

# **LOCK/WAIT FREE SYNCHRONIZATION**

# Synchronization

- Mutex
  - Blocking
- Lock-free
  - At least one operation in a set of concurrent operations finishes in a finite number of its processor's own steps
- Wait-free
  - Every operation finishes in a finite number of its processor's own steps
- Lock-free and wait-free often require hardware supported atomic operations
  - Like compare-and-swap (CAS)

# CUDA Compare and Swap

```
int atomicCAS(int* address,  
              int compare,  
              int val);
```

- Atomically:
- **old = \*address**
  - Could be in global or shared memory
  - There is also a 64-bit version for global memory
- **new = (old == compare ? val : old)**
- **\*address = new**
- Return **old**

# Busy-Wait 2-Mutex?

```
shared int turn = 2;

if(turn != !id) { // I can go in
    turn = id;
    <<< critical section >>
    turn = 2;

    << non-critical section >>
}
```

# Busy-Wait 2-Mutex?

- Proposed by Hyman

```
shared boolean ready[2] = {0,0};  
shared int turn = 0;
```

```
→ while (true) { // Try to acquire lock  
    ready[id] = 1; // Register my interest  
    while (turn != id) { // My turn?  
        while (ready[!id] == 1) ; // Spin  
        turn = id;  
    }  
    <<< critical section >>  
    ready [id] = 0;  
    << non-critical section >>  
}
```



# Busy-Wait 2-Mutex with CAS

```
shared int turn = 2;  
  
while(CAS(&turn, 2, id));  
  <<< critical section >>  
  turn = 2;  
  
  << non-critical section >>
```

# Example: Atomic Updates with CAS

```
class ClassName {  
  
    Data *dptr;  
  
    void Update() {  
        Date *oldptr;  
        Data *stage = new Data("newvalue");  
        do {  
            oldptr = dptr;  
        } while (!CAS(&dptr, oldptr, stage));  
    }  
};
```

# Dynamic Load Balancing

- Static Task list
- While ( Next = WorkList.Front() != END )
  - Perform work
- Find a *busy* processor  $p_b$ 
  - Share its load
- Repeat
  - for a random processor  $p_j$
  - Nonblocking Lock LockList[ $p_j$ ]
- Until lock not acquired
- Share remaining load of processor  $p_b = p_j$ 
  - [Edit Queue]
- unlock LockList[ $p_b$ ]

# Non-blocking Lock

```
bool locked =  
    CAS (&LockList [victim], 0,  
        threadID);
```

- This is generally a busy-wait style

# Edit Queue

- Delete the second half of unprocessed  $\text{WorkList}[p_b]$ 
  - In an array implementation: update  $\mathbf{end}[p_b]$
- Add it to  $\text{WorkList}[p_i]$
- Read new  $\text{WorkList.Front}[p_b]$ 
  - Read  $\mathbf{front}[p_b]$   $\longrightarrow$  **Race with  $p_b$ 's update of its front: front++**
- Advance  $\text{WorkList}[p_i]$  to new  $\text{WorkList.Front}[p_b]$ 
  - Start at new  $\mathbf{current}[p_b]$

