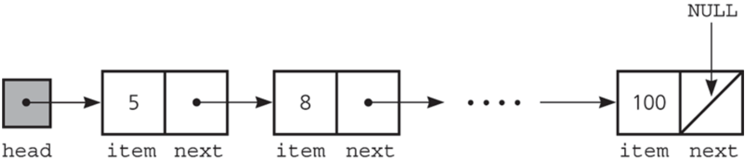
IE 411: Operating Systems

Linked lists: locking and lock-free

Concurrent data structures

- For a data structure, we would like multiple local (independent) operations to be allowed concurrently
- Can use locks to achieve thread-safe access
- But let's see if we can do thread-safe access without any locks at all
- We will illustrate the main ideas using linked lists

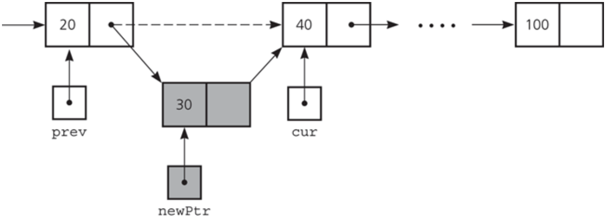
A sorted linked list



A sorted linked list

- To insert a node search for the insertion point (cur, prev)

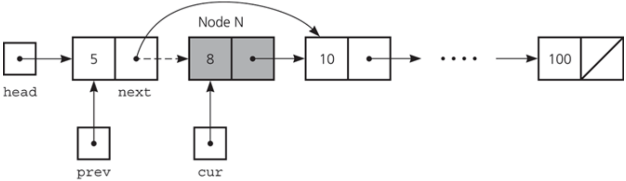
```
newPtr->next = cur;
prev->next = newPtr;
```



A sorted linked list

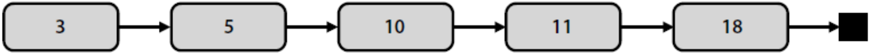
- To delete search for node

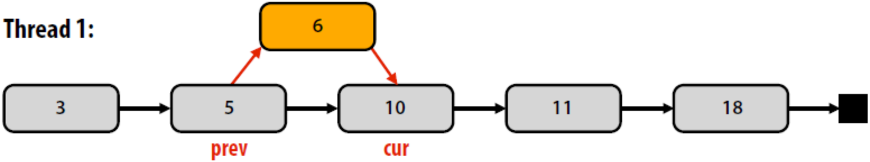
```
prev->next=cur->next;
```



Race condition

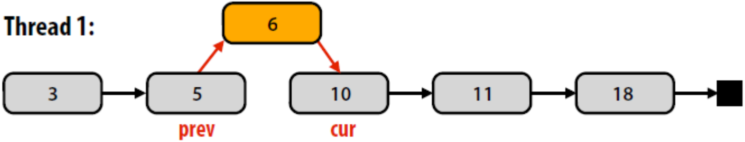
- Thread 1 attempts to insert 6
- Thread 2 attempts to insert 7



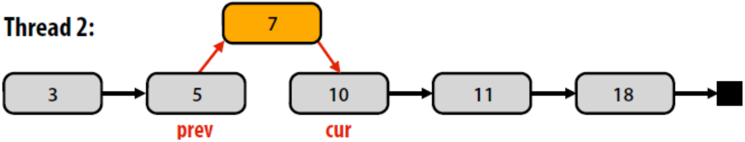


Race condition

Thread 1:

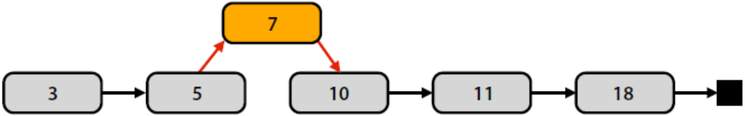


Thread 2:



Thread 1 and thread 2 both compute same prev and cur.
Result: one of the insertions gets lost!

Result: (assuming thread 1 updates `prev->next` before thread 2)



Single global lock

- Use a per-list lock
- Advantages
    - simple to implement

Single global lock

- Use a per-list lock
- Advantages
  - simple to implement
- Disadvantages?

