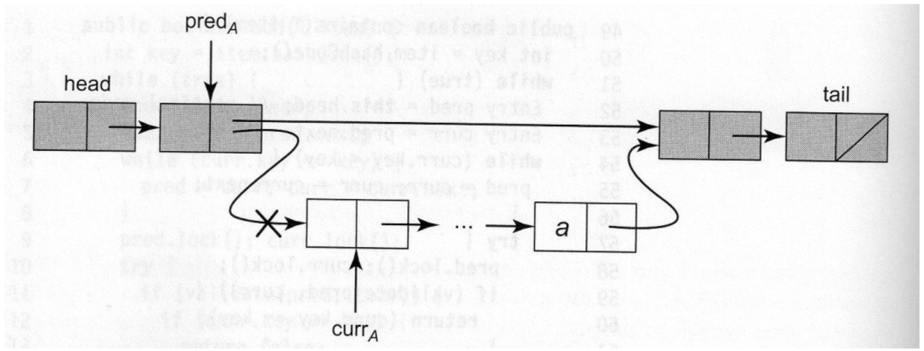
```
boolean insert(int key, T x) {      // code for process p
    Node *pred, *curr;
    boolean result;
    boolean return_flag = 0;

    while (TRUE) {
       pred = head;   curr = pred->next;
       while (curr->key < key) {
              pred = curr;
              curr = curr->next;
       }
       lock(pred->lock); lock(curr->lock);
       if (validate(pred, curr) == TRUE) {
           if (key == curr->key) {
              result = FALSE; return_flag = 1;
           }
           else {
              NODE *node = newcell(NODE);
              node->next = curr;
              node->value = x; node->key = key;
              pred->next = node;
              result = TRUE; return_flag = 1;
           }
       }
       unlock(pred->lock); unlock(curr->lock);
       if return_flag) return result;
    }
}
```

```
boolean delete(int key) {
    // code for process p
    Node *pred, *curr;
    boolean result; boolean return_flag = 0;

    while (TRUE) {
         pred = head;   curr = pred->next;
         while (curr->key < key) {
              pred = curr;
              curr = curr->next;
         }
         lock(pred->lock); lock(curr->lock);
         if (validate(pred, curr)) {
              if (key == curr->key) {
                   pred->next = curr->next;
                   result = TRUE;
              }
              else result = FALSE;
              return_flag = 1;
         }
         unlock(pred->lock); unlock(curr->lock);
         if (return_flag == 1) return result;
    }
}
```

Linked Lists – Optimistic Synchronization

- Validation is necessary.
- Each operation may traverse nodes that have been deleted from the list.
  If a process follows the next fields of deleted nodes, it will eventually return to some node of the list.
- Garbage collection should be done with care.
- The algorithm is not starvation-free, even if all nodes' locks are starvation-free.
- The implementation works well if the cost of traversing the list twice without locking is significantly less than the cost of traversing the list once with locking.
- Search() requires to get locks ☹

Figure 9.15: M. Herlihy and N. Shavit, The Art of Multiprocessor Programming, Morgan Kauffman, 2008



Linked Lists – Lazy Synchronization

Main Ideas

- We add in each node a boolean marked field which indicates whether this node is in the set.
- Traversals do not lock and do not validate.
- Insert() locks the target's predecessor and adds the new node.
- Delete is realized in two steps:
  - mark the node as deleted
  - physically remove the node

```
boolean validate(NODE *pred, NODE *curr) {
    if (pred->marked == FALSE &&
        curr->marked === FALSE &&
        pred->next == curr) return TRUE;
    else return FALSE;

}
```

```
typedef struct node {
            T d;
            int key;
            boolean marked;
            Lock lock;
            NODE *next;
} NODE;
shared NODE *head, *tail;
```

```
boolean search(int key) {
    NODE *curr;
    boolean result;

    curr = head;
    while (curr->key < key)
            curr = curr->next;
    if (curr->marked !=TRUE
                && key==curr->key)
        return TRUE;
    else return FALSE;
}
```

