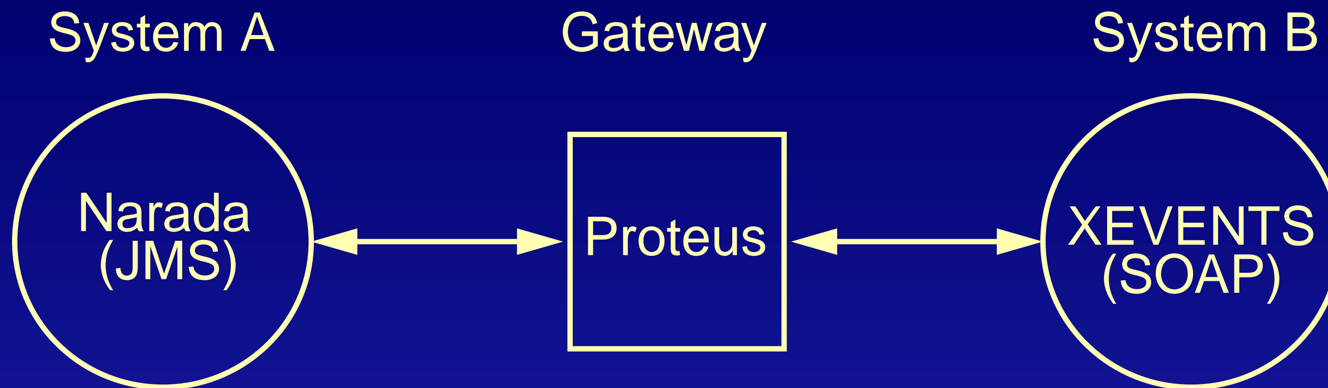
The Proteus Multiprotocol Message Library

Kenneth Chiu
Dennis Gannon

Extreme Lab
Indiana University

Motivation

- Protocol selection/negotiation
  - SOAP is popular, but slow.
  - Switch to binary protocols when supported by both ends.
- Grid intermediaries



Proteus

Proteus functions as a mediator between providers and clients.

- Provider

  Implement the protocols.
- Client

  Any program that uses Proteus.
- Node

  Addressable entity. Messages are sent and received through nodes.

Client A

Client B

N1

N2

N3

$Core_A$

$Core_B$

$P1_A$

$P2_A$

$P2_B$

$P1_B$

Network

Message Model

- A Proteus message consists of a set of named parts.
- Each part contains either:
  - Vobject

    Structured, accessible data.
  - Matter

    Data in an opaque form.
- Parts partition information according to roles.

  For example, a gateway that only needs routing information may only examine the header part.

Vobjects

- Primary building block is a *vobject*, which is an ordered set of typed, named fields. Each field may contain:
  - A primitive value.
  - A nested vobject.

- Primary Proteus mechanism for manipulating message contents is data binding.

```
struct RecordVobject {
    int i;
};
struct MyVobject {
    RecordVobject record;
} vobj;
...
int i = vobj.record.i;
```

- Programmatic interface also useful.

Matter

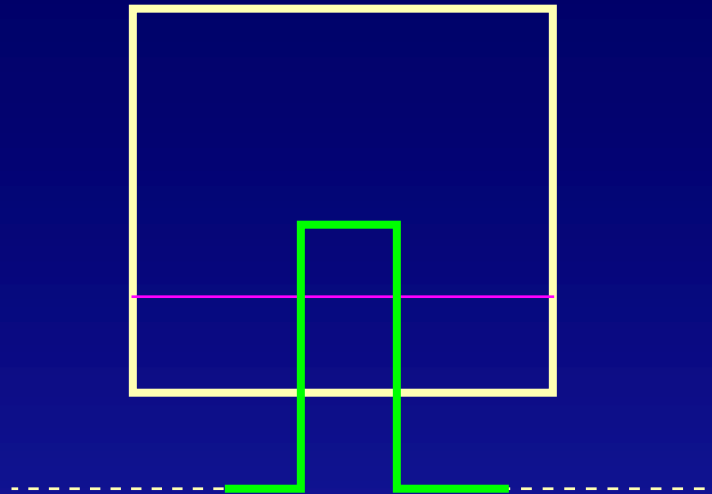
- Deserializing into a vobject not always desirable.
  - Requires compile-time knowledge of format and type.

    Impractical for intermediaries.
  - Inefficient.
- Matter is a vobject in raw, protocol-specific form.
- Two pieces of information are attached to matter.
  - Provider descriptor
  - Encoding attributes

    For efficient encapsulation during transport.

Wormhole Matter

- A client may indicate to a provider that a part will only be retransmitted, and retransmitted at most once.
- A provider may then optimize retransmission.



Generic Serialization

- Typical serialization code entangles type dependencies and protocol dependencies.

```
serializeMyVobject() {
    SOAP_pack_int(i);
    SOAP_pack_float(f);
}
```

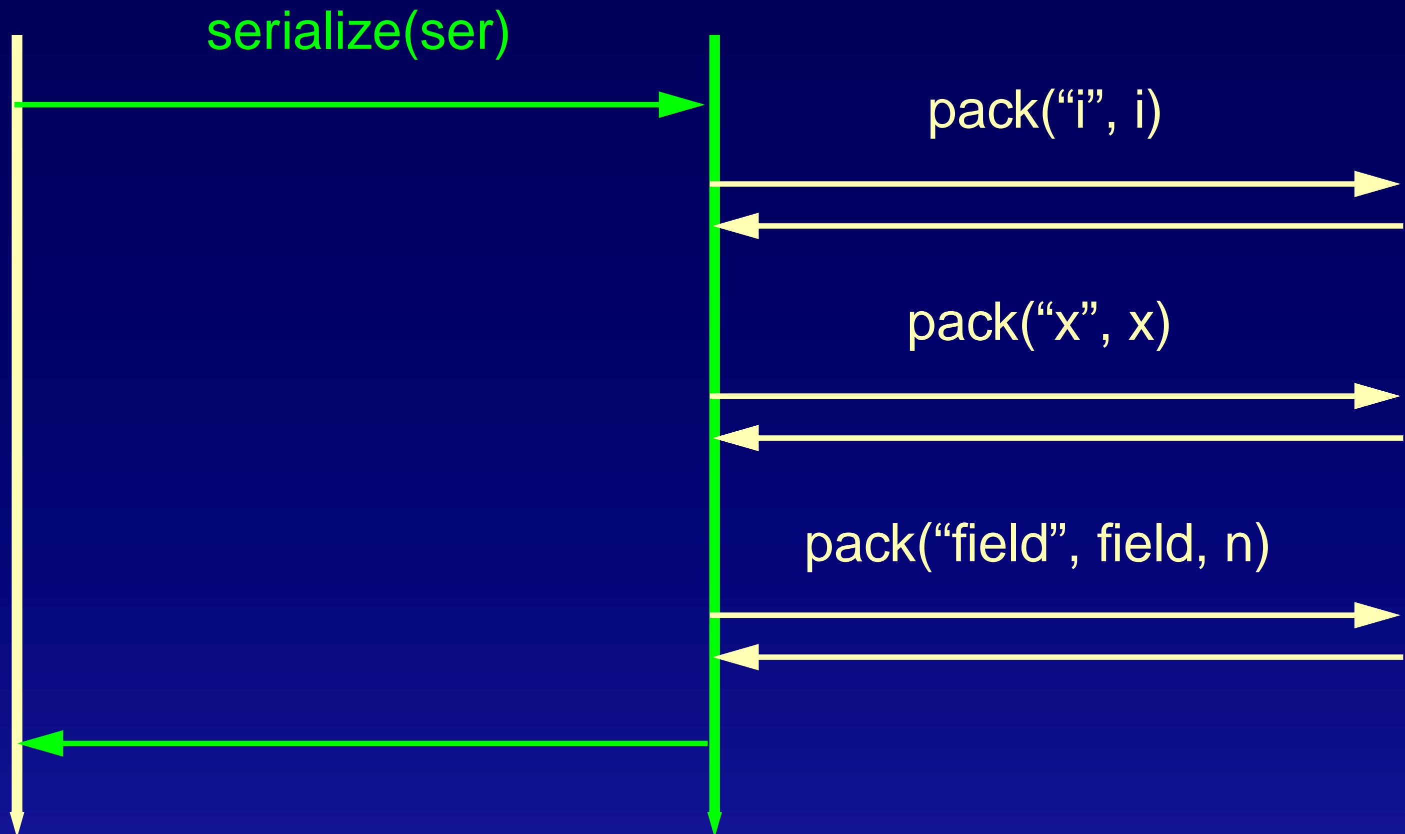
- This is fine for uniprotocol systems, but results in a multiplicative code increase for multiprotocol systems.

