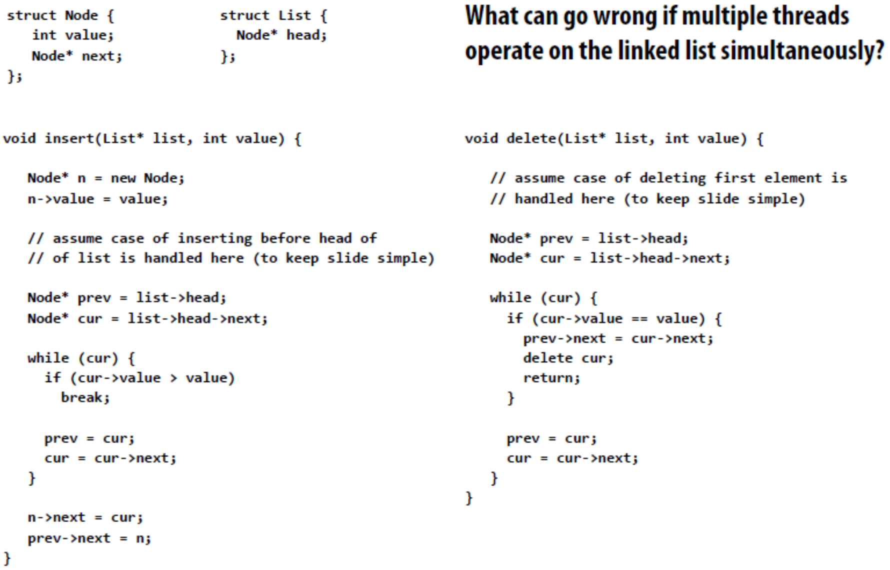
IE 411: Operating Systems

Non-blocking linked lists

A sorted linked list



```
struct Node {
   int value;
   Node* next;
};
```

```
struct List {
  Node* head;
};
```

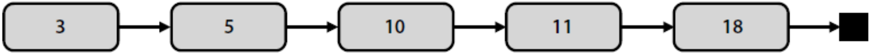
What can go wrong if multiple threads operate on the linked list simultaneously?

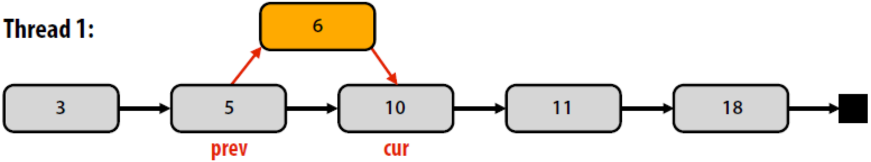
```
void insert(List* list, int value) {

   Node* n = new Node;
   n->value = value;

   // assume case of inserting before head of
   // of list is handled here (to keep slide simple)

   Node* prev = list->head;
   Node* cur = list->head->next;

   while (cur) {
     if (cur->value > value)
       break;

     prev = cur;
     cur = cur->next;
   }

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

```
void delete(List* list, int value) {

   // assume case of deleting first element is
   // handled here (to keep slide simple)

   Node* prev = list->head;
   Node* cur = list->head->next;

   while (cur) {
     if (cur->value == value) {
       prev->next = cur->next;
       delete cur;
       return;
     }

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

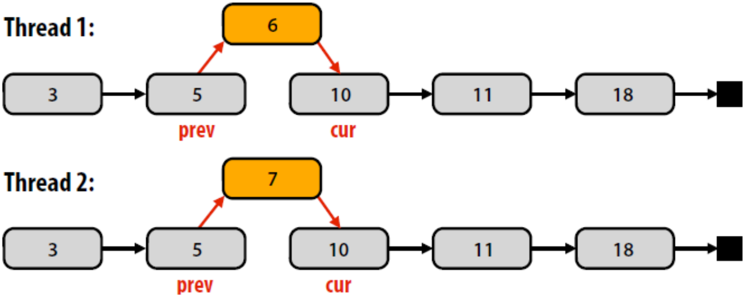
Example: concurrent insertion

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





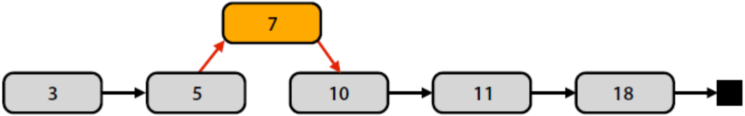
Example: concurrent insertion


Thread 1:

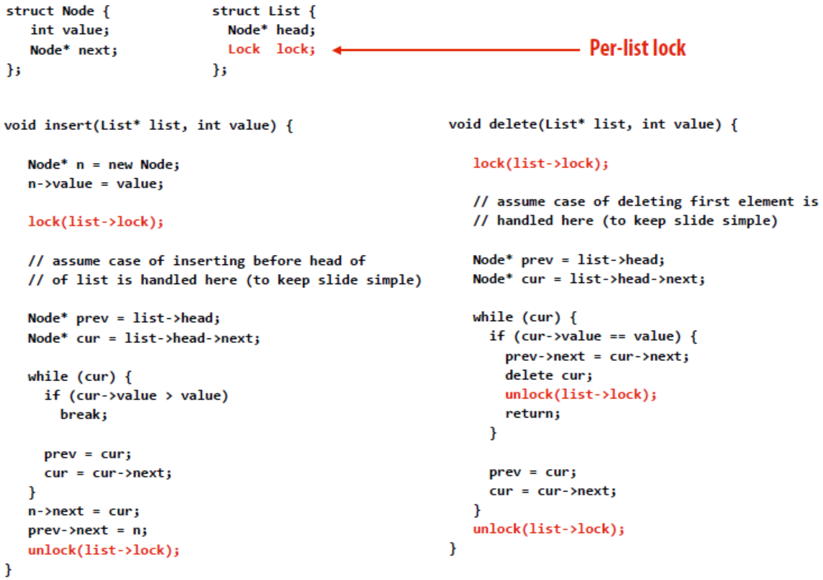
Thread 2:

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)



Solution 1: global locking



```
struct Node {           struct List {
   int value;             Node* head;
   Node* next;            Lock  lock;
};                      };
```

Per-list lock

```
void insert(List* list, int value) {

   Node* n = new Node;
   n->value = value;

   lock(list->lock);

   // assume case of inserting before head of
   // of list is handled here (to keep slide simple)

   Node* prev = list->head;
   Node* cur = list->head->next;

   while (cur) {
     if (cur->value > value)
       break;

     prev = cur;
     cur = cur->next;
   }
   n->next = cur;
   prev->next = n;
   unlock(list->lock);
}
```

```
void delete(List* list, int value) {

   lock(list->lock);

   // assume case of deleting first element is
   // handled here (to keep slide simple)

   Node* prev = list->head;
   Node* cur = list->head->next;

   while (cur) {
     if (cur->value == value) {
       prev->next = cur->next;
       delete cur;
       unlock(list->lock);
       return;
     }

     prev = cur;
     cur = cur->next;
   }
   unlock(list->lock);
}
```

Single global lock

- Advantages
    - simple to implement

Single global lock

- Advantages
    - simple to implement
- Disadvantages?

Single global lock

- Advantages
  - simple to implement
- Disadvantages?
  - Operations on the data structure are serialized

Single global 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

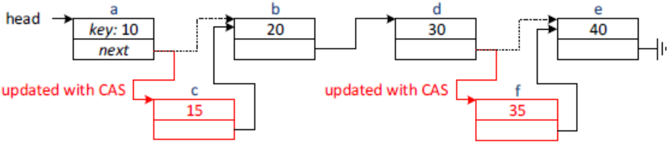
Atomic Compare-and-Swap (CAS)

```
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.

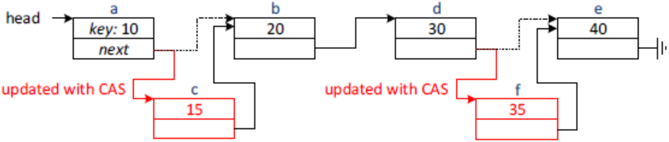
Adding nodes

- Ex: use CAS to add nodes 15 and 35



Adding nodes

- Ex: use CAS to add nodes 15 and 35



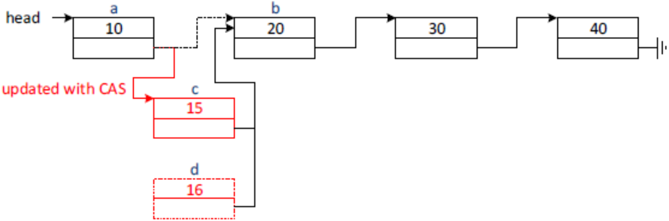
- search for insertion point, initialise next pointer and then execute with correct parameters to insert node into list