Linked Lists – Lazy Synchronization

```
boolean insert(int key, T x) {      // code for process p
    Node *pred, *curr;
    boolean result;
    boolean return_flag = 0;

    while (TRUE) {
       pred = head;   curr = pred->next;
       while (curr->key < key) {
              pred = curr;
              curr = curr->next;
       }
       lock(pred->lock); lock(curr->lock);
       if (validate(pred, curr) == TRUE) {
           if (key == curr->key) {
              result = FALSE; return_flag = 1;
           }
           else {
              NODE *node = newcell(NODE);
              node->next = curr;
              node->value = x; node->key = key;
              pred->next = node;
              result = TRUE; return_flag = 1;
           }
       }
       unlock(pred->lock); unlock(curr->lock);
       if return_flag) return result;
    }
}
```

```
boolean delete(int key) {
    // code for process p
    Node *pred, *curr;
    boolean result; boolean return_flag = 0;

    while (TRUE) {
         pred = head;   curr = pred->next;
         while (curr->key < key) {
              pred = curr;
              curr = curr->next;
         }
         lock(pred->lock); lock(curr->lock);
         if (validate(pred, curr)) {
              if (key == curr->key) {
                  curr->marked = TRUE;
                  pred->next = curr->next;
                  result = TRUE;
              }
              else result = FALSE;
              return_flag = 1;
         }
         unlock(pred->lock); unlock(curr->lock);
         if (return_flag == 1) return result;
    }
}
```

Linked Lists – Lazy Synchronization

- We say that an item is in the set, if, and only if it is referred to by an unmarked reachable node.
- Lemma: Any unmarked reachable node remains reachable even if its predecessor is logically or physically deleted.
- Insert() and delete() are not starvation-free since list traversals may be arbitrarily delayed by ongoing modifications.

Linearization Points

- Insert():
  - Successful: when pred->next changes to point to node.
  - Unsuccessful: at the point that it acquires the lock to curr for the last time.
- Delete()
  - Successful: when the mark is set.
  - Unsuccessful: at the point that it acquires the lock to curr for the last time.
- Search()
  - Successful: when an unmarked matching node is found.
  - Unsuccessful: Can we linearize an unsuccessful search when it detects that the node it is looking for is marked?

Linked Lists – Lazy Synchronization

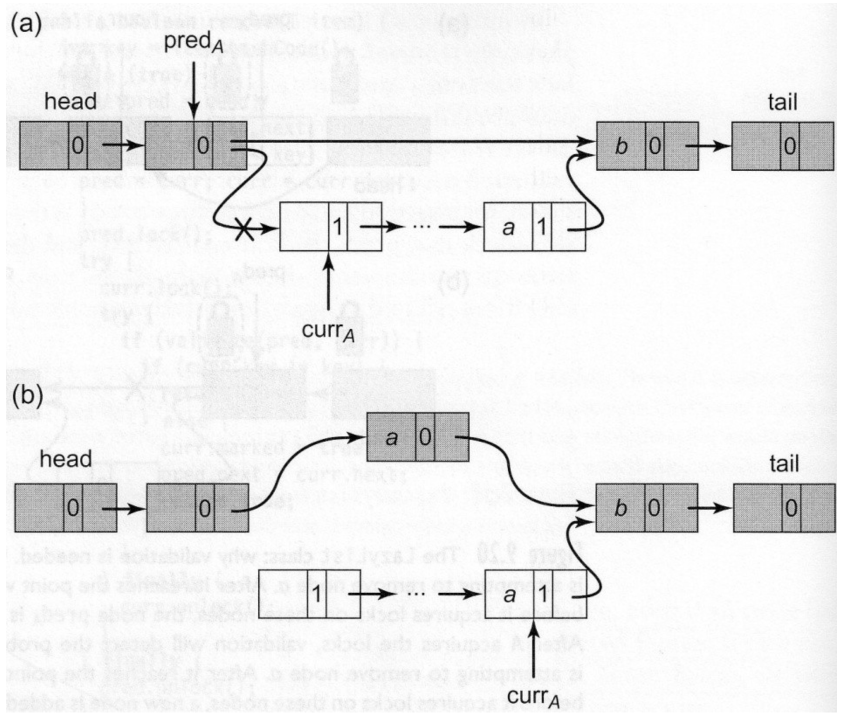


Figure 9.21: M. Herlihy and N. Shavit, The Art of Multiprocessor Programming, Morgan Kauffman, 2008

An unsuccessful search() is linearized at the earlier of the following points:

1. the point where a removed matching node, or a node with key greater than the one being searched is found, and
2. the point immediately before a new matching node is inserted to the list.