- Use double-dispatch to separate the type-dependencies from the protocol-dependencies.

```
struct NestedVobject;
struct MyVobject {
    NestedVobject nested;
    int i;
};

MyVobject::serialize(const char *name,
                     Serializer *ser) const {
    ser->beginVobject(name, "MyVobject");
    nested->serialize("nested", ser);
    ser->pack<int>("an_int", 12345);
    ser->endVobject();
}
```

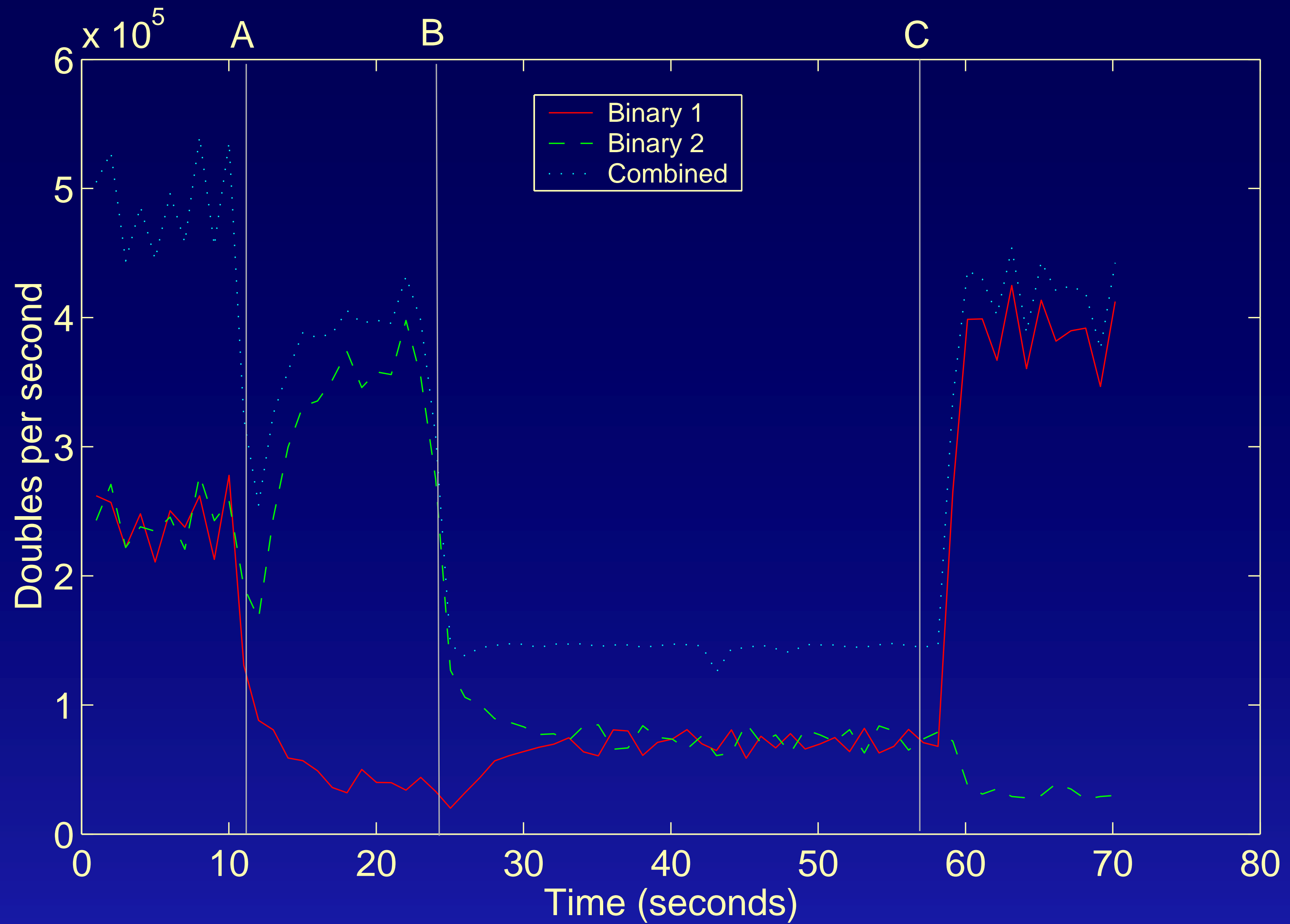
SOAPProvider MyObj SOAPSerializer



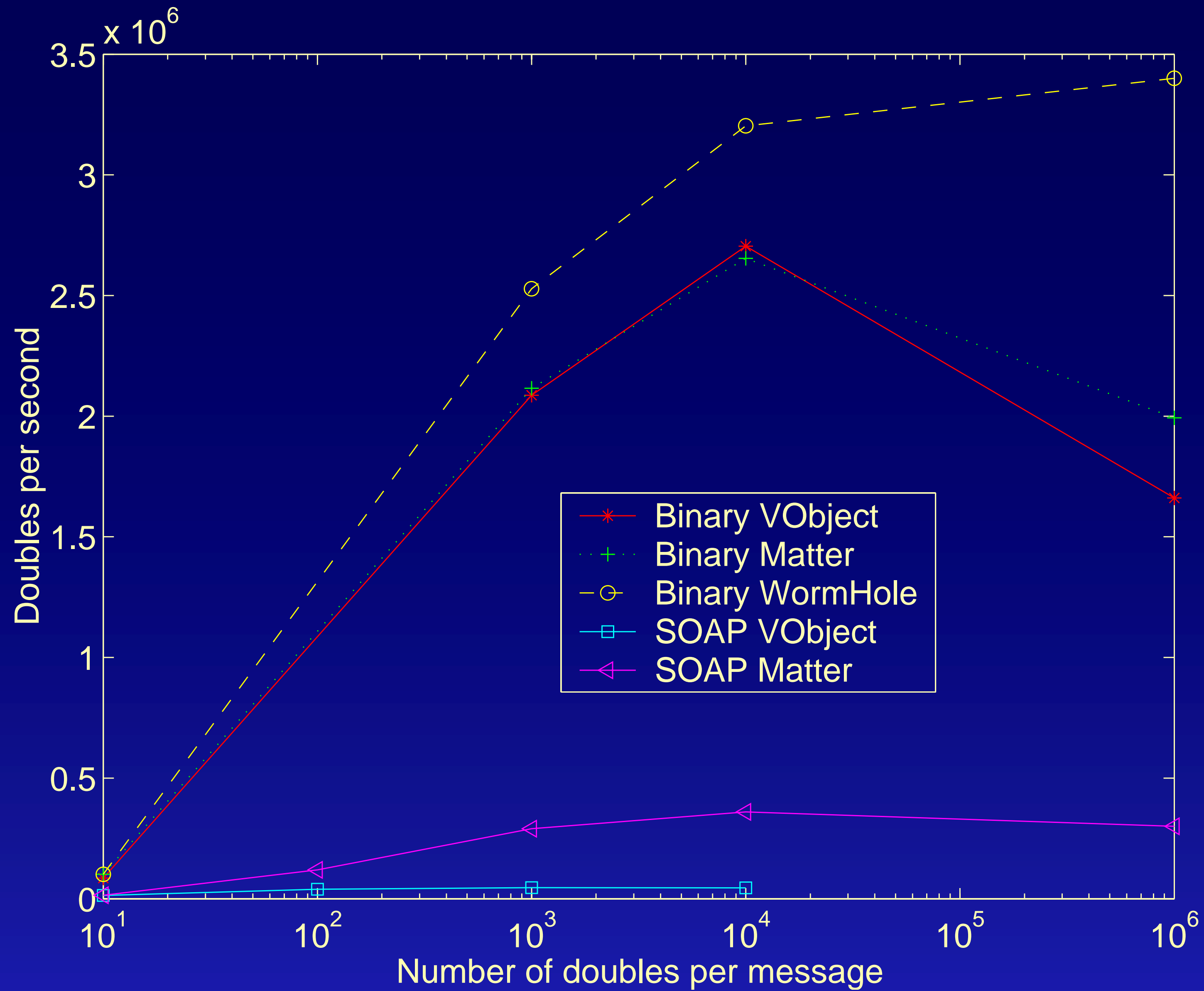
Nodes

- Serve as a binding locus for various pieces of information.
- Each node has a multiaddress, which is just an aggregation of uniaddresses.
- A selector object is bound to each node. The selector is responsible for choosing a uniaddress when a message is sent.

Performance



Matter



Future Work

- Security
- WSDL Integration
- Parallel Communication
- Route Feedback

Non-Blocking Synchronization

Problems with Mutual Exclusion

- Fault-intolerance

  A process which fails while holding a lock leaves the system in an inconsistent state.

- Priority inversion

  A low-priority process can indefinitely block a high-priority process (Mars Pathfinder).

- Deadlock
- Low throughput

  High-contention, too many writes.

Terms

- Wait-free

  All processes deterministically guaranteed to complete.
- Non-blocking (lock-free)

  One process guaranteed to complete.
- Type-stable memory (TSM)

  A formalism of the idea that an object remains valid even after it is "freed."
- Consensus number

  An object has a consensus number N if it can be used to solve the consensus problem for at most N processes.

Synchronization Instructions

- TEST&SET (2)

  Atomically read a shared variable and set it to one.

- FETCH&ADD (2)

  Atomically read a shared variable and increment it.

- LOAD-LINKED/STORE-CONDITIONAL (∞)

  The first instruction reads a variable. The second instruction writes it only if no other writes intervene.

- COMPARE&SWAP (∞)

  Given a shared variable V, an old value O and a register N containing the new value, the instruction

  COMPARE&SWAP V,O,N

  will swap V with N only if $V = O$.

  Two variations of COMPARE&SWAP

  - Double COMPARE&SWAP

    Two discontiguous words. Not commonly available.

  - Double-word COMPARE&SWAP

    One aligned double-word. Commonly available.

Previous Work

- Herlihy (1991)

  General transformation techniques. Consensus hierarchy.
- Valois (1995)

  Linked lists using COMPARE&SWAP. Traversal requires writes.
- Michael and Scott (1996)

  FIFO queues using double-word COMPARE&SWAP.
- Greenwald and Cheriton (1996)

  Linked list using double COMPARE&SWAP.