```
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

Adding nodes

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

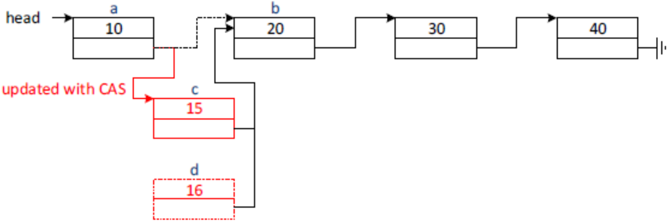


```
CAS(&a->next, b, c); // first CAS executed will succeed..
CAS(&a->next, b, d); // and thus second CAS executed will FAIL
```

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

Adding nodes

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

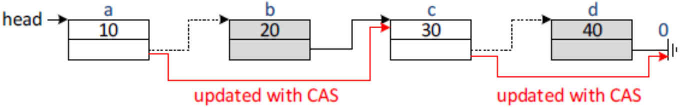


```
CAS(&a->next, b, c); // first CAS executed will succeed..
CAS(&a->next, b, d); // and thus second CAS executed will FAIL
```

- 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

Removing nodes

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

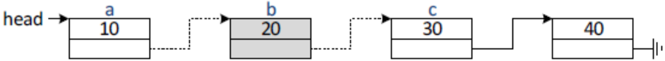


```
CAS(&a->next, b, c); // remove node b (20)
CAS(&c->next, d, 0); // remove node d (40)
```

- disjoint access parallelism

Removing nodes

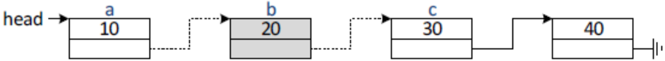
- if two threads try to remove the same node



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

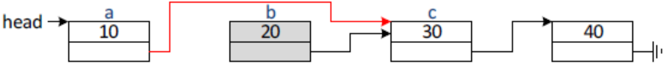
Removing nodes

- if two threads try to remove the same node



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

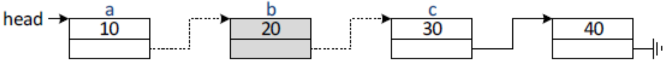
- assume first CAS executed succeeds



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

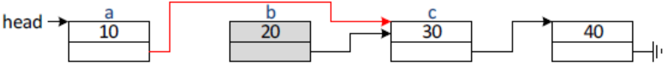
Removing nodes

- if two threads try to remove the same node



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

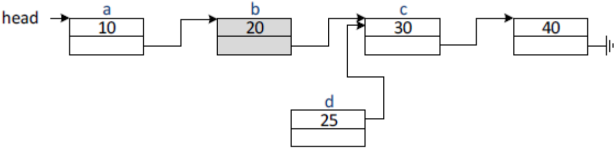
- assume first CAS executed succeeds



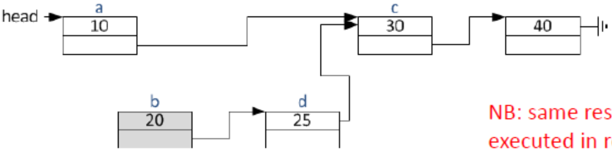
- then second CAS executed fails as `a->next != b`
- RETRY on failure, which means searching AGAIN for node (which may not be found)

What doesn't work...