# Load Stealing

**Victim:**

```
ProcessMyShare:
```

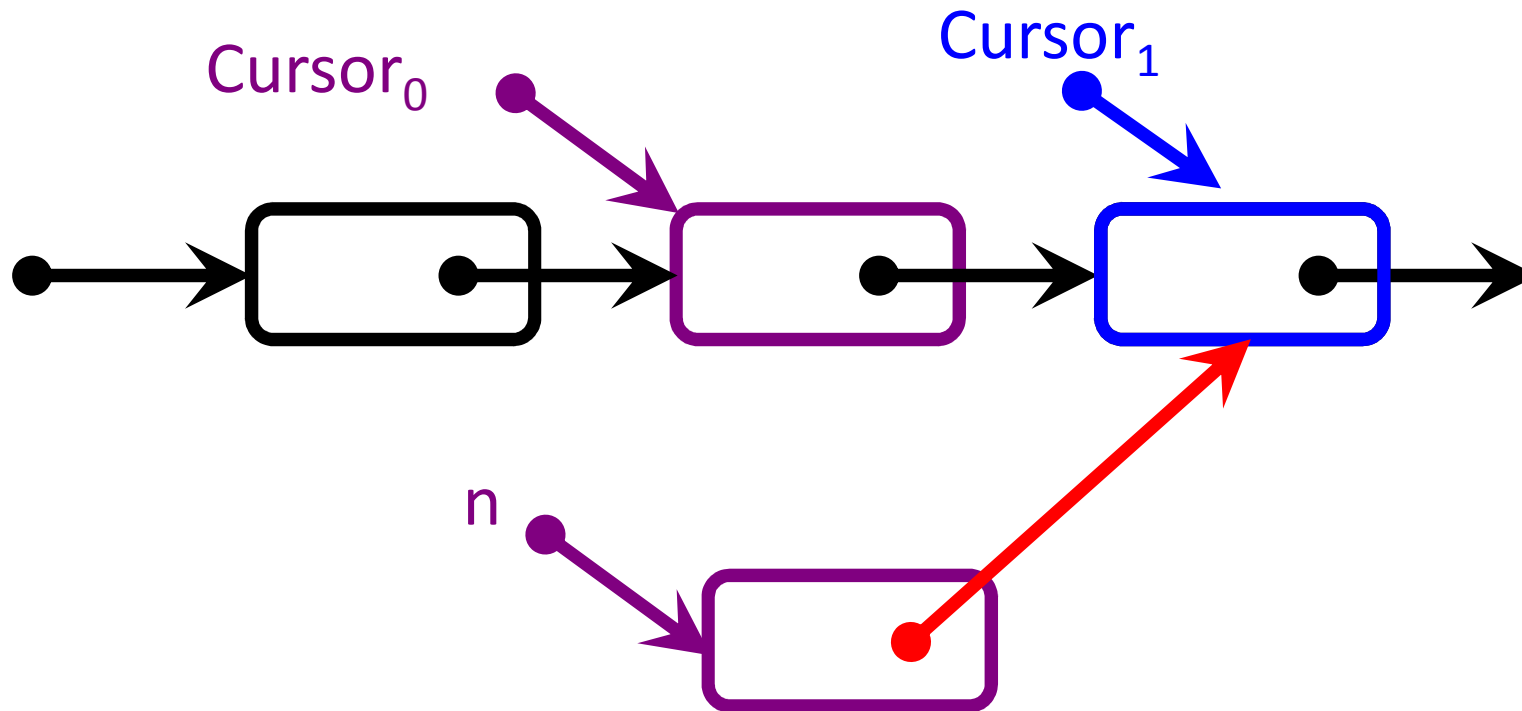
```
Oldfront =  
    AtomicInc (&front);  
if (Oldfront <= End)  
    WorkOn (oldFront);
```

