IBM T. J. Watson Research Center

Practical Tradeoffs In Non-Blocking Design

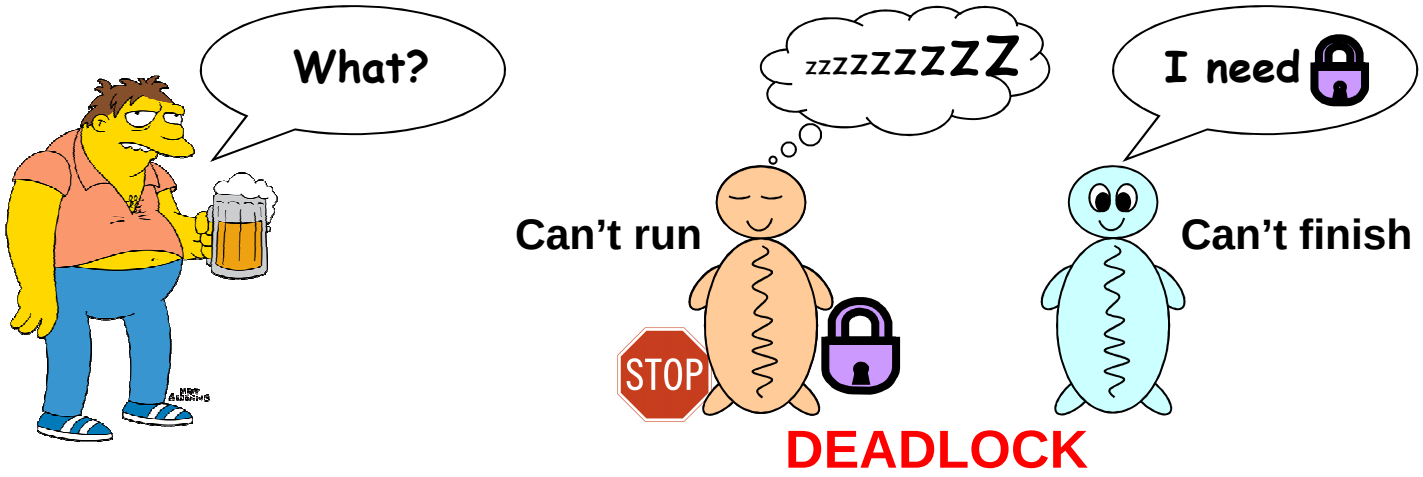
Maged Michael
IBM T J Watson Research Center

SRDC 2013, Crete, 10 June 2013

Common Uses of Non-Blocking Operations

- **Soft Real-time Applications**
  - Common in media players
  - Non-blocking operations are used to reduce the likelihood of noticeable delays
  - For preemption tolerance and to prevent priority inversion
- **Kill-Safe Systems**
  - In server systems with processes representing client requests, the ability to terminate processes asynchronously can allow high server throughput.
  - Non-blocking operations are used to prevent deadlock
- **Async Signal-Safety**
  - In some cases it is desirable to share data between signal handlers and the threads that may be interrupted asynchronously.
  - Non-blocking operations are used to prevent deadlock

Deadlock Due to Blocking in Signal Handling



- A thread acquires a lock to operate on some shared data
- The scheduler interrupts the thread to deliver a signal
- The signal handler runs
- The signal handler needs to acquire the lock

The interrupted thread will not run until the signal handler completes

The signal handler will not complete until the interrupted thread releases the lock

Non-Blocking Features and Design Issues

Progress Guarantees

- **Which operations need to be non-blocking?**
  - **Async signal safety:**
    - Only the signal handler operations need to be non-blocking.
    - The operations of interrupted threads can be blocking
    - E.g., [Heller et al 2005] List: Wait-free lookup and blocking add and remove.
  - **Soft Real-time:**
    - Only high-priority operations need to be non-blocking.
    - Low priority operations can be blocking

Progress Guarantees

- **Level of Progress Guarantee**
  - **Async signal safety:**
    - Signal handler operations need to be obstruction-free
  - **Kill-safety:**
    - Obstruction-free is sufficient to avoid deadlock due to termination
    - Lock-free is desirable to avoid live-lock among active processes of long delays
  - **Soft real-time:**
    - Ideally, wait-free to bound the number of steps
    - Lock-free and obstruction-free are acceptable to reduce the likelihood of long delays

