Concurrent Data Structure

Jinzhen Wang
jwang96@charlotte.edu

Department of Computer Science
UNC Charlotte

Learning Outcomes

At the end of this lecture, you will be able to

- Make data structure thread safe
- Show a trade-off between fine-grain locking and coarse-grain locking
- Give a cause of why instructions might not be executed in the order they are written
- Name one atomic operation
- Implement one example of a lock-free data structure

Outline

One lock to solve them all

A simple issue

A typical data race on a data structure.

```
Set s;
s.add(something);
```

One lock to solve them all

A simple issue

A typical data race on a data structure.

```
Set s;
s.add(something);
```

A simple solution

Mutual exclusion works every time.

```
Set s;
mut.lock();
s.add(something);
mut.unlock();
```

Good Software Engineering

Thread safe objects

```
template <typename T>
class Threadsafe_set {
  std::set<T> s;
  std::mutex mut;

  void add (T& something) {
    std::lock_guard<std::mutex>(mut);
    s.add(something);
  }
  //...
};
```

Some libraries have different data structures for thread-safe and thread-safe data structures. For instance `Vector` and `ArrayList` in java.

What if waiting is necessary?

Suppose you have a producer-consumer problem implemented using a queue.

Using only mutexes

```
pthread_mutex_t mut;
while(queue.empty()) {
  pthread_mutex_unlock(&mut);
  sched_yield(); // give up the CPU
  pthread_mutex_lock(&mut);
};
```

What if waiting is necessary?

Suppose you have a producer-consumer problem implemented using a queue.

Using only mutexes

```
pthread_mutex_t mut;
while(queue.empty()) {
  pthread_mutex_unlock(&mut);
  sched_yield(); // give up the CPU
  pthread_mutex_lock(&mut);
};
```

Issues?

What if waiting is necessary?

Suppose you have a producer-consumer problem implemented using a queue.

Using only mutexes

```
pthread_mutex_t mut;
while(queue.empty()) {
  pthread_mutex_unlock(&mut);
  sched_yield(); // give up the CPU
  pthread_mutex_lock(&mut);
};
```

Issues?

This is busy-waiting. The thread keeps checking the condition in a loop and yields the CPU in between. This is inefficient and wastes CPU cycles.

What if waiting is necessary?

Let's try something else: Using Condition Variables

```
template <typename T>
class blocked_queue {
  std::queue < T > q;
  pthread_mutex_t mut;
  pthread_cont_t cond;

  void pop (T* popto) {
    pthread_mutex_lock(&mut);
    while(q.empty()) {
    pthread_cond_wait(&cond,
      &mut);
  }
  *popto = q.pop_front();
  pthread_mutex_unlock(&mut);
}
```

```
void push (cont T& pushit) {
  pthread_mutex_lock(&mut);
  q.push_back(pushit);
  pthread_cond_signal(&cond);
  pthread_mutex_unlock(&mut);
}
```

How to end applications? (Threads could be stuck waiting.)

- push dummy items to make threads get out. (This breaks semantic)

Clean queue exit

Using Condition Variables

```
template <typename T>
class blocked_queue {
  std::queue < T > q;
  pthread_mutex_t mut;
  pthread_cont_t cond;

  void pop (T* popto) {
    pthread_mutex_lock(&mut);
    while(q.empty()) {
    pthread_cond_wait(&cond,
      &mut);
  }
  *popto = q.pop_front();
  pthread_mutex_unlock(&mut);
}
```

```
void push (cont T& pushit) {
  pthread_mutex_lock(&mut);
  q.push_back(pushit);
  pthread_cond_signal(&cond);
  pthread_mutex_unlock(&mut);
}

void all_quit() {
  pthread_mutex_lock(&mut);
  quit = true;
  pthread_cond_broadcast(&cond);
  pthread_mutex_unlock(&mut);
}
}
```

Hash Tables

Hash Table

- A hash table is a data structure that maps keys to values using a hash function.
- The hash function computes an index from the key, which determines the bucket in the hash table.
- Hash tables provide average O(1) time complexity for insertions, deletions, and lookups.
- However, collisions can occur when multiple keys map to the same bucket.

Open Hashing

- Open hashing, also called separate chaining, is a collision resolution technique for hash tables.
- Each bucket in the hash table contains a linked list (or another collection) to store multiple elements that hash to the same bucket.
- When collisions occur, the new element is added to the collection in the corresponding bucket.

Hash Tables

- Simple collision handling: Easy to implement with linked lists or other collections.
- No fixed bucket size: Each bucket can dynamically grow to accommodate any number of elements.
- Efficient insertion and deletion: Adding or removing elements is straightforward.
- Minimal clustering: Open hashing reduces clustering compared to other methods like open addressing.