Single global lock

- Use a per-list lock
- Advantages
  - simple to implement
- Disadvantages?
  - Operations on the data structure are serialized

Single global lock

- Use a per-list lock
- Advantages
  - simple to implement
- Disadvantages?
  - Operations on the data structure are serialized
  - May limit application performance

Lock-free algorithms

- protecting DS (e.g. BST, linked list) with a single lock is pessimistic as it assumes conflicts will occur
- a lockless algorithm is optimistic as it assumes conflicts unlikely to occur and, when they are detected, they are resolved

Lock-free algorithms

- protecting DS (e.g. BST, linked list) with a single lock is pessimistic as it assumes conflicts will occur
- a lockless algorithm is optimistic as it assumes conflicts unlikely to occur and, when they are detected, they are resolved
- Advantages compared to locking?

Lock-free algorithms

- protecting DS (e.g. BST, linked list) with a single lock is pessimistic as it assumes conflicts will occur
- a lockless algorithm is optimistic as it assumes conflicts unlikely to occur and, when they are detected, they are resolved
- Advantages compared to locking?
  - allows concurrency while there are no conflicts which hopefully is so most of the time

Atomic Compare-and-Swap (CAS)

```
bool CAS(
     memory location L,
     expected value V at L,
     desired new value V1 at L
);
```

If (the expected value V at memory location L == the current value at L), CAS succeeds by storing the the desired value V1 at L and returns TRUE.

Adding nodes

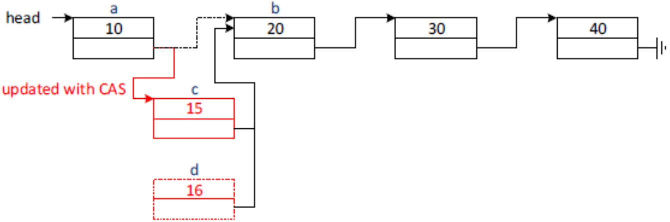
- One list, three nodes: `a`, `d`, `e`
- To insert node `c`
  1. `c->next = d`
  2. ATOMICALLY

```
if (a->next == d)
   a->next = c
else
   fail
```

- This translates into `CAS(&a->next, d, c)`
- CAS succeeded: `c` was successfully inserted between `a` and `d`
- CAS failed: retry

Adding nodes

- if 2 threads try to add nodes at the same position

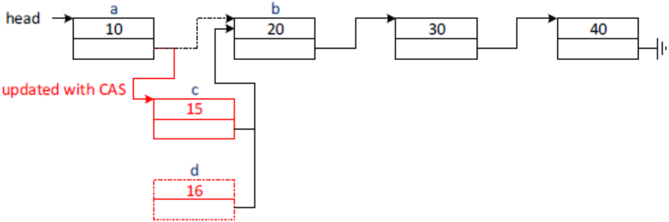


```
CAS(&a->next, b, c); // first CAS executed will succeed..
CAS(&a->next, b, d); // and thus second CAS executed will FAIL
```

- first CAS executed succeeds, second will fail as `a->next != b`

Adding nodes

- if 2 threads try to add nodes at the same position

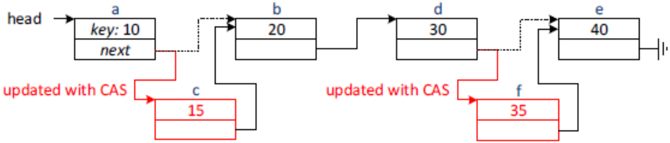


```
CAS(&a->next, b, c); // first CAS executed will succeed..
CAS(&a->next, b, d); // and thus second CAS executed will FAIL
```

- first CAS executed succeeds, second will fail as `a->next != b`
- RETRY on failure, which means searching for insertion point AGAIN and, if key not found, set up and re-execute CAS

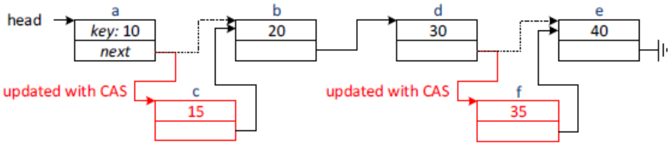
Adding nodes

- Ex: use CAS to add nodes 15 and 35



Adding nodes

- Ex: use CAS to add nodes 15 and 35



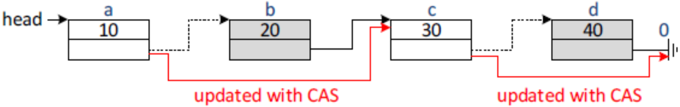
- search for insertion point, initialise next pointer and then execute with correct parameters to insert node into list

```
CAS(&a->next, b, c);    // add node c between a and b
CAS(&d->next, e, f);    // add node f between d and e
```