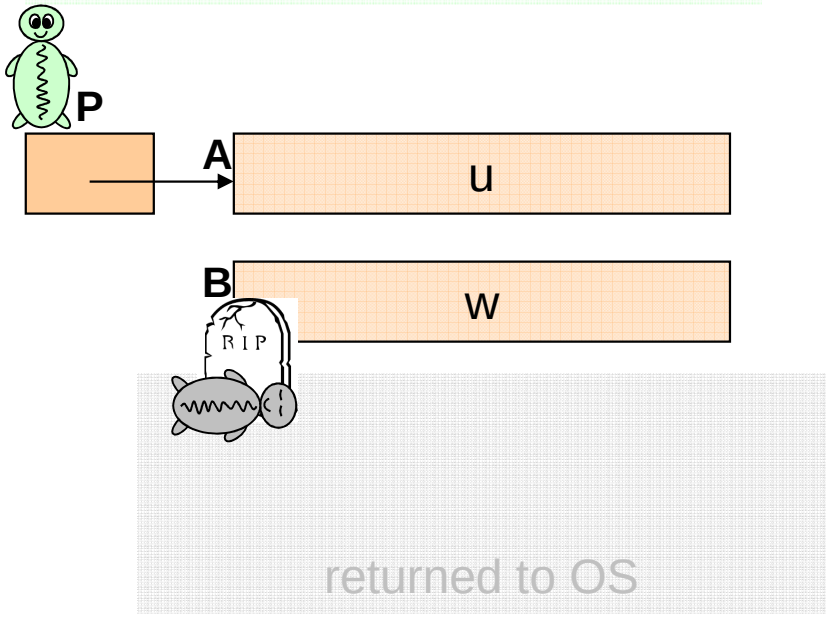
Progress Guarantees

- **Inter-operation progress guarantees**
  - E.g., it might be desirable in a search structure to prevent the starvation of lock-free add operation due to conflict with lookup operations
  - I.e., add operations may be lock-free with respect to each other and other update operations, but they are wait-free with respect to lookup operations.

The Memory Reclamation Problem

Example: **Large CAS(u,v)**

```
1  q := P
3  r := (*q == *u) // non-atomic
   if r
      n := new Block(v)
      r := CAS(P,q,n)
      delete  r ? q : n
```



1. Thread i reads pointer value A from P
2. Thread j sets P to B and frees A to OS
3. Thread i accesses free memory

ACCESS VIOLATION

The Memory Reclamation Problem

- A thread i reads a pointer to a dynamic memory location
- Another thread j removes the block and frees it
- Thread i dereferences the pointer to access the freed block
  - Thread i might read/write unmapped memory

    → **access violation**
  - Thread i might read unrelated data from the recycled block

    → **return incorrect result**
  - Thread i might write into the recycled node

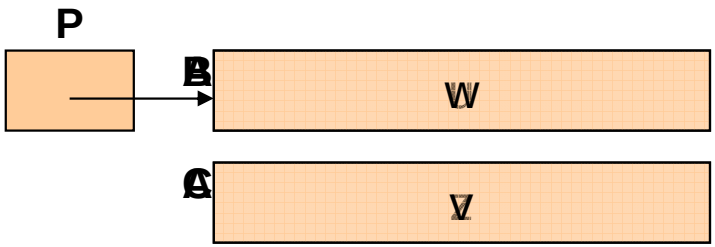
    → **corrupt some shared structure**

- **How to be able to reclaim dynamic memory blocks removed from non-blocking structures and guarantee that no thread will access the contents of free blocks?**

The ABA Problem

Example: **Large CAS(u,v)**

```
1  q := P
2  r := (*q == *u) // non-atomic
   if r
6     n := new Block(v)
7     r := CAS(P,q,n)
      delete  r ? q : n
```



1. Thread i reads A from P
2. Thread i reads the value of *u from *A
3. Thread j sets P to B
4. Thread j reuses block A to hold value z
5. Thread j sets P to A again
6. Thread i allocates block C to hold value v
7. Thread i checks that P is equal to A
   CAS succeeds although *P == z != *u

INCORRECT OUTCOME

Memory Reuse / ABA Prevention Constraints

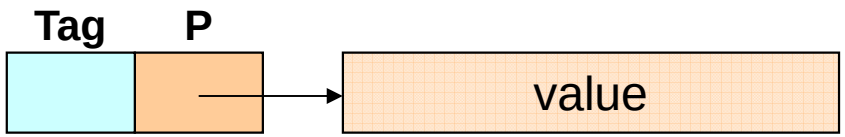
- **Object lifetime**
  - Object can be reused or reclaimed after deletion
  - Or object may be moved between structures and reinserted without going through reclamation
- **Levels of memory reuse:**
  - Type-preserving reuse only.
  - Memory can be coalesced or divided for reuse.
  - Memory can also be returned to the operating system.