**Thief:**

```
myEnd = End;  
End = (myEnd-front) / 2  
Myfront = front;  
updateMyGlobals ();  
ProcessMyShare ();
```

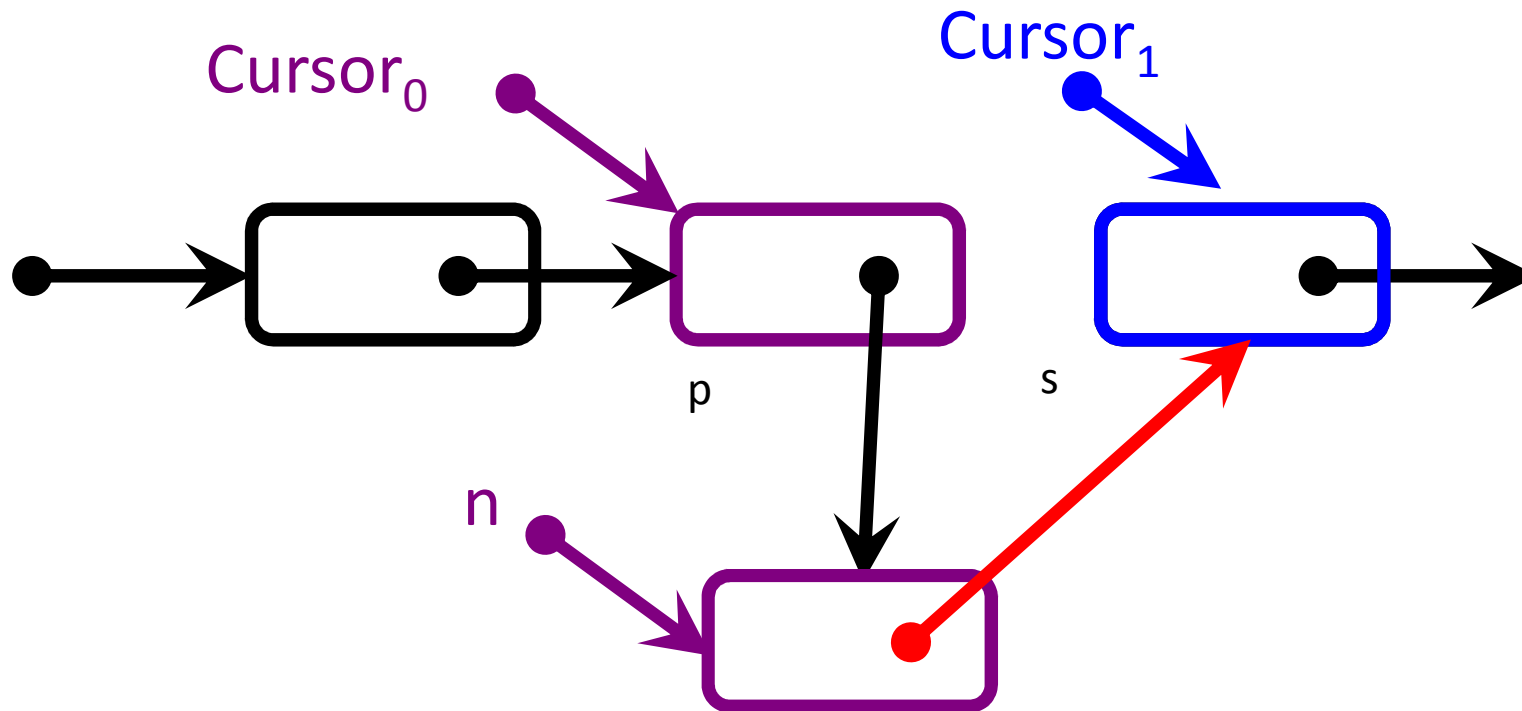
# Lock-free Linked List

- Insertion: Switch in the new node atomically



# Lock-free Linked List

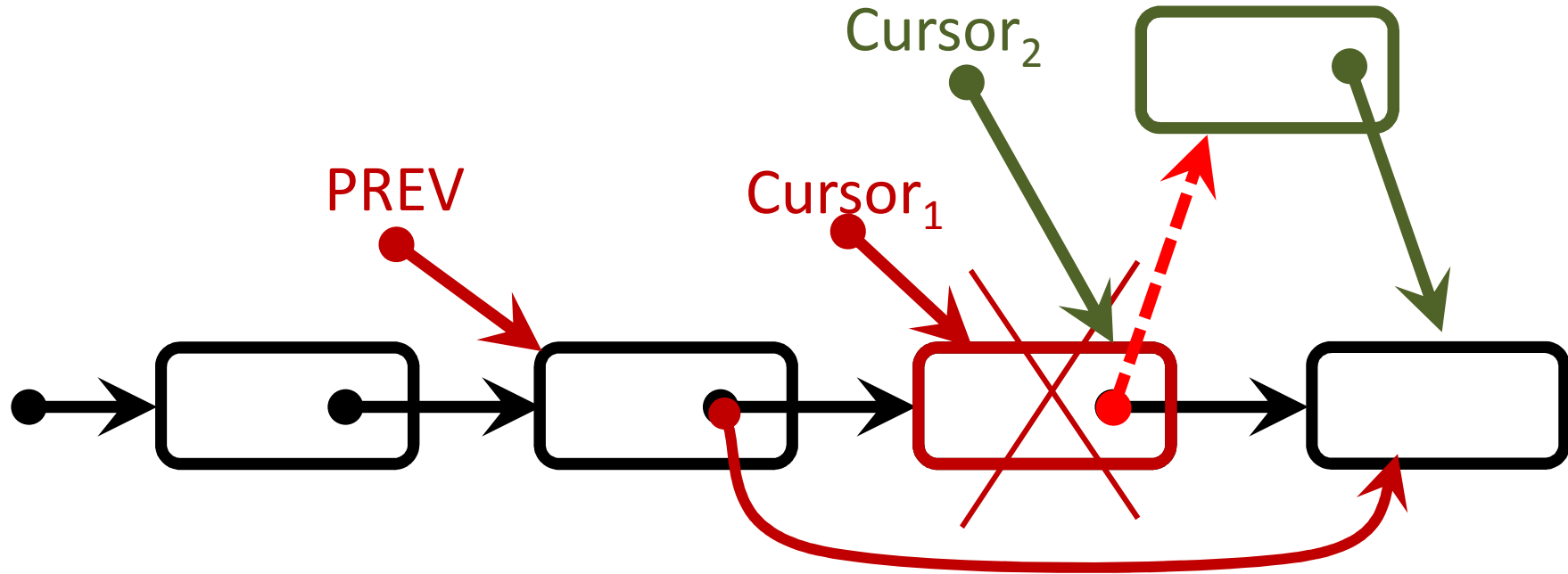