Problem Definition

- Singly-linked list
- Arbitrary insertion
- Arbitrary removal
- Linearizable
- Assume TSM
- Assume double-word COMPARE&SWAP

Linearizability

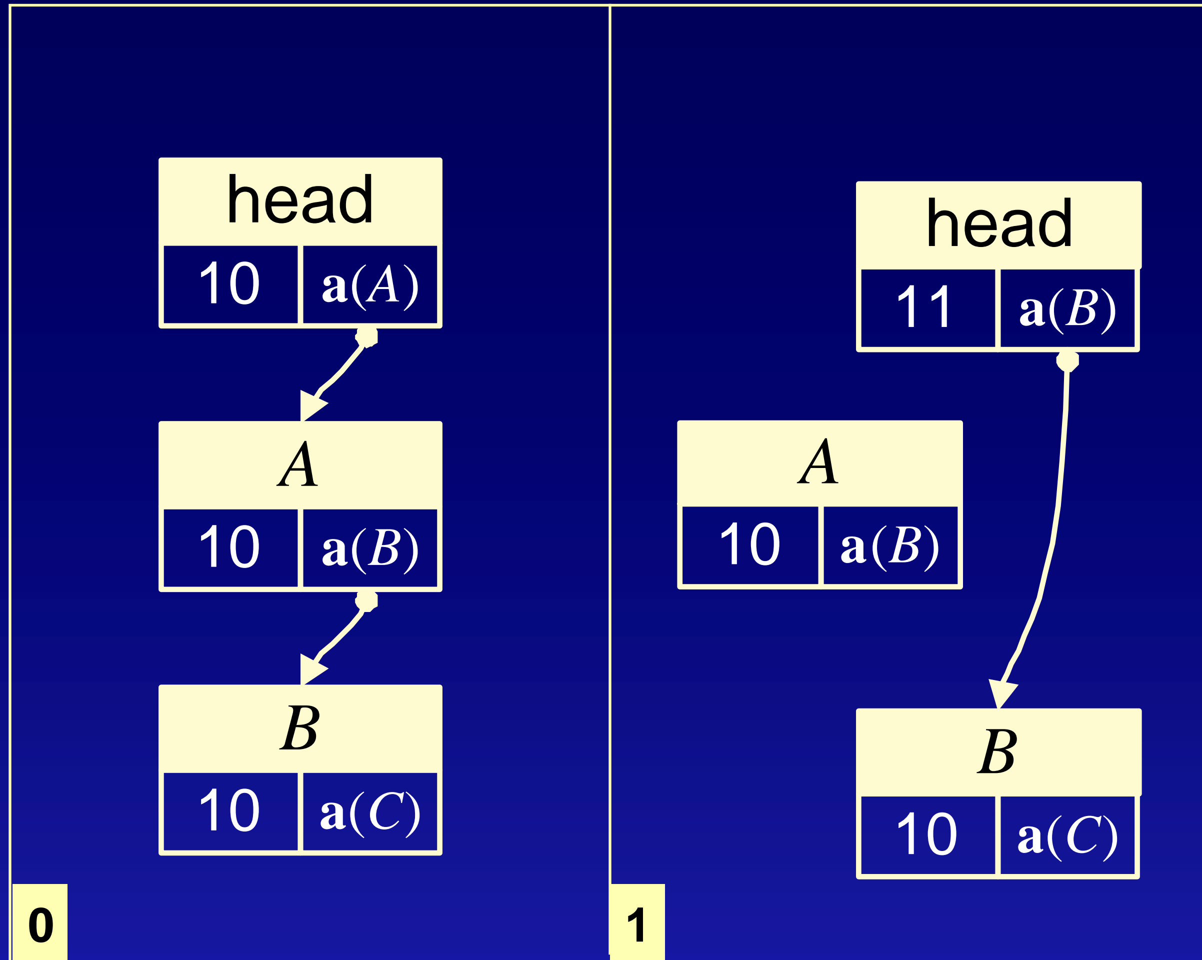
- An object is linearizable if it preserves ordering of non-concurrent operations, and executes concurrent operations as if they were in some sequential order.

Not linearizable	Linearizable
A: Push(1)	A: Push(1)
B: Push(2)	B: Push(2)
B: Ok	B: Ok
A: Ok	A: Ok
C: Push(3)/Ok	C: Push(3)/Ok
D: Pop/Ok(1)	D: Pop/Ok(3)
D: Pop/Ok(3)	D: Pop/Ok(1)
D: Pop/Ok(2)	D: Pop/Ok(2)

- Contrast with sequential consistency.

Example of NBS

Popping an element from a stack.

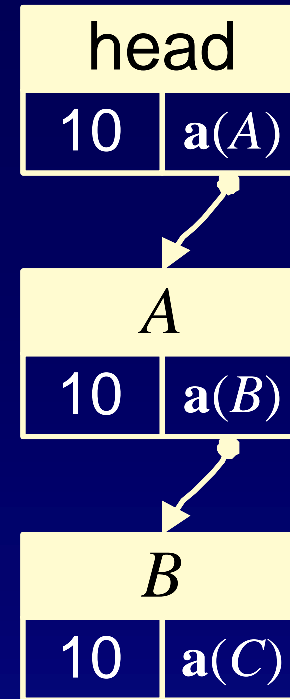


a(E) = address of element E

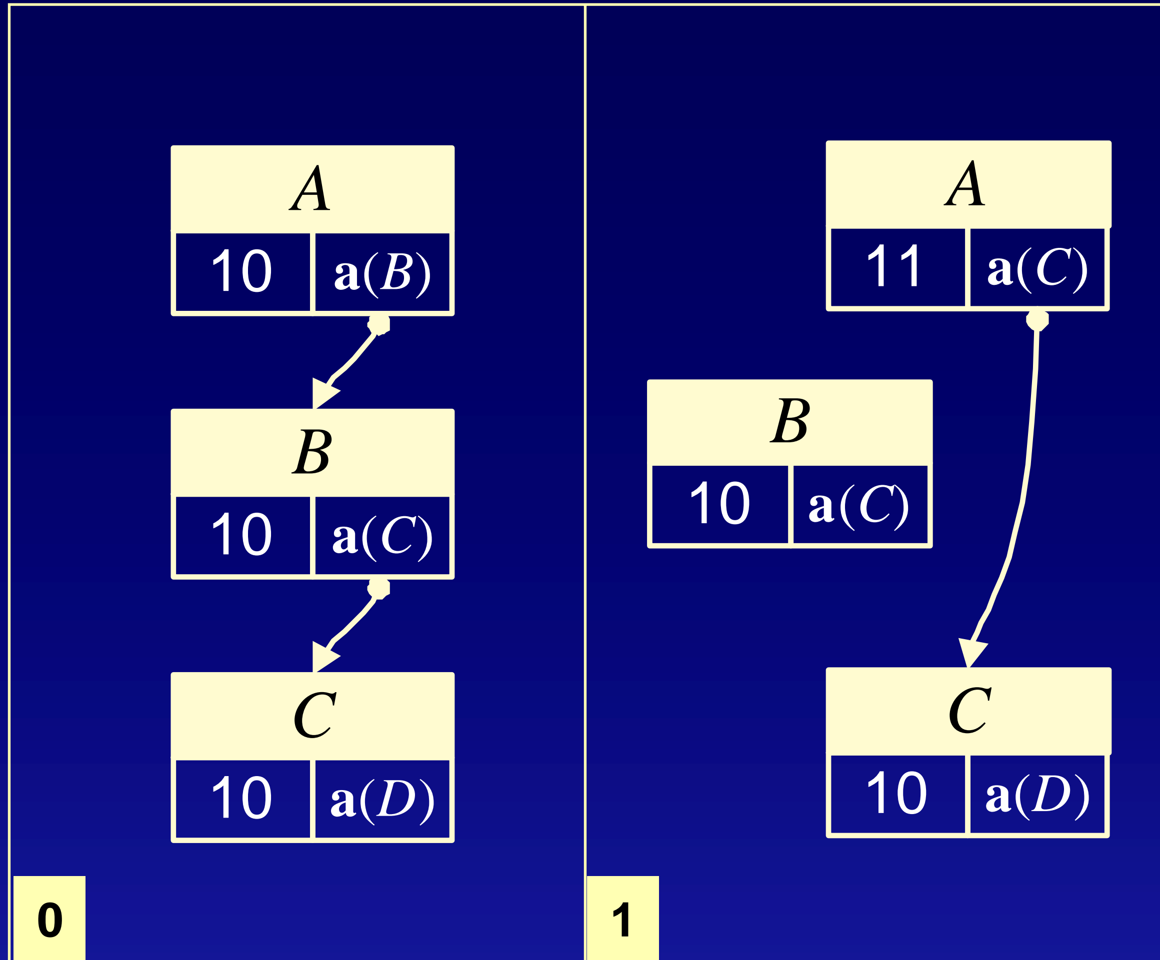
```
struct Link {
   int gen;
   Element *ptr;
} head;

struct Element {
   ...
   Link next;
};

do {
   Link new_head = org_head = head;
   new_head.ptr = new_head.ptr->next.ptr;
   new_head.gen++;
} while (!cas(&head, org_head, new_head));
```