- disjoint-access parallelism

Removing nodes

- search for node and then execute CAS with correct parameters to remove node from list
- consider 2 threads removing non-adjacent nodes

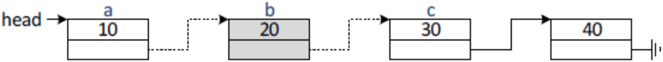


```
CAS(&a->next, b, c); // remove node b (20)
CAS(&c->next, d, 0); // remove node d (40)
```

- disjoint access parallelism

Removing nodes

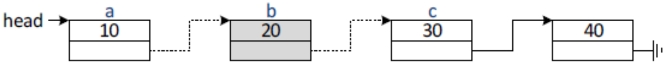
- if two threads try to remove the same node



```
CAS(&a->next, b, c);
CAS(&a->next, b, c);
```

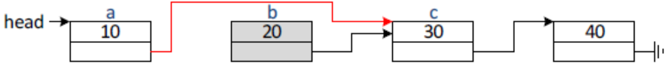
Removing nodes

- if two threads try to remove the same node



```
CAS(&a->next, b, c);
CAS(&a->next, b, c);
```

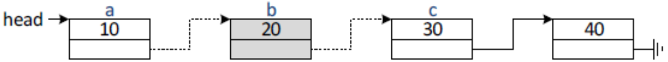
- assume first CAS executed succeeds



- then second CAS executed fails as `a->next != b`

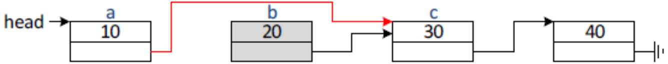
Removing nodes

- if two threads try to remove the same node



```
CAS(&a->next, b, c);
CAS(&a->next, b, c);
```

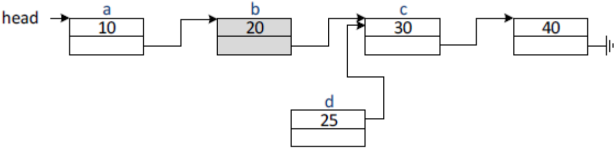
- assume first CAS executed succeeds



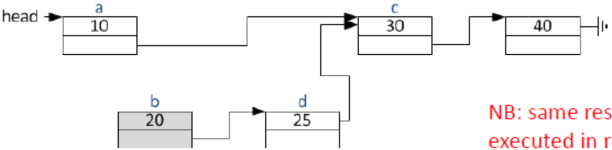
- then second CAS executed fails as `a->next != b`
- RETRY on failure, which means searching AGAIN for node (which may not be found)

What doesn't work...

- consider removing node 20 and adding node 25 concurrently



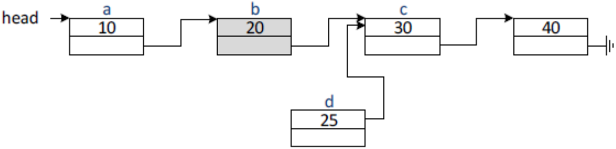
```
CAS(&a->next, b, c);   // remove 20
CAS(&b->next, c, d);   // add 25
```



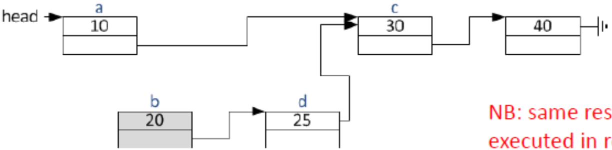
NB: same result if CAS instructions executed in reverse order

What doesn't work...

- consider removing node 20 and adding node 25 concurrently



```
CAS(&a->next, b, c);  // remove 20
CAS(&b->next, c, d);  // add 25
```

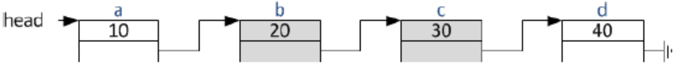


NB: same result if CAS instructions executed in reverse order

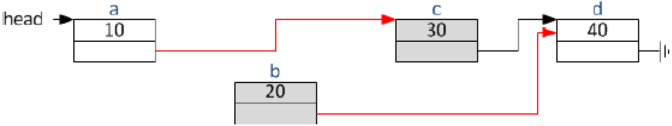
- NOT what was intended!