ABA-Prevention Tags

If a value is susceptible to the ABA problem a tag is co-located with it in a block that can be accessed atomically.

The tag is read and updated atomically with the value to prevent ABA cases

Dynamic objects may be reused immediately but reuse may be restricted



CAS(u,v) : boolean

```
<q,t> := <P,Tag>
r := (*q == *u) // non-atomic
if r
   n := new Block(v)
   r := CAS(<P,Tag>,<q,t>,<n,t+1>)
   reclaim r ? q : n
return r
```

Read(u)

```
do
  <q,t> := <P,Tag>
  *u := *q        // non-atomic
until <P,Tag> == <q,t>
```

ABA-Prevention Tags

Pros and Cons

Reuse of reclaimed memory is restricted ***(Con)***

Hinders unmapping and coalescing

Objects can be deleted and reinserted ***(Pro)***

Operations on the tag are wait-free ***(Pro)***

But immediate reuse of memory may cause an algorithm to become lock-free

Readers do not starve writers ***(Pro)***

Immediate reclamation ***(Pro)***

Require wide atomic primitives ***(minor Con)***

Historical Digression: *40-Bit Uniqueness Value*

United States Patent [19] [11] **3,886,525**

Brown et al. [45] **May 27, 1975**

[54] **SHARED DATA CONTROLLED BY A PLURALITY OF USERS**

[75] Inventors: **Paul J. Brown**, Poughkeepsie; **Ronald M. Smith**, Wappingers Fall both of N.Y.

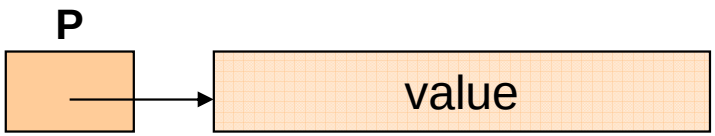
[73] Assignee: **International Business Machines Corporation**, Armonk, N.Y.

[22] Filed: **June 29, 1973**

The instruction Compare and Swap is provided in a form which updates an entire double word in storage, and as such may be used directly to update shared data areas up to a double word in length. In addition, the double word provides sufficient information space to contain two complete storage addresses, or one storage address and a 40-bit uniqueness value. This provides the program with sufficient function to safely program the control of larger shared storage areas while maintaining the possibility of an endless loop at a much lower level than in prior art techniques. This is accomplished as follows:

1. Unconditional swapping of a single address is sufficient to program a first-in, first-out single-user-at-a-time sequencing mechanism.
2. Conditional swapping of a single address is sufficient to program a last-in, first-out single-user-at-a-time sequencing mechanism.
3. Conditional swapping of an address and 40-bit uniqueness value is sufficient to program the acquisition and return of a free storage block which may be used for event control blocks, program save areas for task switching, and work areas to accomplish the task switching.

Ideal Wait-Free Garbage Collection



CAS(u,v) : boolean

```
 q := P
 r := (*q == *u) // non-atomic
 if r
    n := new Block(v)
    r := CAS(P,q,n)
return r
```

Read(u)

```
 q := P
 *u := *q        // non-atomic
```

Wait-free ***(Pro)***

Arbitrary memory reclamation ***(Pro)***

Immediate reclamation of reclaimable memory ***(Pro)***

Explicit deallocation not required ***(Pro)***

No ABA prevention on reinsertion *before reclamation* ***(Con)***

Epoch-Based Reclamation

E.g., RCU (read-copy-update) heavily-used in the Linux kernel

Depends on the notion of *quiescence points*, where a thread is guaranteed not to hold references to removable memory blocks

A removed block is reclaimed only after each thread (that could have had access to it) has gone through at least one quiescence point after the block was removed

CAS(u,v) : boolean

```
q := P
r := (*q == *u) // non-atomic
if r
   n := new Block(v)
   r := CAS(P,q,n)
   if r
     RETIRE(q)
   else
     reclaim n
return r
```

Read(u)

```
q := P
*u := *q        // non-atomic
```

Epoch-Based Reclamation

Pros and Cons

Arbitrary reuse of reclaimed memory ***(Pro)***

Wait-free readers ***(Pro)***

Writers are not really nonblocking ***(Con)***
- Reclamation may be delayed indefinitely