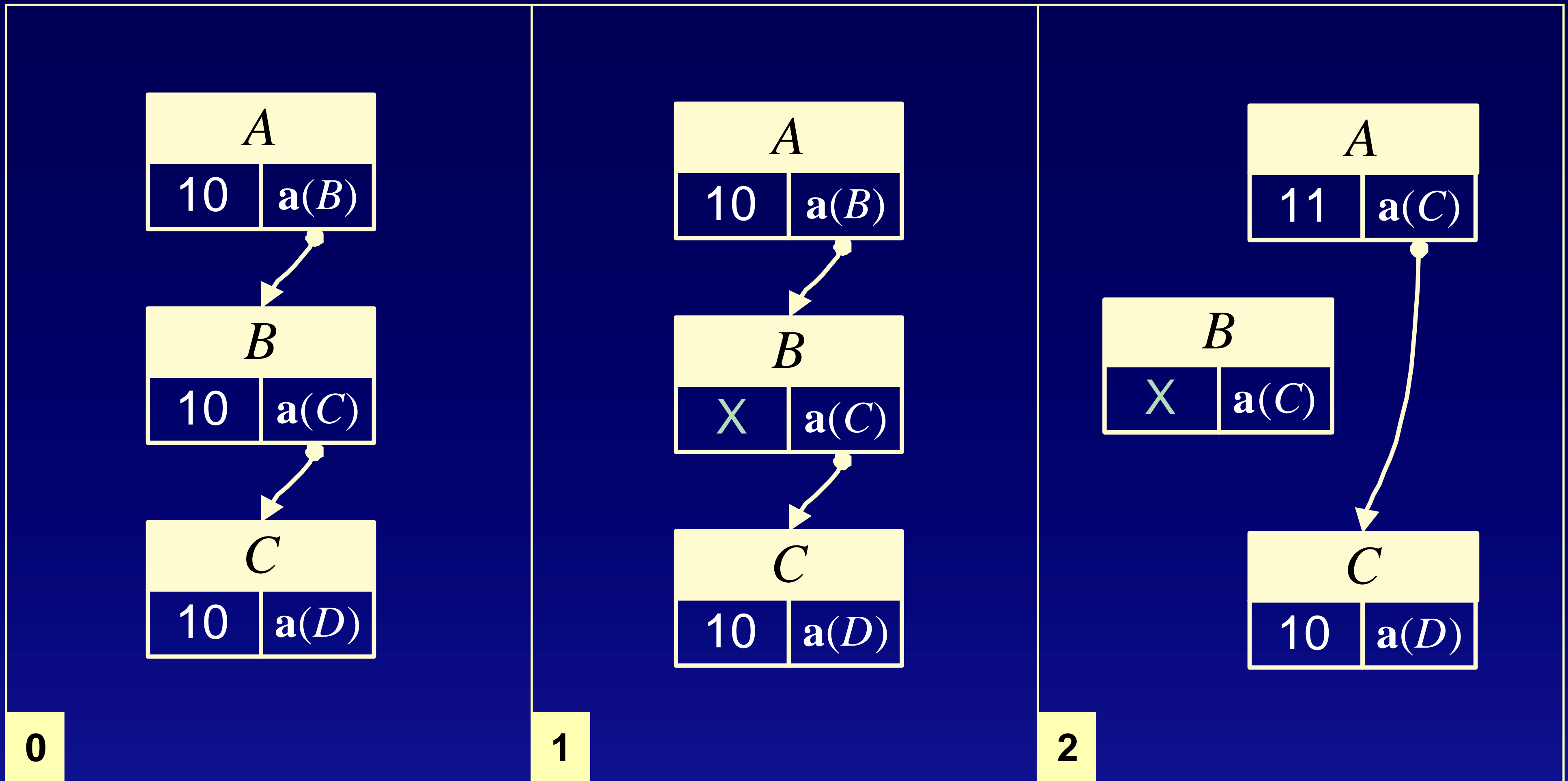


Does This Work for Lists?



Key difference is that elements are removed only from the head of a stack.

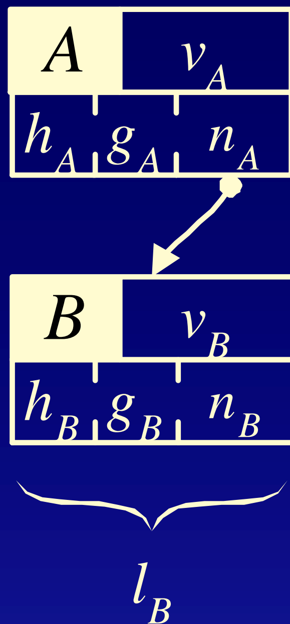
Marking



To preserve non-blocking property, other processes are allowed to remove a marked element.

Two Tags

- Two tags are stored along with each pointer. This triple constitutes a link.



$$g_E = \text{generation of element } E$$

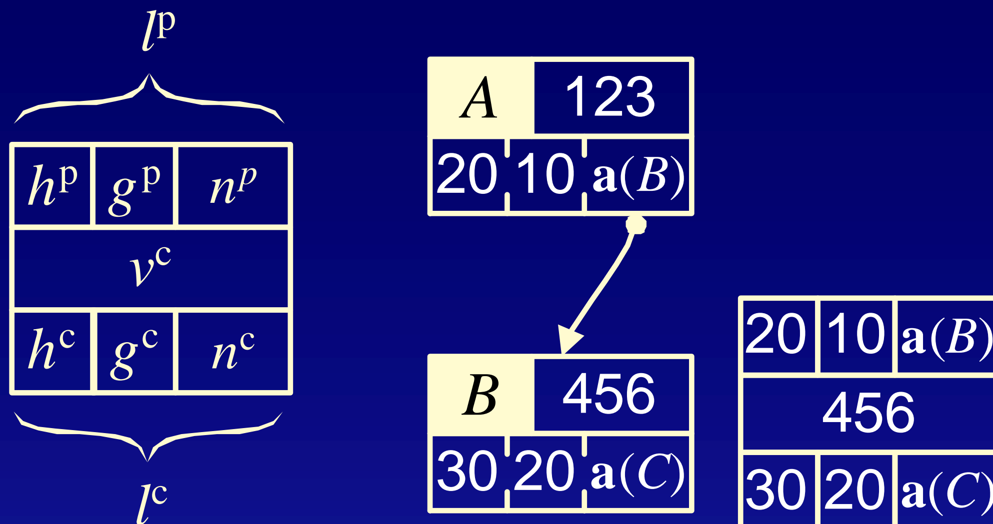
$$h_E = \text{generation of successor to } E$$

$$n_E = \text{pointer to successor}$$

$$l_E = (g_E, h_E, n_E)$$

Cursor

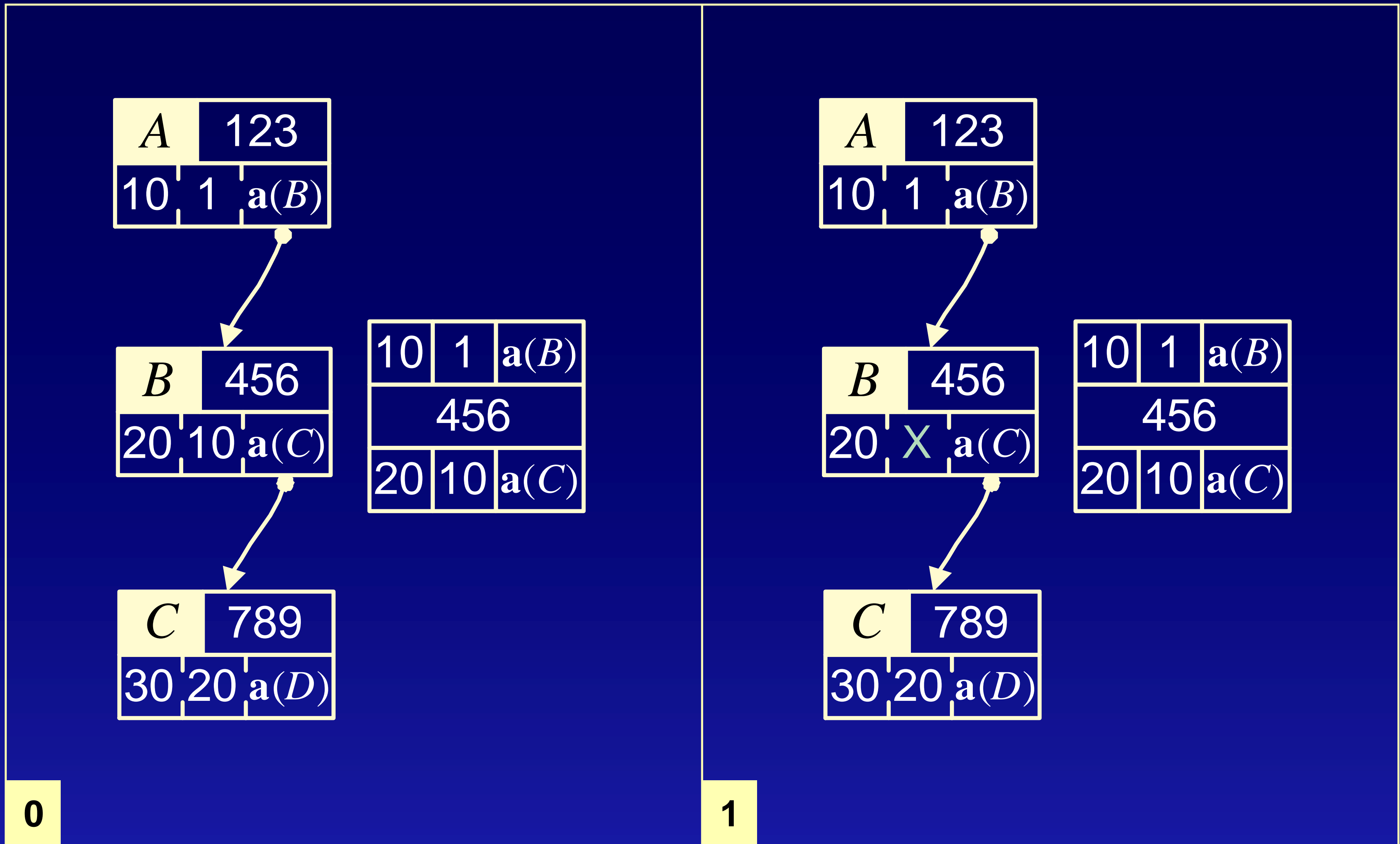
- Represents a snapshot in “time.”
- Contains two links and the value of the visited element.
- Aids detection of conflicts.