Linked Lists – Non-blocking Synchronization

Main Ideas

- The node's next and marked fields are treated as a single atomic unit: any attempt to update the next field when the marked field is TRUE will fail.
- Implement the marked field by "stealing" a bit from the next pointer.

```
typedef struct node {
            int key;
              struct node *next;
} NODE;

NODE *head *tail;

void InitializeList() {
        head = newcell (NODE);
        tail = newcell (NODE);
        head->next = tail;
        head->key = MININT;
        tail->next = NULL;
        tail->key = MAXINT;
}
```

get_unmarked_reference(NODE *r)
// checks if r is marked. If yes, it returns
// the unmarked version f r. If no, it returns r

get_marked_reference(NODE *r)
// checks if r is marked. If yes, it returns r;
// otherwise, it returns the marked version of r.

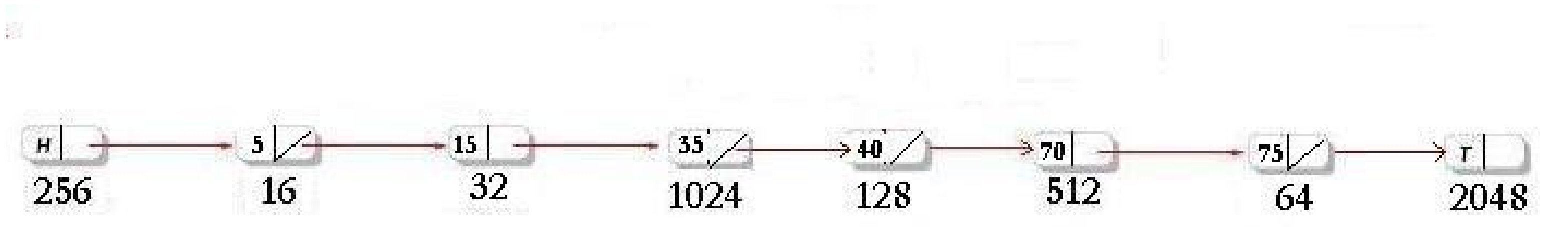
is_marked_reference(NODE *r)
// returns TRUE if r is marked; FALSE otherwise

➧ None of these routines changes the value of the pointer!

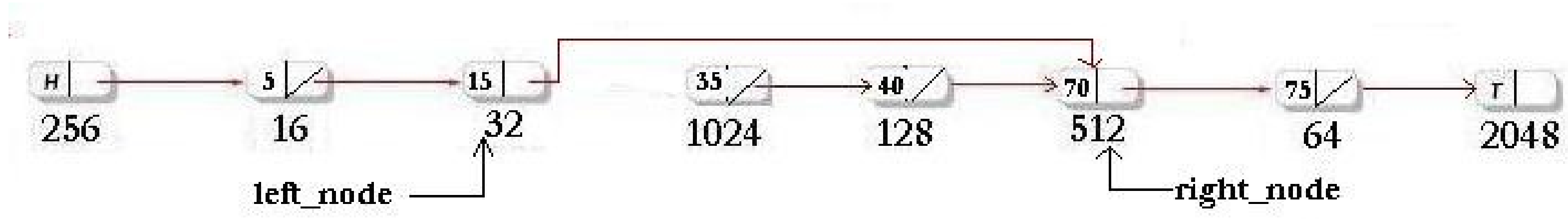
Non-blocking Synchronization

```
NODE *search (int search_key, NODE **left_node) {
NODE *left_node_next, *right_node;
search_again:
        do {
            NODE *t = head;
            NODE *t_next = head->next;
            do {
                if (!is_marked_ref(t_next)) {
                    (*left_node) = t;
                    left_node_next = t_next;
                }
                  t = get_unmarked_ref(t_next);
                  if (t == tail) break;
                  t_next = t->next;
            } while (is_marked_ref (t_next) || (t->key < search_key));        /*B1*/

            right_node = t;
            if (left_node_next == right_node) {        // notice that if left_node_next were marked,
                                                       // this condition would evaluate to FALSE
                if (is_marked_ref (right_node->next))
                     goto search_again;                                        /* G1 */
                else return right_node;                                        /* R1 */
            }
            if (CAS (&(left_node->next), left_node_next, right_node)) {       /* C1 */
                          // notice that if left_node_next were marked, this CAS would be unsuccessful
                if (is_marked_ref(right_node->next))
                       goto search_again;                                      /* G2 */
                else return right_node;                                        /* R2 */
            } // if
        } while (TRUE);                                                        /* B2 */
} // search
```