- Insertion: Switch in the new node atomically



But what if a concurrent  
delete(Cursor<sub>1</sub>) happened?

```
n->next = Cursor0->next  
CAS (&Cursor0->next, n->next, n)
```

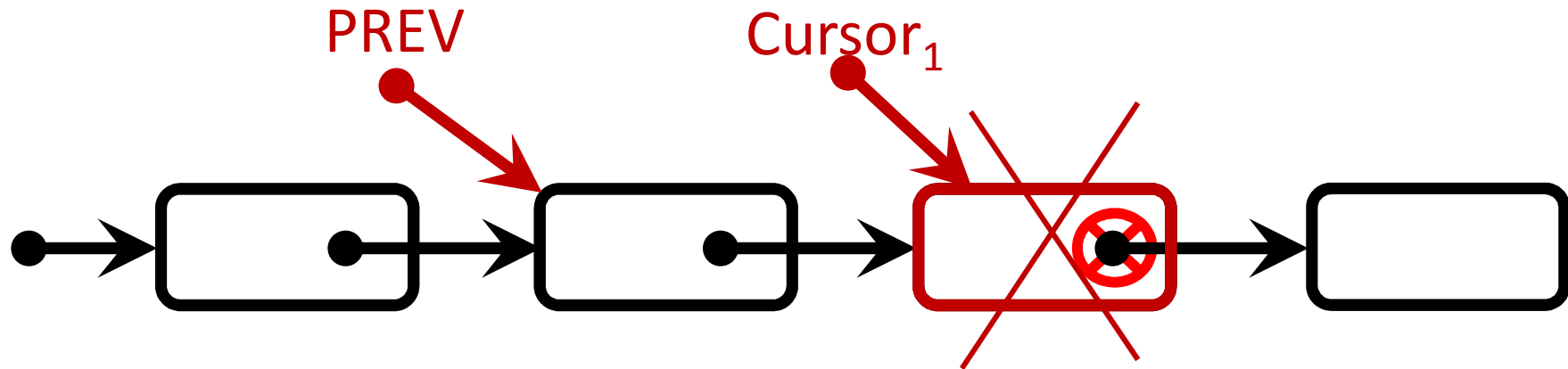
# Deletion



But what if, say, a concurrent  
`insertAfter(Cursor2)` happened?