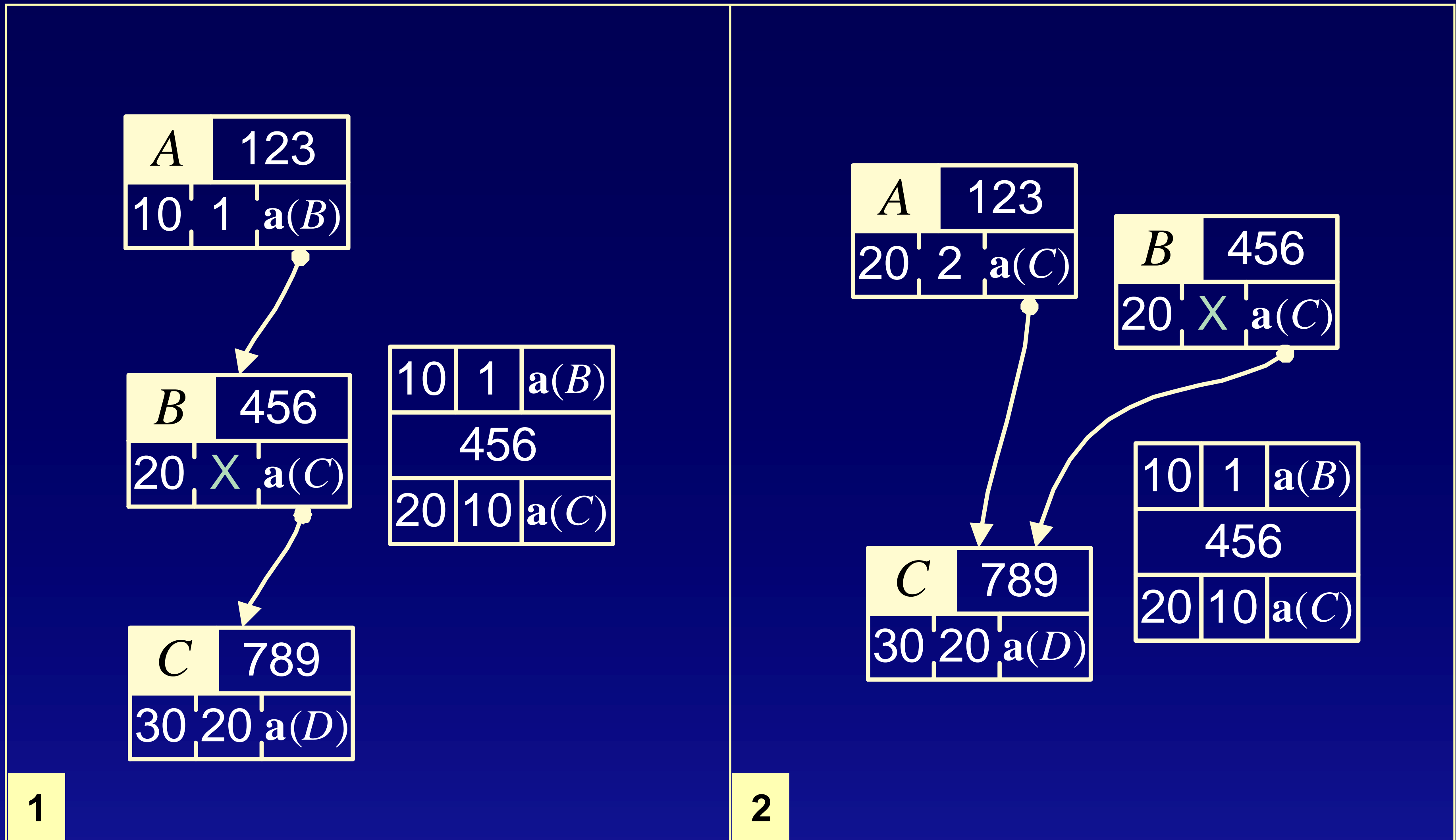


If the cursor is visiting B, then normally

$$l^p = l_A$$
$$l^c = l_B$$
$$v^c = v_B$$

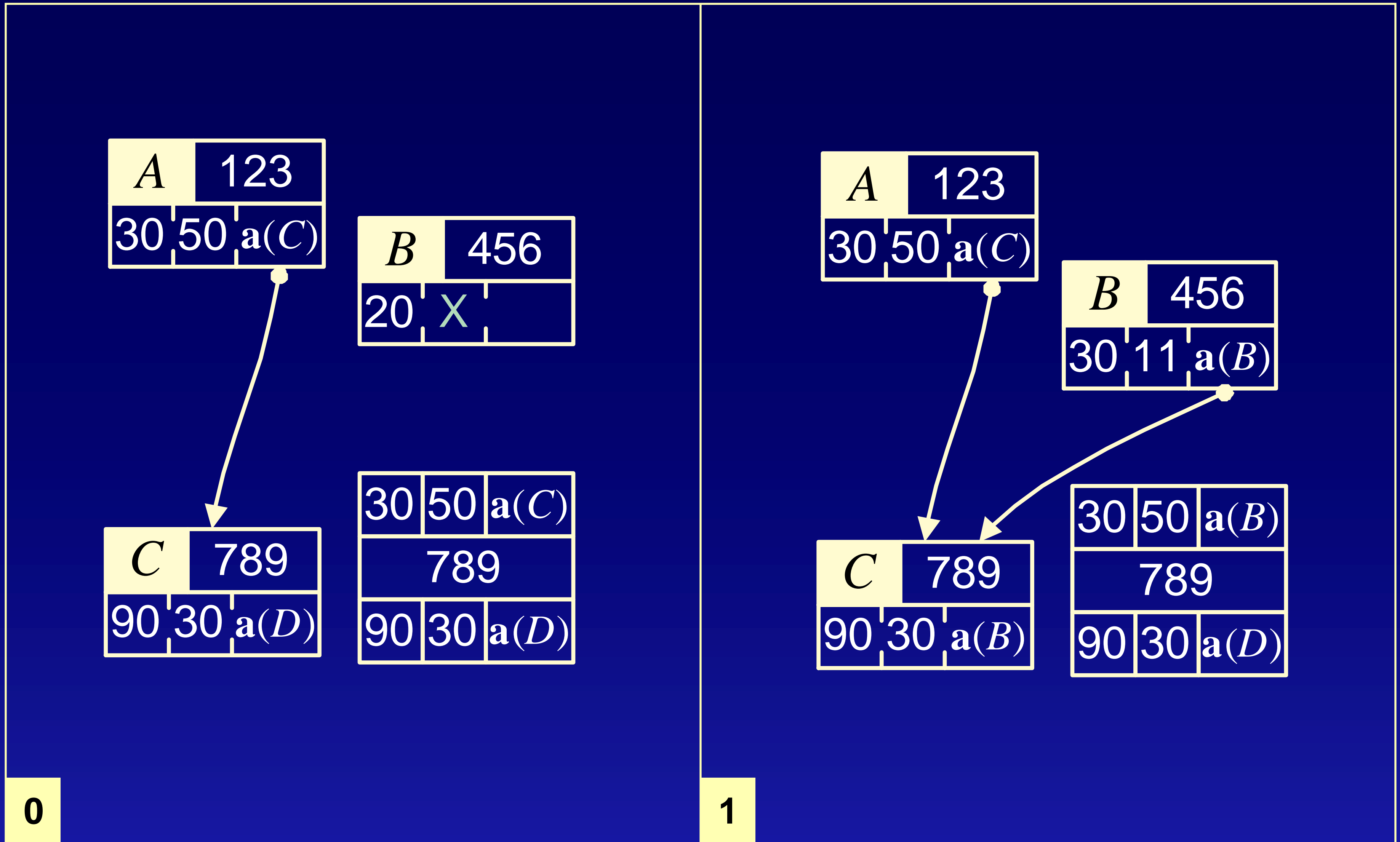
Removal

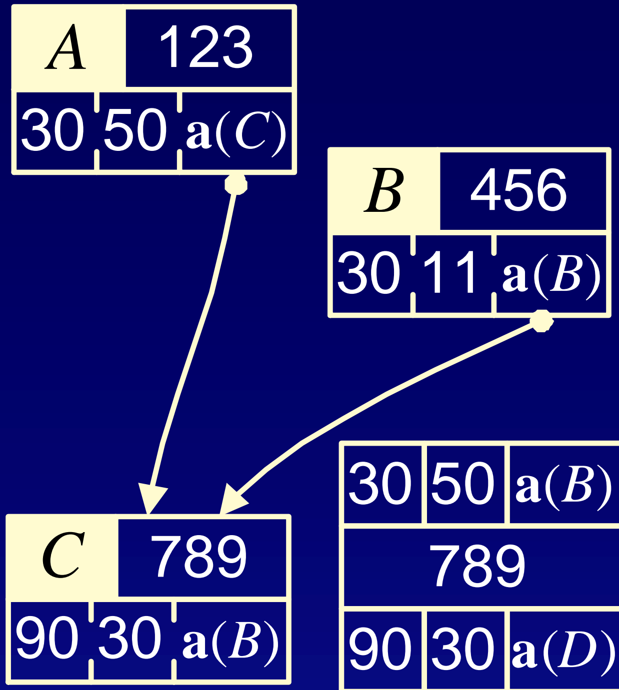
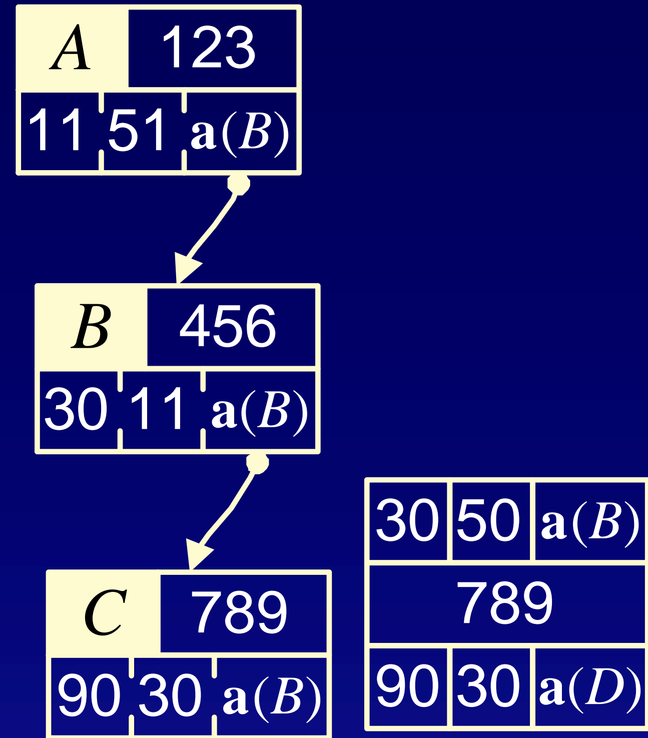