What doesn't work...

- imagine deleting adjacent nodes



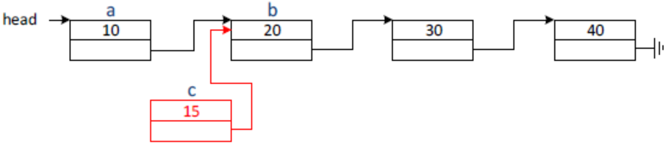
```
CAS(&a->next, b, c);  // remove 20
CAS(&b->next, c, d);  // remove 30
```



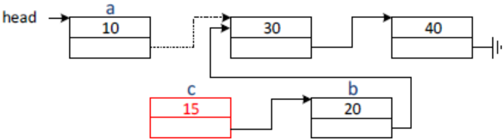
- AGAIN NOT what was intended!

ABA Problem

- imagine insertion point found, BUT before `CAS(&a->next, b, c)` is executed, thread is pre-empted

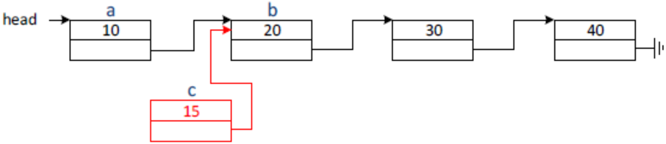


- another thread then removes b from list

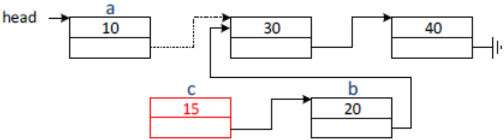


ABA Problem

- imagine insertion point found, BUT before `CAS(&a->next, b, c)` is executed, thread is pre-empted



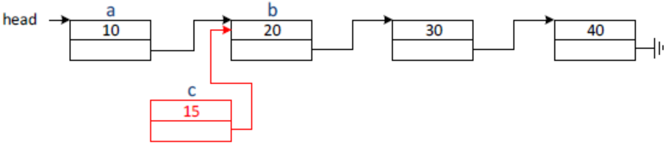
- another thread then removes b from list



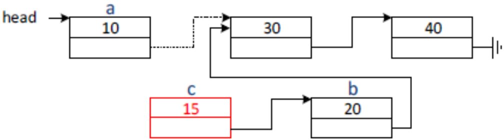
- if thread adding 15 resumes execution, the CAS fails which is OK in this case

ABA Problem

- imagine insertion point found, BUT before `CAS(&a->next, b, c)` is executed, thread is pre-empted



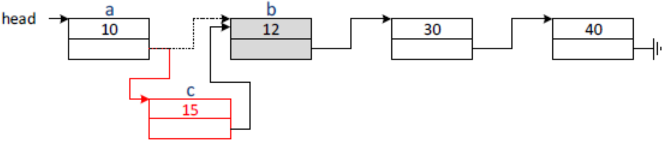
- another thread then removes b from list



- if thread adding 15 resumes execution, the CAS fails which is OK in this case
- BUT what bad thing can happen?

ABA Problem

- if the memory used by b is reused, for example by a thread adding key 12 to the list before thread adding 15 resumes . . .
- when the thread adding 15 to list resumes, its CAS will succeed and 15 will be added into the list at the wrong position



ABA Problem

- avoid the ABA problem by not reusing nodes:
    - nodes cannot be reused if any thread has or can get a pointer to the node

ABA Problem

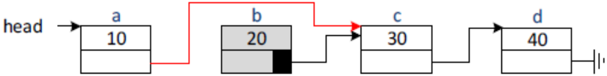
- avoid the ABA problem by not reusing nodes:
  - nodes cannot be reused if any thread has or can get a pointer to the node
- Disadvantages?