- consider removing node 20 and adding node 25 concurrently



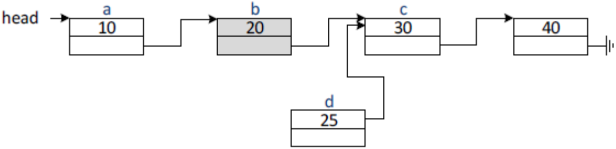
```
CAS(&a->next, b, c);   // remove 20
CAS(&b->next, c, d);   // add 25
```



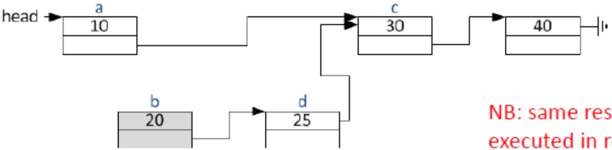
NB: same result if CAS instructions executed in reverse order

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
```

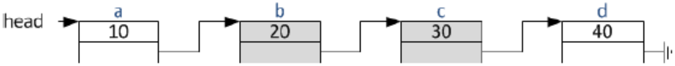


NB: same result if CAS instructions executed in reverse order

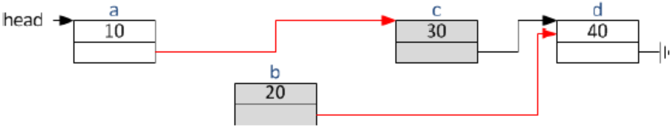
- NOT what was intended!

What doesn't work...

- imagine deleting adjacent nodes



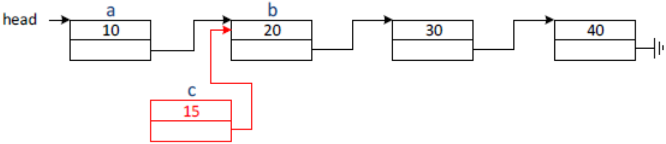
```
CAS(&a->next, b, c);   // remove 20
CAS(&b->next, c, d);   // remove 30
```



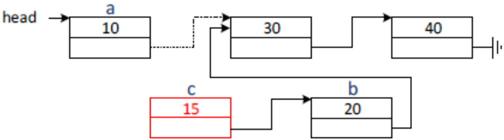
- AGAIN NOT what was intended!

ABA Problem

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

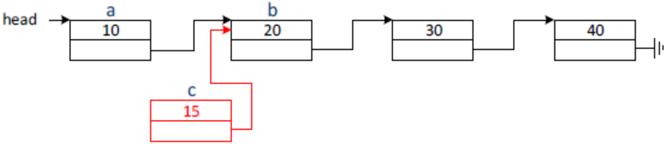


- another thread then removes b from list

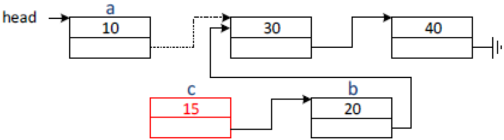


ABA Problem

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



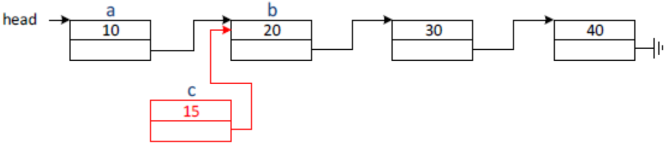
- another thread then removes b from list



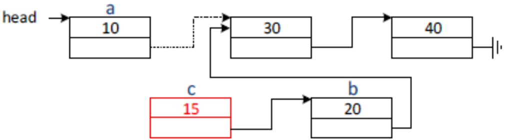
- if thread adding 15 resumes execution, the CAS fails which is OK in this case

ABA Problem

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



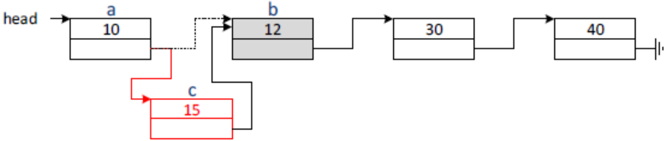
- another thread then removes b from list



- 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