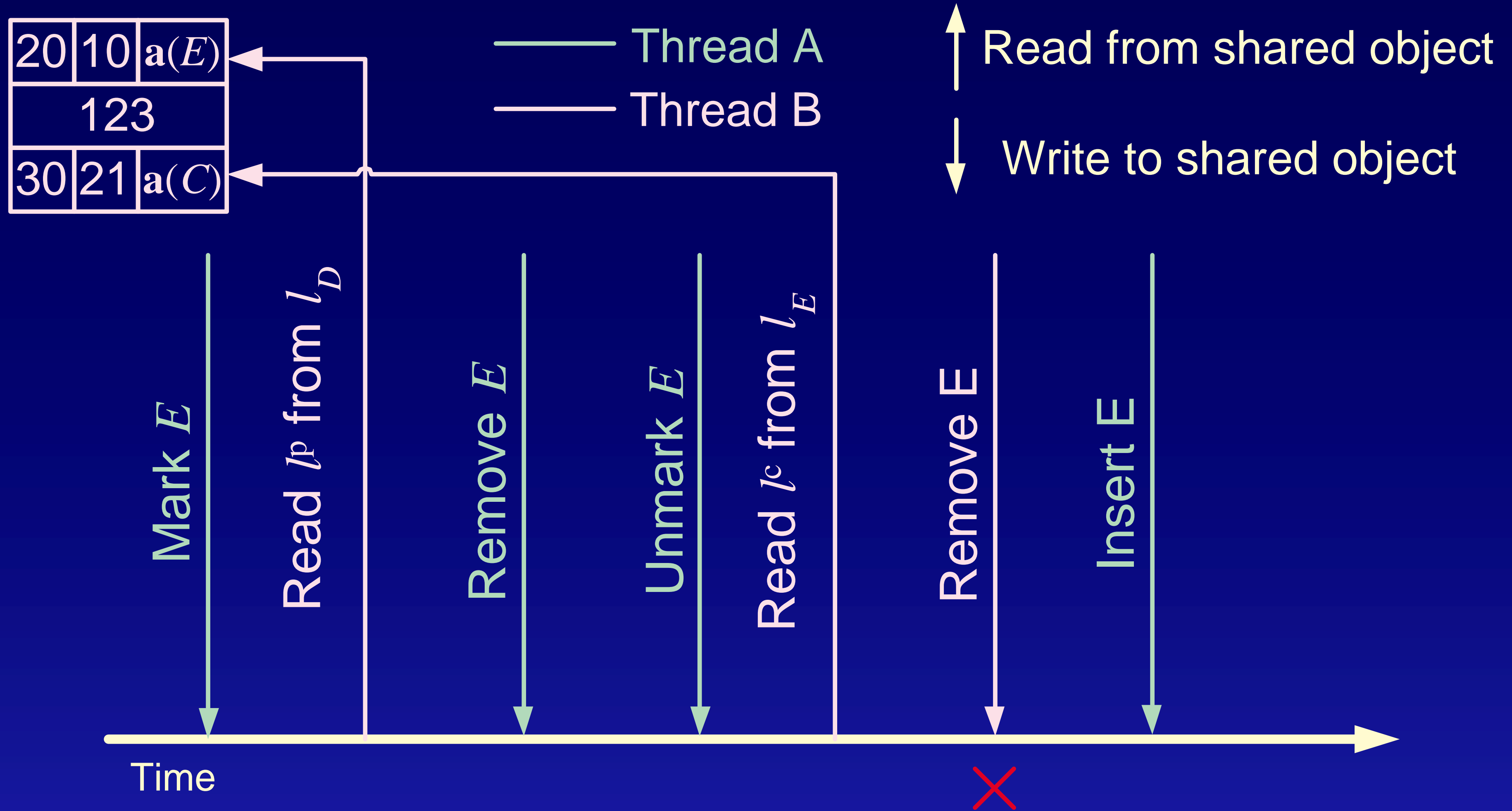
If a conflict is detected, must restart from head of list. Minor conflicts can be fixed-up with restarting.

Insertion




A
123
30
50
a(C)
B
456
30
11
a(B)
C
789
90
30
a(B)
30
50
a(B)
789
90
30
a(D)
1

A
123
11
51
a(B)
B
456
30
11
a(B)
C
789
90
30
a(B)
30
50
a(B)
789
90
30
a(D)
2

Purpose of h



Algorithm Outline

- Ordered insert

```
Element *e = ...;

RESTART:

    Cursor c = begin();

    while (!proper_position()) {
        if (!c.advance()) {
            goto RESTART;
        }
    }

    if (!c.insert(e)) {
        goto RESTART;
    }
```

Performance

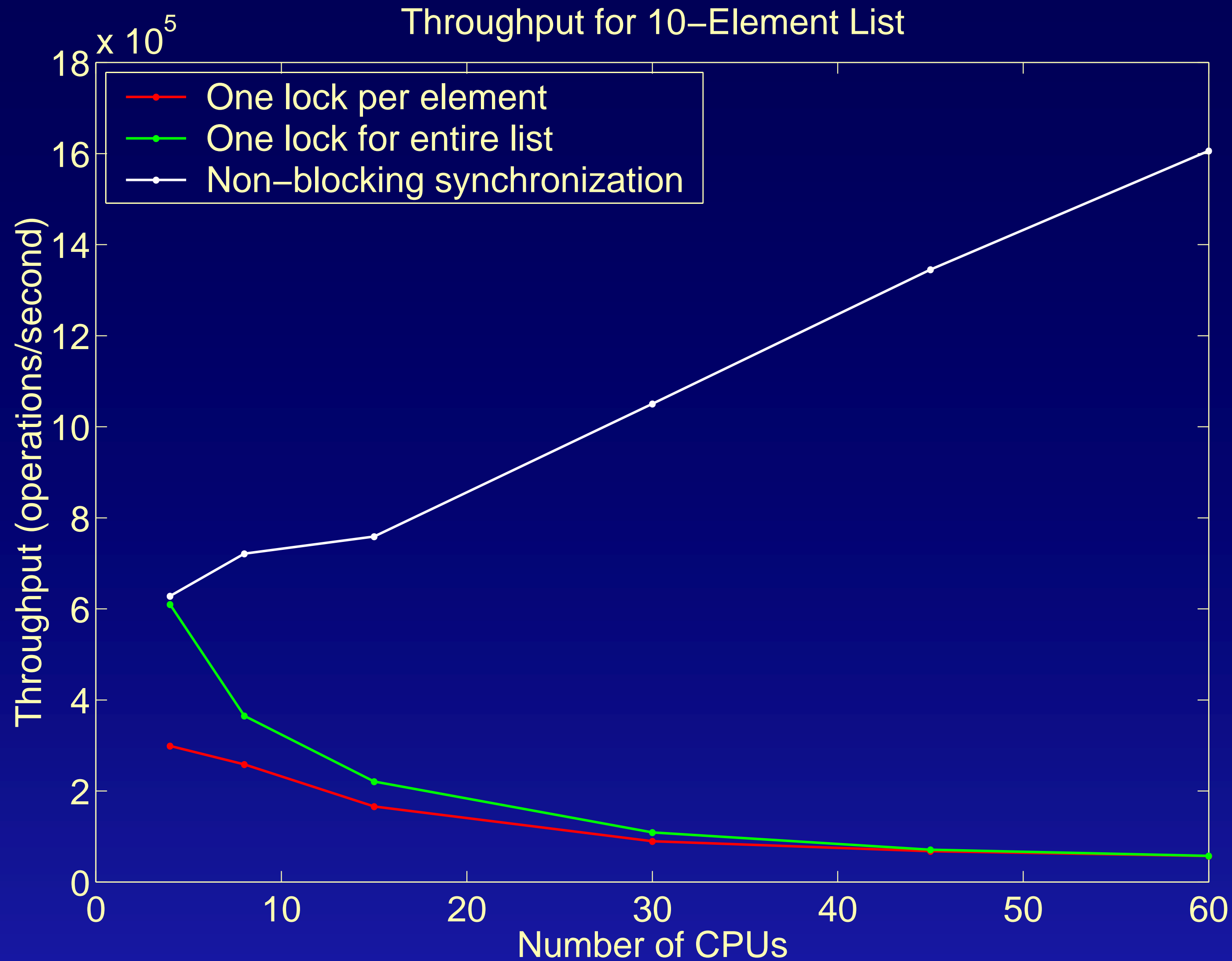
- Tests conducted on an E10000.
- Operations are a mixture of search&removal, and insertion.

```
while (true) {
    r = random();
    Element *e = list.remove(r);
    if (e != NULL) {
        e->value = random();
        list.insertSorted(e);
    }
}
```