Non-blocking Synchronization



List where nodes with keys 5, 35, 40, 75 are marked



The previous list after the execution of search(70).

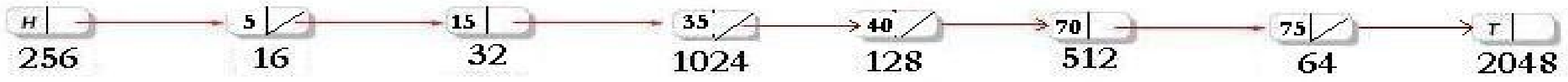
Non-blocking Synchronization

- The search() ensures that the following conditions hold for left_node and right_node:
  1. the key of the left_node must be less than the search key and the key of the right_node must be greater than or equal to the search key.
  2. left_node and right_node must be unmarked
  3. right_node must be the immediate successor of left_node.
- Condition 1 holds since otherwise the search() would have ended earlier.
- To show that the other two conditions hold, we consider the following cases:
  - Search() returns on line 15: The conditions were TRUE when line 9 was executed.
  - Search() returns on line 20: The conditions were TRUE when line 17 was executed.

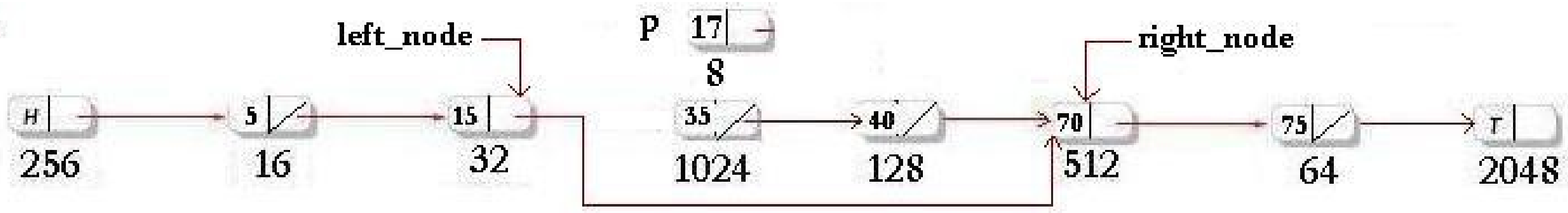
Non-blocking Synchronization

```
boolean ListInsert (int key)
{
(21)      NODE *new = newcell(NODE);  new->key = key;
(22)      NODE *right_node, *left_node;
         do {
(23)              right_node = search (key, &left_node);
(24)              if ((right_node != tail) && (right_node->key == key))
(25)                      return FALSE;
(26)              new_node->next = right_node;
(27)              if (CAS (&(left_node->next), right_node, new_node))
(28)                      return TRUE;
(29)      } while (TRUE);
}
```