Readers cannot starve writers ***(Pro)***

Unbounded number of not-yet-reclaimable removed blocks ***(Con)***

No reader overhead in common path ***(Pro)***

No special primitives needed ***(Pro)***

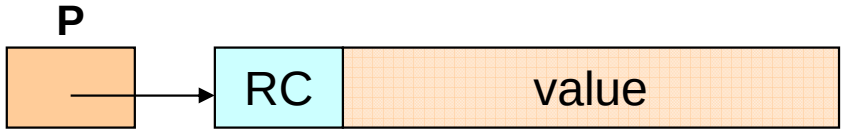
No ABA prevention on reinsertion before reclamation ***(Con)***

Reference Counting

Each dynamic object contains (or is associated with) a reference counter

Threads increment/decrement the reference counter to reflect the number of active references to the object

The dynamic objects is reclaimed only when the reference counter falls to zero



SAFE_READ(P) : Pointer to Block

```
loop
  do
    q := P
    if q == null return null
  until fetch_add_if_nonzero(q.RC,1)
  if P == q return q
  RELEASE(q)
```

RELEASE(q,c)

```
rc = subtract_and_fetch(q.RC,c)
if rc == 0
  RELEASE(q.ptr,1) for all q.ptr
  reclaim q
```

CAS(u,v) : boolean

```
q := SAFE_READ(P)
r := (*q == *u) // non-atomic
if r
   n := new Block(v)
   n.RC := 1
   r := CAS(P,q,n)
   if r
     RELEASE(q,2)
   else
     reclaim n
return r
```

Read(u)

```
q := SAFE_READ(P)
*u := *q        // non-atomic
RELEASE(q,1)
```

Reference Counting

Pros and Cons

Hinders arbitrary memory reclamation ***(Con)***

Lock-free (using CAS) ***(Pro)***

Maybe wait-free using specialized wait-free primitives

Readers write to multi-writer shared variables ***(Con)***

Readers may starve writers ***(Con)***

Immediate reclamation of reclaimable memory ***(Pro)***

Explicit deallocation not required ***(Pro)***

No ABA prevention on reinsertion before reclamation ***(Con)***

Hazard Pointers

- A hazard pointer is single-writer multi-reader pointer
- Each hazard pointer has one owner (that can write to it)
- By setting a hazard pointer to the address of a dynamic block, the owner thread is telling other threads:

  "if any of you remove this block
  after the last time I set this hazard pointer to this block
  don't reclaim this block until I change my hazard pointer"

CAS(u,v) : boolean

```
q := P
*myHP := q // non-atomic
if q != P return false
r := (*q == *u) // non-atomic
if r & *u == *v
   n := new Block(v)
   r := CAS(P,q,n)
   RETIRE r ? q : n
*myHP := null  // optional
return r
```

As long as *myHP remains equal to q
safe access: q will not be freed

no ABA: q will not be reinserted

Read – Using Hazard Pointers

Read(u)

```
do
  q := P
  *myHP := q
until q == P
*u := *q        // non-atomic
*myHP := null   // optional
```

Lock-free

Can Read be wait-free?

Yes

Hazard Pointers

Pros and Cons

Unrestricted reclamation ***(Pro)***

Wait-free / Lock-free ***(Pro)***

Readers cannot starve writers ***(Pro)***

Bounded not-yet-reclaimed space O(W*R) ***(mixed)***

O(W*R) space and O(1) time per reclaimed object

or O(R) space and O(R) time per reclaimed object

R is # of active readers and W is # active writers

No ABA prevention on reinsertion before reclamation ***(Con)***

Memory Reclamation and ABA Prevention

	ABA Tags	Epoch	RC	HP
Reclamation	Restricted*	Unrestricted	Restricted	Unrestricted
Progress	Wait-free*	Wait-free readers Blocking writers	Lock-free	Wait-free*
Writer Starvation	No	No	Yes	No
Not yet reclaimed Space	No	Unbounded	No	O(W*R)
ABA Prevention with immediate reinsertion	Yes	No	No	No
Special Primitives	CAS double	No	CAS	No

Effect of Non-Faulting Loads on Reclamation

- From the classic lock-free LIFO free list

Pop

```
do
   <p,t> := Anchor
   n := *p
until
CAS(Anchor,<p,t>,<n,t+1>)
return p
```

No memory reclamation problem
on systems with non-faulting loads

Another Historical Digressiong

The LIFO free list is likely the first nontrivial lock-free algorithm

IBM System/370
Principles of Operation

Fourth Edition (September 1974)