ABA Problem

- avoid the ABA problem by not reusing nodes:
  - nodes cannot be reused if any thread has or can get a pointer to the node
- Disadvantages?
  - will quickly run out of memory

Marked nodes

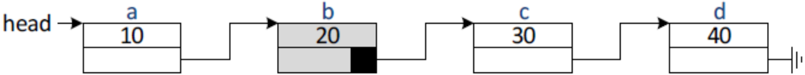
- Use two step removal e.g. remove(20)



1. atomically mark node by setting LSB of next pointer (logically removes node)
2. remove node by updating next pointer using CAS

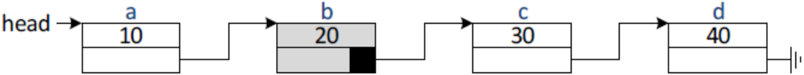
Marked nodes

- Marked node indicated by an ODD address in its next field
    - OK as addresses normally aligned on at least 4 byte boundary [2 or 3 LSBs normally 0]
- e.g., to atomically mark node b [logically remove]



Marked nodes

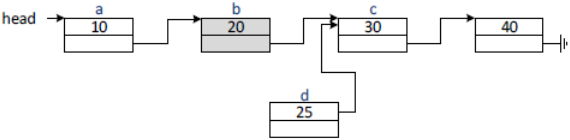
- Marked node indicated by an ODD address in its next field
  - OK as addresses normally aligned on at least 4 byte boundary [2 or 3 LSBs normally 0]
- e.g., to atomically mark node b [logically remove]



```
CAS(&b->next, c, c+1) //assumes node UNMARKED
```

Revisit: adding node and removing node

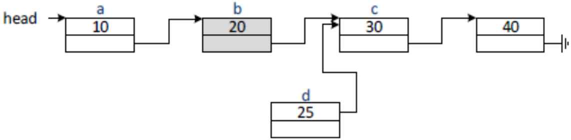
- imagine adding node [25] and removing node [20] concurrently



```
(1) CAS(&b->next, c, d);   // add 25
    and
(2) if (CAS(&b->next, c, c+1) == 1) // MARK node b and then
(3)     CAS(&a->next, b, c);        // remove b [20]
```

Revisit: adding node and removing node

- imagine adding node [25] and removing node [20] concurrently

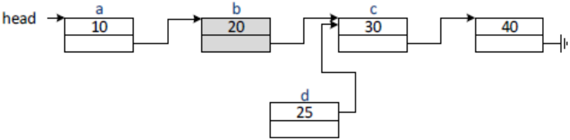


```
(1) CAS(&b->next, c, d);   // add 25
    and
(2) if (CAS(&b->next, c, c+1) == 1) // MARK node b and then
(3)     CAS(&a->next, b, c);        // remove b [20]
```

- if (1) executed first, (2) will fail as `b->next != c`

Revisit: adding node and removing node

- imagine adding node [25] and removing node [20] concurrently

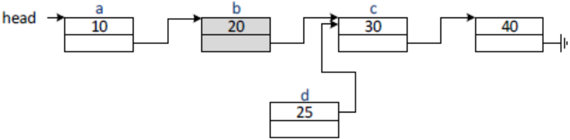


```
(1) CAS(&b->next, c, d);   // add 25
    and
(2) if (CAS(&b->next, c, c+1) == 1) // MARK node b and then
(3)     CAS(&a->next, b, c);        // remove b [20]
```

- if (1) executed first, (2) will fail as `b->next != c`
- if (2) executed first, (1) will fail as `b->next != c`

Revisit: adding node and removing node

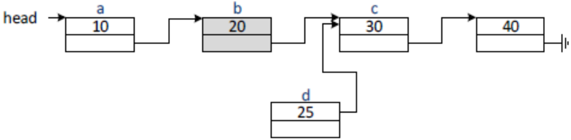
- imagine adding node [25] and removing node [20] concurrently



```
(1) CAS(&b->next, c, d);   // add 25
    and
(2) if (CAS(&b->next, c, c+1) == 1) // MARK node b and then
(3)     CAS(&a->next, b, c);        // remove b [20]
```

Revisit: adding node and removing node

- imagine adding node [25] and removing node [20] concurrently



```
(1) CAS(&b->next, c, d);   // add 25
    and
(2) if (CAS(&b->next, c, c+1) == 1) // MARK node b and then
(3)     CAS(&a->next, b, c);        // remove b [20]
```

- if (3) fails, it means that a no longer points to b, BUT b is logically marked and can be removed later
  - OK for list to contain temporary marked nodes

What still needs to be done?

- Previous solution avoids ABA problem by NOT re-using nodes
    - there is no code for freeing or reusing nodes

What still needs to be done?

- Previous solution avoids ABA problem by NOT re-using nodes
  - there is no code for freeing or reusing nodes
- Solutions with memory management:
  - A Pragmatic Implementation of Non-Blocking Linked Lists, Tim Harris, 2001
  - Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects, Maged M. Michael, 2004