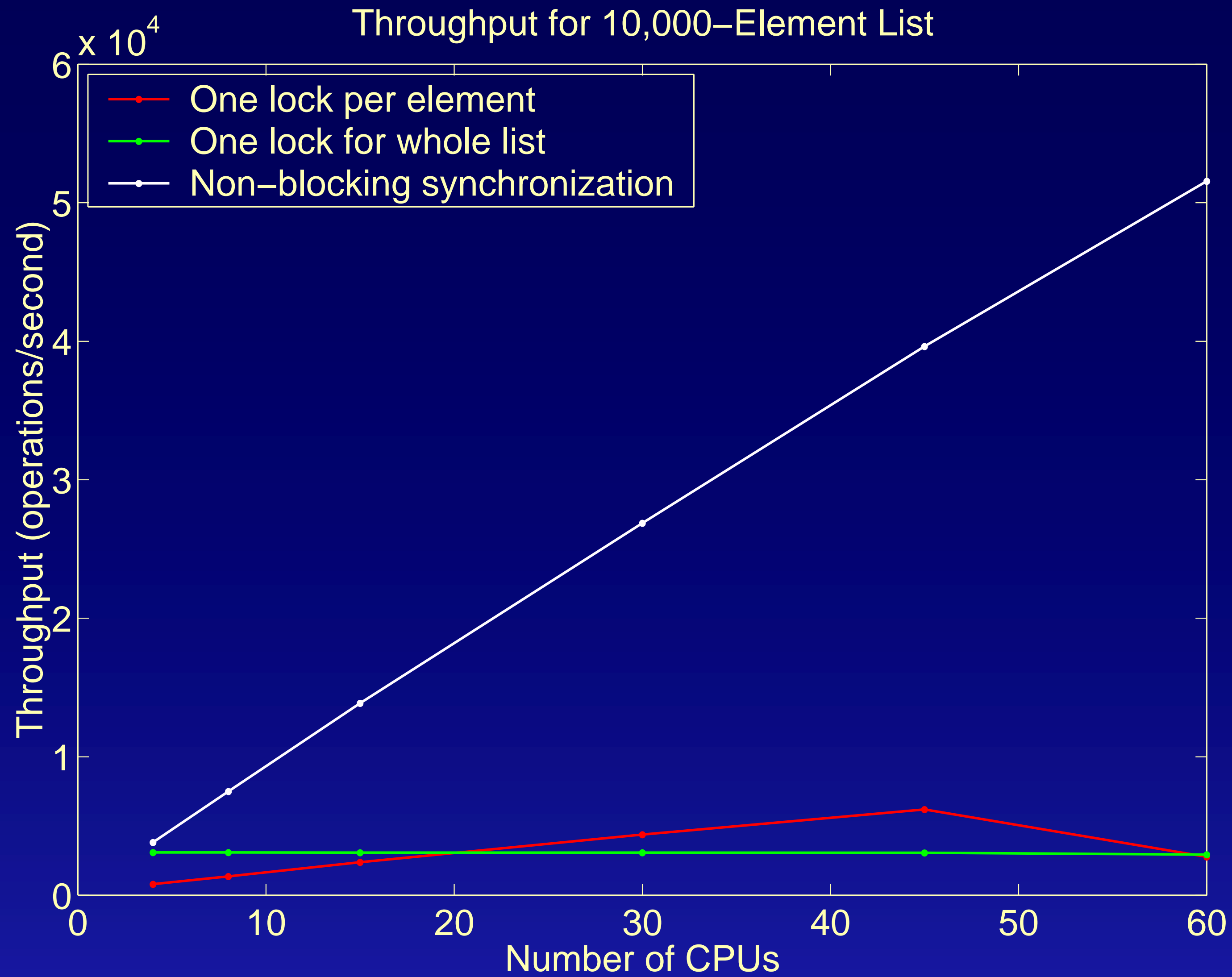
- Tested against one lock for each element and one lock for the entire list. Locks in the locking algorithms are spin locks spinning on read.

- Comparison against locking methods intended merely as a baseline sanity check.

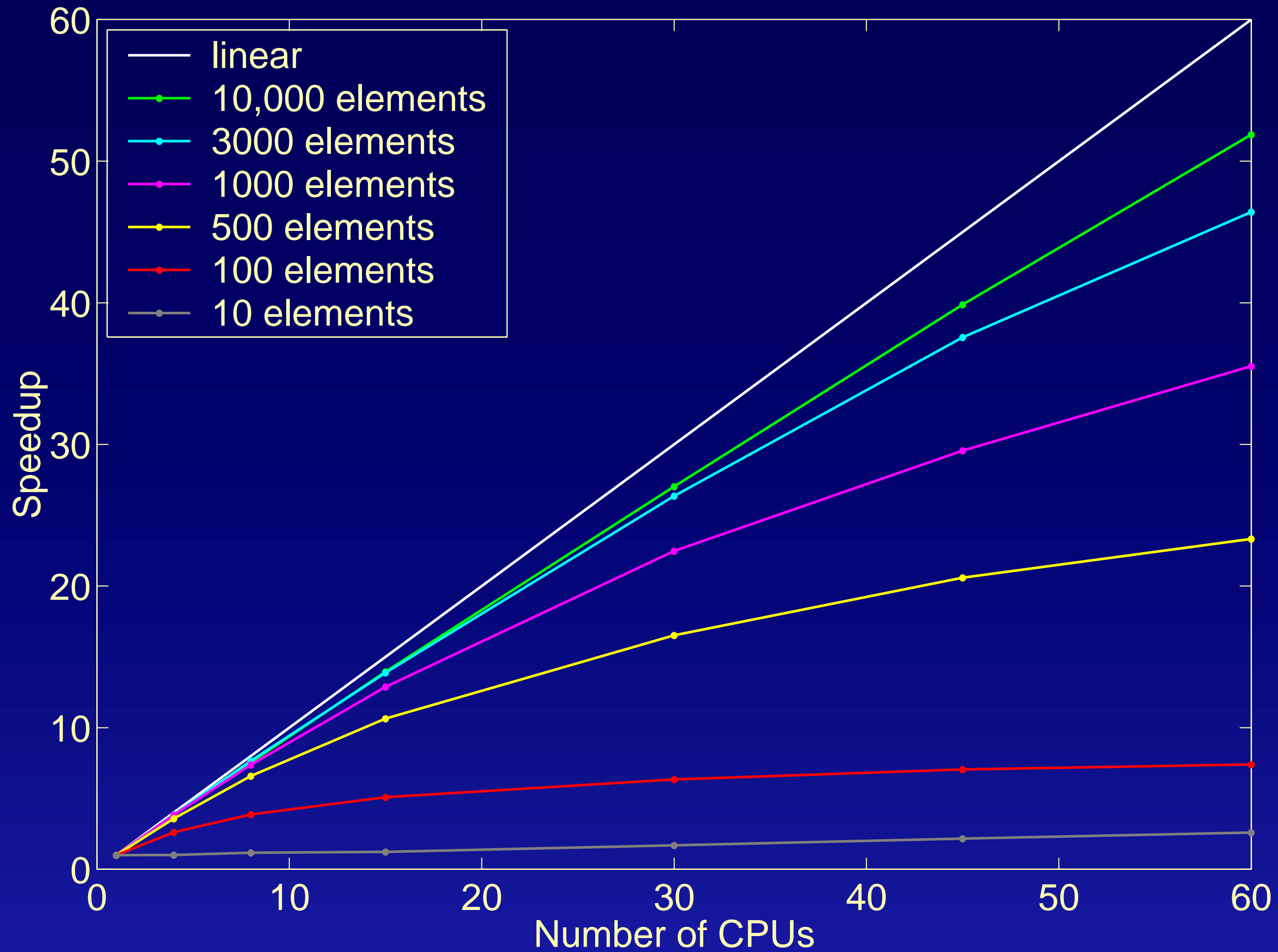
10 Elements



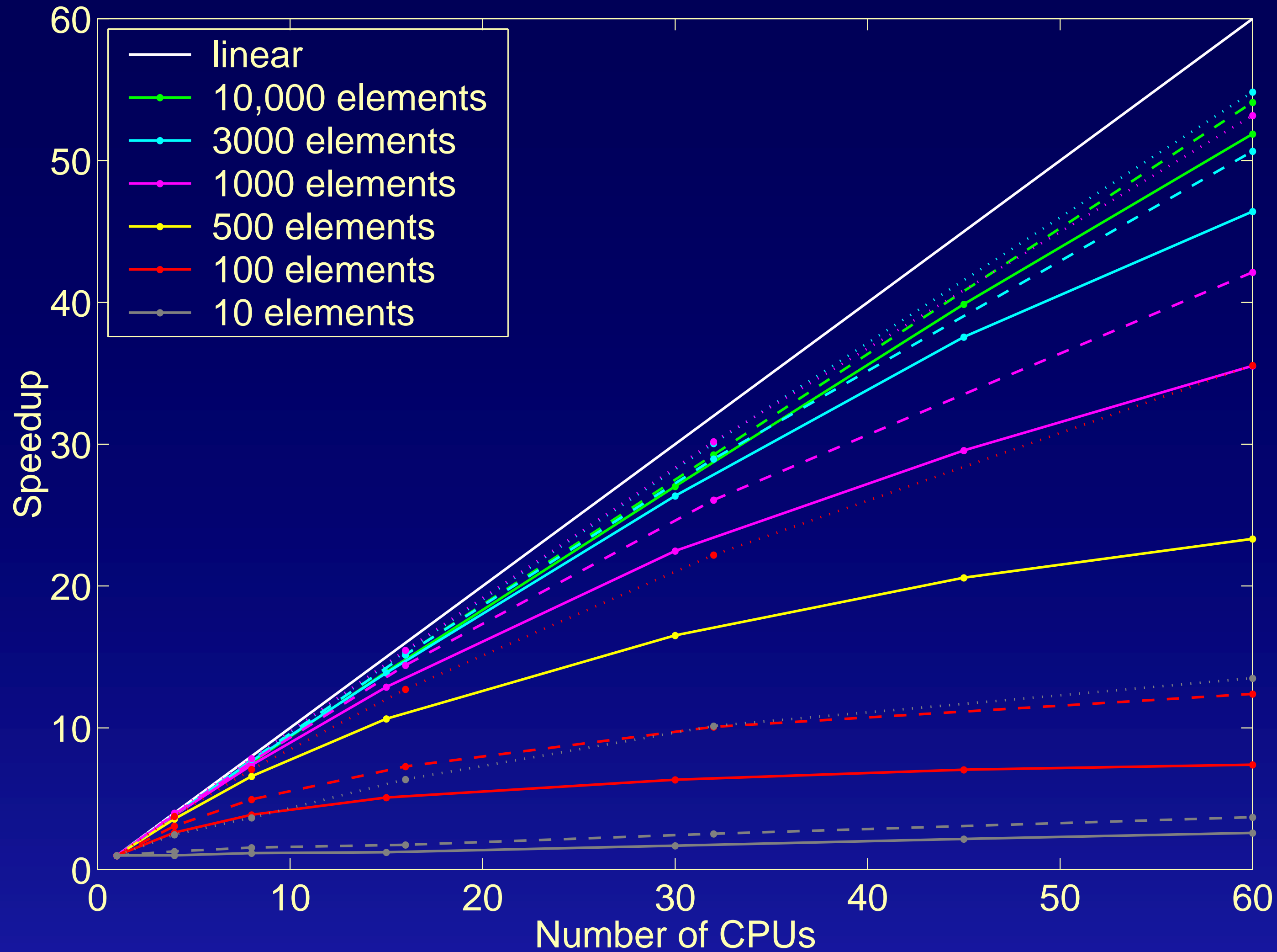
10,000 Elements



Speedup



Expensive Compares



To Do

- Formal verification

  A number of incorrect algorithms have been published. Would like to use machine-assisted.