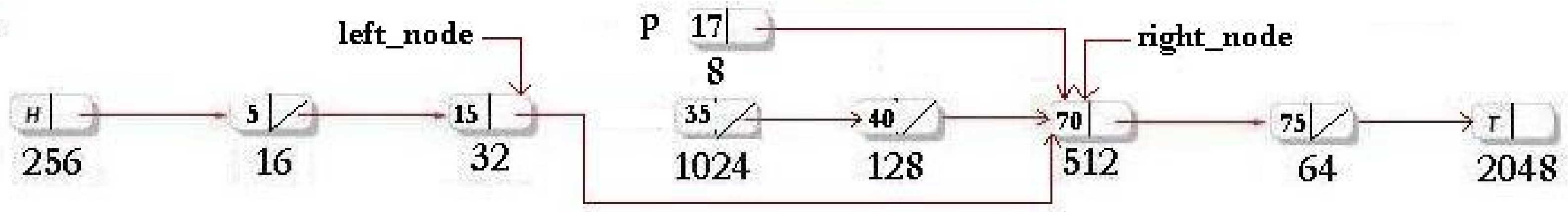
Non-blocking Synchronization



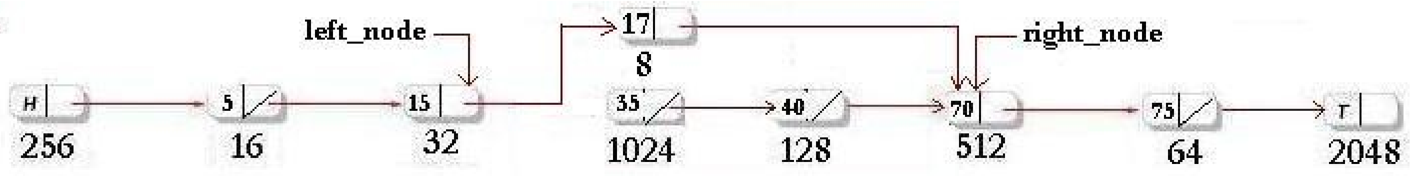
List where nodes with keys 5, 35, 40, 75 are marked



List after the execution of search (line 3).



List after the execution of line 6 of insert(17).



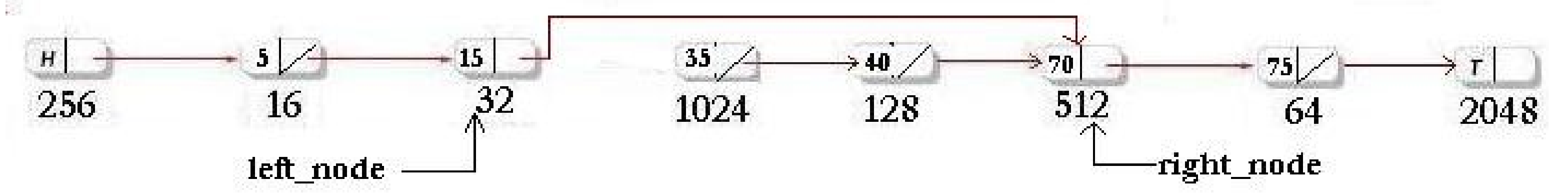
List after the completion of insert(17).

Non-blocking Synchronization

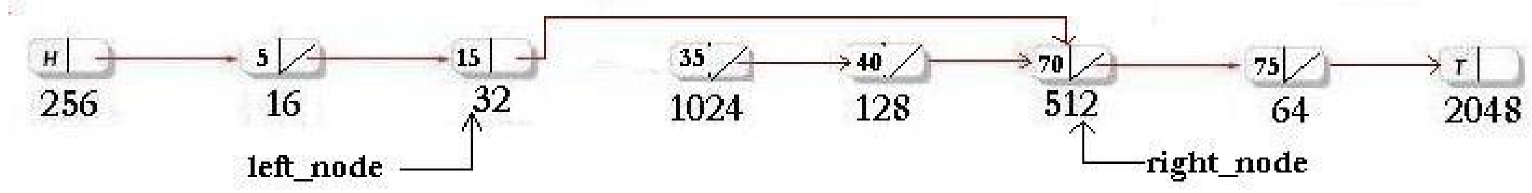
```
boolean ListDelete (int key)
 {
    NODE *right_node, *right_node_next, *left_node;
   do {
            right_node = search (search_key, &left_node);
            if ((right_node == tail) || (right_node->key != search_key))
                  return false;
            right_node_next = right_node->next;
            if (!is_marked_reference(right_node_next))
                 if(CAS(&(right_node.next),right_node_next,
                        get_marked_reference(right_node_next)))
                   break;
    } while (true);
   if (!CAS (&(left_node->next), right_node, right_node_next))
       right_node = search (right_node->key, &left_node);
   return TRUE;
}
```

•An item is in the set if and only if it is an unmarked reachable node.