[Harris 01] uses markers to get  
past transient states

# Deletion [Harris 01]



```
NEXT = Cursor1.next ;  
CAS (&Cursor1.next, NEXT, NEXT|MARK)
```

And then:

```
CAS(&(PREV.next), Cursor1, NEXT)
```

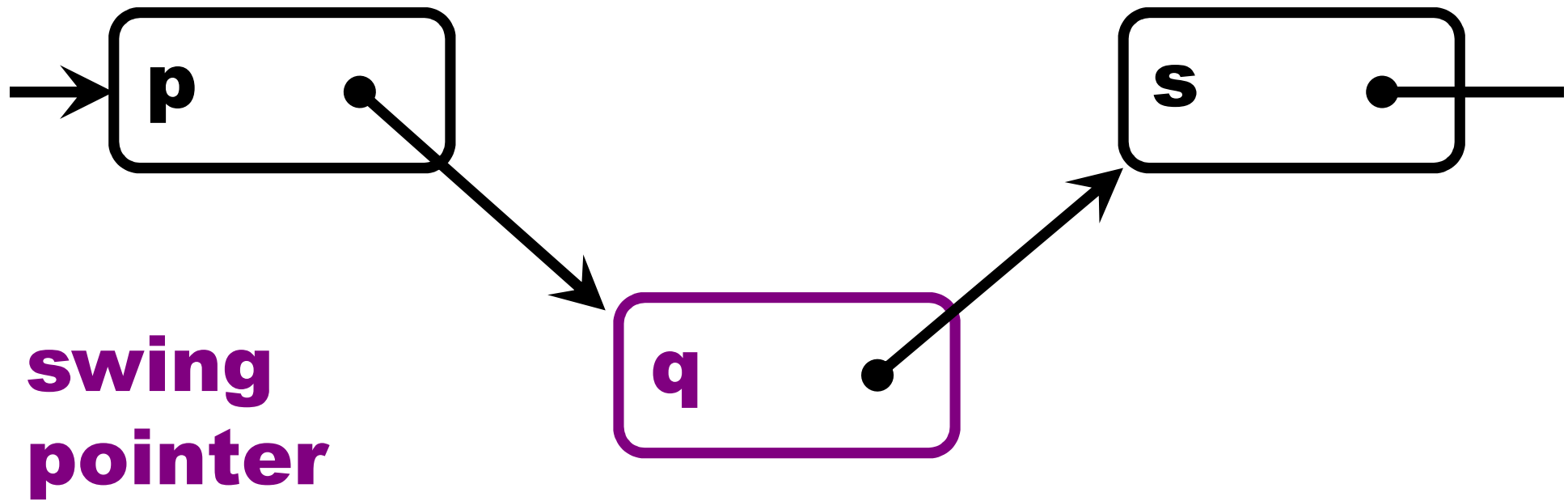
Can something go wrong in between?

# Deletion [Harris 01]

```
do {
    update(&curr, &prev);
    Node *curr_next = curr.next;
    if (! marked_bit(curr_next)) // If marked, retry
        if (CAS(&curr.next, curr_next, mark(curr_next)))
            break; // Was able to mark
} while (true);
// Now fix list
if (!CAS(&(prev.next), curr, curr_next))
    Update(&curr, &prev); // also deletes marked nodes
return true;
```



# ABA problem



# ABA Solutions

- Double Compare&Swap
- No Cell Reuse
- Memory Management

# Insert ( p, x )

- q = new cell
- Repeat
  - r = SafeRead ( p -> next )
  - Write ( q -> next, r )
- until Compare&Swap( p -> next, r, q )

# struct Cursor {

- node \* target; // -> data
- node \* pre\_aux; // -> preceding auxiliary  
node
- node \* pre\_cell; // -> previous cell  
};

Update(cursor c) {

- // Updates pointers in the cursor so that it becomes valid.
- // removes double aux\_node.

};

# Try\_delete(cursor c) {

- c.pre\_cell = next // deletes cell
- back\_link = c->pre\_cell
- delete pre\_aux
- Concurrent deletions may stall process and create chains of aux nodes.
- The last deletion follows the back\_links of the deleted cells.
- After all deletions the list will have no extra aux\_nodes

};