- Performance enhancements

  Reduce memory operations. Reduce restarts. Perhaps wait a bit if encounter a marked element. Hybrid methods.

- Relaxed memory models

  Will require memory barriers, acquire/release. Verification more difficult than sequential.

- Investigate performance on NUMA architectures

Scratch Slides

Goals

- Communicate with non-Proteus clients and servers.
  - Non-Proteus client to a Proteus server.
  - Proteus client to a non-Proteus server.
- Dynamic loading of protocols.

```
AdaptSelector::send(const Scheme &scheme,
                    const Nodepnt &ret_addr
                    const Message &msg) {
    int u;
    double r = drand48();
    // Choose each uniaddress at least
    // 1% of the time.
    if (r < .01) {
        u = 0;
    } else if (r < .02) {
        u = 1;
    // Otherwise, choose based on how
    // fast it is.
    } else {
        // Renormalize.
        r = (r - .02)/.98;
        double sqr0 = t_put[0]*t_put[0];
        double sqr1 = t_put[1]*t_put[1];
        double cut = sqr0/(sqr0 + sqr1);
        if (r < cut) {
            u = 0;
        } else {
            u = 1;
```

```
            }
        }
        Time start(Time::Now());
        scheme[u].send(msg, ret_addr);
        Time elapsed = Time::Now() - start;
        t_put[u] = .9995*t_put[u]
             + .0005/elapsed.toDouble();
    }
```

Model Checking

Traditional model checking does not seem well-suited.

- Can only check finite instances.
- Difficult to express safety properties as temporal logic expression over state. TLA (Lamport) might be more expressive.

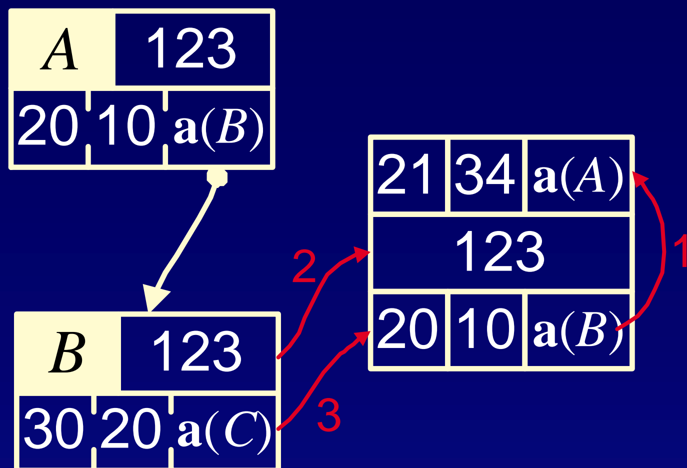
```
Port *sim1 = ...; PortAddress vis1 = ...;
sim1->connect(vis1);
```

Consensus Hierarchy

- Some primitives are more powerful than others.
- An object with consenus number N can be used to implement any wait-free object for N or few processes.
- The consensus number defines a robust hierarchy.

Advancing a Cursor

The cursor below is being advanced from E_1 to E_2.



$$l_p \leftarrow l_c$$
$$v_c \leftarrow v_2$$
$$l_c \leftarrow l_2$$

Clearly not atomic, but ignore that for now.

Validating a Cursor

Advancing a cursor is not an atomic operation. It may no longer be valid. Validation is the process of fixing it up. Two conditions require fixing.

20	10	$\mathbf{a}(B)$
123		
30	X	$\mathbf{a}(C)$

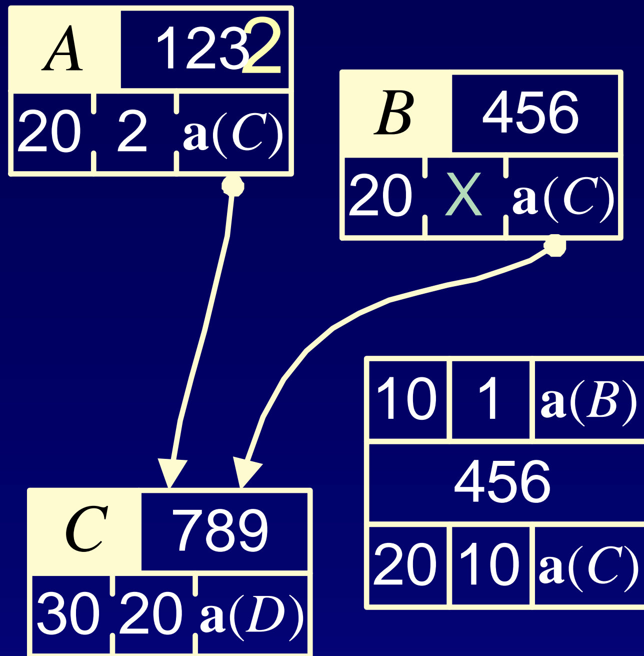
If the current element has been marked, then we need to remove it before continuing. If, upon trying to do so, we detect that gi-1 != gp, then this indicates the current element may have been removed and reinserted. This may cause portions of the list to be skipped, so a restart is necessary. If only h differs, fix and continue.

Another condition is that h may be out-of-date. That is, hp != gc. We try to correct this situation. If it fails because h

has changed, we can reload with the new value of h and try again.

If it fails because g has changed, many things could have happened, and we restart.

<table>
<tr><td>20</td><td>10</td><td>$\mathbf{a}(B)$</td></tr>
<tr><td colspan="3">123</td></tr>
<tr><td>30</td><td>20</td><td>$\mathbf{a}(C)$</td></tr>
</table>

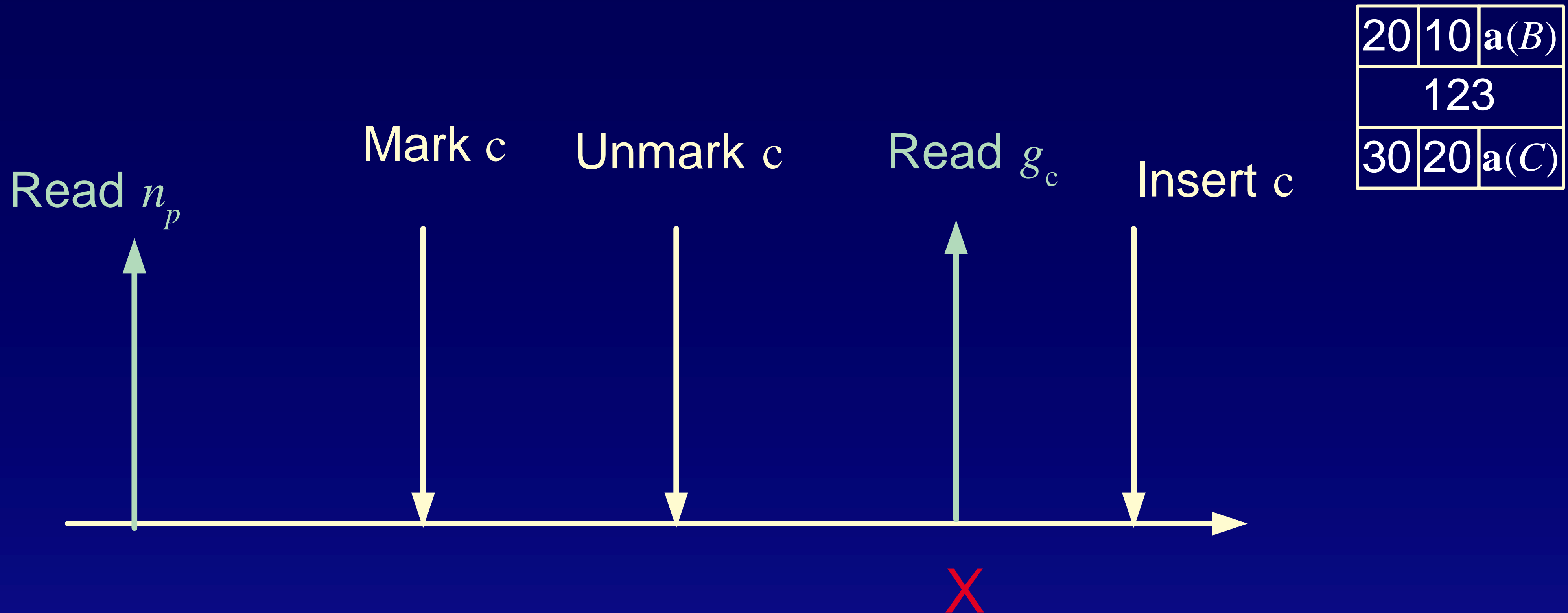


If the COMPARE&SWAP fails, it might be for a number of reasons. Must retraverse the whole list. If we don't find this element again, then we can be sure it has been removed properly.

The h Field

The h field of the link serves at least two important functions. The first is that it XXXX that when a cursor is

advanced, the value read from the element is consistent with the links.



The second important function is that it prevents an element about to be inserted from being prematurely remarked for removal.

20	10	$\mathbf{a}(B)$
123		
30	20	$\mathbf{a}(C)$