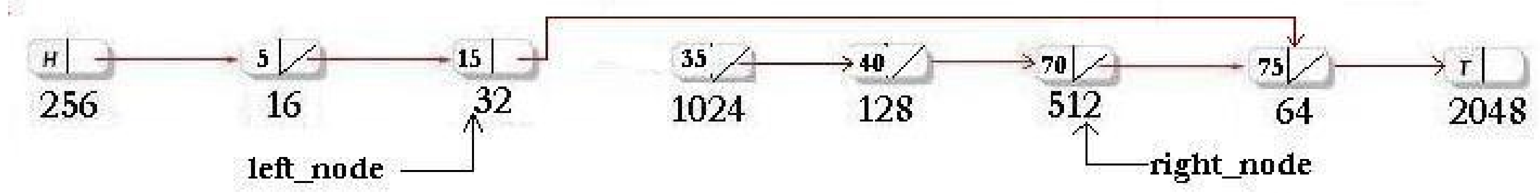
Non-blocking Synchronization



List after the execution of line 2 of delete



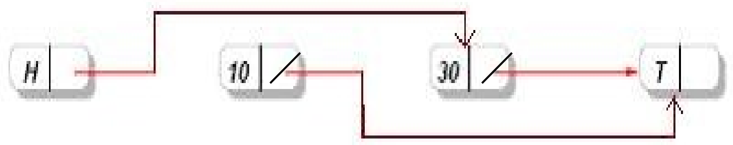
List after the marking of the node

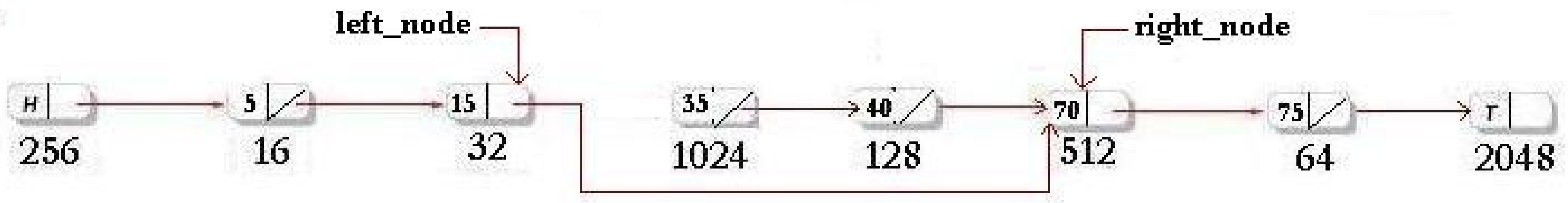


List after the physical deletion of the node

Non-blocking Synchronization

- As each thread traverses the list, it cleans up the list by physically removing any marked nodes it encounters.





Non-blocking Synchronization

Linearization Points

- Let $op_{i,m}$ be the mth operation by p_i, and let $d_{i,m}$ the last configuration at which the conditions of search() are satisfied during the execution of $op_{i,m}$.

```
boolean ListFind(int key) {
(42)    NODE *right_node, *left_node;
(43)   right_node = search(key, &left_node);
(44)     if (right_node == tail ||
                              right_node->key != key)
(45)           return FALSE;
(46)     else return TRUE;
}
```

- If $op_{i,m}$ is a Find() or an unsuccessful Insert() or Delete(), we linearize it at $d_{i,m}$.
  - Successful Find and Unsuccessful Insert: at that point, the right node was unmarked and contained the search key.
  - Unsuccessful Find or Unsuccessful Delete: at that point the left and right nodes were unmarked and contained keys strictly-less than and strictly-greater than the search key.
- If $op_{i,m}$ is a successful update:
  - Let $u_{i,m}$ be the configuration at which the CAS of $op_{i,m}$ inserts a node or $op_{i,m}$ logically deletes a node. We insert the linearization point of $op_{i,m}$ at this configuration.

Bibliography

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

- M. Herlihy and N. Shavit, The Art of Multiprocessor Programming, Morgan Kauffman, 2008 (Chapters 9, 10, 11)
- Timothy L. Harris, "A Pragmatic Implementation of Non-blocking Linked-Lists", 15th International Symposium on DIStributed Computing (DISC'01), pp. 300-314, 2001.

End of Section

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

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο

ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ ΚΑΙ ΘΡΗΣΚΕΥΜΑΤΩΝ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ

ΕΣΠΑ
2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ

Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

Notes

Licensing Note

Reference Note

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.