Choice of Data Types and Structures

- Difficulty of algorithms: E.g., trees vs. lists and hash tables for search structures
- Unnecessary semantics: E.g., FIFO vs. LIFO or arbitrary order
- Memory management effects: Dynamic structures vs. static structures with and without resizing

Other issues

- **Frequency of update or deletion**
  - A disadvantage of epoch-based mechanisms
- **Max. concurrent threads**
  - A disadvantage of hazard pointers
- **Portability / hardware primitives**
  - May be an issue in the future with HTM
  - So far, HTM architectures (except IBM System Z) are best-effort. Non-HTM paths are required.
- **Memory ordering and language features**
  - E.g. Java volatiles and C++ atomics on weak memory models

Concluding Example: Requirements

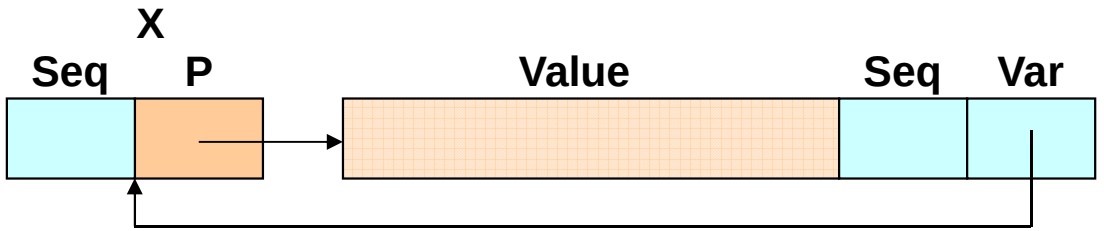
- Kill safe system
- Data records:
  - Variable-sized and possibly large
  - May be moved between structures and reinserted
- Frequent updates
- Potentially thousands of concurrent threads

Concluding Example: Solution

- Lock-free linked list structures
- Use lock-free memory allocator for the records
- Need lock-free ABA prevention and full memory reclamation for records, and need to allow immediate reinsertion of records
  - Cannot use ABA tags on the records
  - Cannot use GC-based reclamation methods on the records
  - No acceptable solution without changing the structures
- Use a level of indirection with small fixed-sized blocks and move or duplicate key fields in the small blocks. Non-blocking operations access only the small blocks
- Epoch-based reclamation is not desirable due to frequent updates
- Hazard pointers are not desirable due to the potentially large number of concurrent threads
- Option: Use ABA tags for the small blocks. No reclamation of the small blocks. Dependent on double-width atomic primitives.
- Option: Use reference counting on small blocks with copy-on-write to prevent ABA. No reclamation of small blocks.

THANK YOU

Wait-Free Read – Using Hazard Pointers



Read(u)

```
q := P
*myHP := q
if q == P
  *u := *q        // non-atomic
  *myHP := null   // optional
  return
seq := Seq
SetTrap(X,seq)
q := P
*myHP := q
b := GetCapturedBlock()
if b == null
  *u := *q
else
  *u := b.Value
*myHP := null
ReleaseTrap()
```

SetTrap(X,seq,id)

```
mytrap.Var := X
mytrap.Seq := seq
mytrap.Captured := mytag
*mytrapHP = mytag // mytrap.HP
trap[id].Valid := true
mytag := next nonpointer value
```

GetCapturedBlock() : Pointer to Block

```
b := mytrap.Captured
if b is a pointer value
   return b
return null
```

ReleaseTrap()

```
mytrap.Active := false
mytrap.Captured := null
*mytrapHP := null
```

Wait-Free CAS – Using Hazard Pointers

CAS(u,v) : boolean

```
q := P
*myHP := q
if q != P return false
r := (*q == *u) // non-atomic
if r & *u == *v
   seq := q.Seq
   n := new Block(v,seq+1,X)
   CAS(Seq,seq-1,seq)
   r := CAS(P,q,n)
   if r
      RETIRE(q)
   else
      reclaim n
*myHP := null  // optional
return r
```

ScanTraps(q)

```
mylist.add(q)
if mylist > THRESHOLD
  forall traps t
    if t.Active
      tag := t.Captured
      if tag is a pointer continue
      var := t.Var
      seq := t.Seq
      b := mylist.lookup(var,seq)
      if b != null
        if CAS(t.Captured,tag,b)
          CAS(t.HP,tag,b)
  forall p in mylist RETIRE(p)
```