- Memory overhead: Each bucket stores a linked list or collection, leading to increased memory usage.
- Degraded performance under high load: If too many elements hash to the same bucket, performance can degrade to O(n) for search, insertion, and deletion operations.
- Non-constant time operations in buckets: Searching through a list in a bucket requires traversing the list, which can be inefficient if the list is long.

```
class OpenHashTable {
  std::vector<std::list<int>> table;
  int size;
public:
  OpenHashTable(int s) : size(s) {
    table.resize(size);
  }
  void insert(int key) {
    int index = key % size;
    table[index].push_back(key);
  }
  bool search(int key) {
    int index = key % size;
    for (int x : table[index]) {
      if (x == key) return true;
    }
    return false;
  }
};
```

Locking granularity for hash tables

- Hash tables are commonly used in multi-threaded environments.
- Locking mechanisms are used to ensure thread safety.
- **Locking granularity** refers to how much data a single lock protects.
- The choice of granularity affects both performance and correctness.

Coarse-Grained Locking

- A single lock protects the entire hash table.
- Simple to implement but leads to poor performance with many threads.
- All operations (insert, search, delete) must acquire the same lock.

Pros:

- Easy to implement.
- Ensures correctness with no race conditions.

Cons:

- Poor scalability.
- High lock contention in multi-threaded environments.

```
pthread_mutex_t table_lock;

void insert(int key, int value) {
  pthread_mutex_lock(&table_lock);
  // Insert item into hash table
  pthread_mutex_unlock(&table_lock);
}

void search(int key) {
  pthread_mutex_lock(&table_lock);
  // Search item in hash table
  pthread_mutex_unlock(&table_lock);
}
```

Fine-Grained Locking

- Each bucket in the hash table has its own lock.
- Allows multiple threads to work on different buckets concurrently.
- Reduces contention and increases parallelism.

Pros:

- Higher parallelism and better scalability.
- Reduced lock contention for large hash tables.

Cons:

- More complex to implement and manage multiple locks.
- Uneven contention if hash function isn't well-distributed.

```
class ConcurrentHashTable {
std::vector<std::mutex> bucket_locks;  //
    One lock per bucket
std::vector<std::list<std::pair<int, int
    >>> table;
public:
void insert(int key, int value) {
  int index = hash_function(key);
  std::lock_guard<std::mutex> lock(
      bucket_locks[index]);
  table[index].push_back({key, value});
}
void search(int key) {
  int index = hash_function(key);
  std::lock_guard<std::mutex> lock(
      bucket_locks[index]);
  // Search in table[index]
}
};
```

Lock-Free Hash Tables

- No locks are used. Instead, atomic operations (e.g., CAS) are employed.
- Threads operate on the hash table concurrently without waiting for locks.

Pros:

- Eliminates lock contention.
- High performance in high-concurrency environments.

Cons:

- Very complex to implement.
- Difficult to handle rehashing and resizing safely.

Hybrid Locking Strategies

- Combines both coarse-grained and fine-grained locking.
- Coarse-grained locking is used for operations like resizing the hash table.
- Fine-grained locking is used for operations like insertion, search, and deletion.

Pros:

- Balances performance and complexity.
- Reduces lock contention for frequent operations.

Cons:

- More complex to implement and maintain.

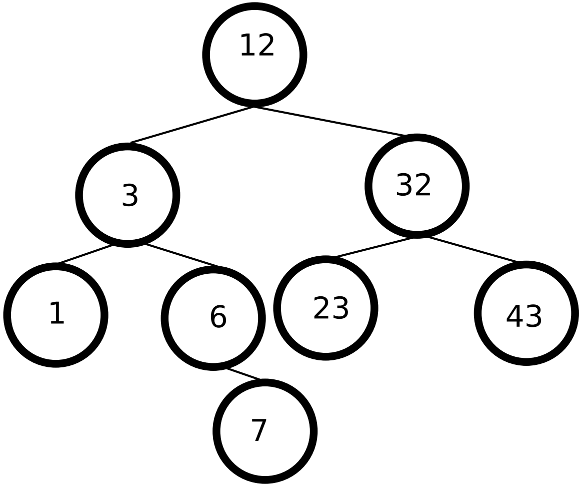
Conclusion

- Locking granularity in hash tables significantly impacts performance.
- Coarse-grained locking is simple but has poor scalability.
- Fine-grained locking improves scalability but adds complexity.
- Lock-free hash tables offer high performance but are difficult to implement.
- Hybrid strategies combine the benefits of both approaches.

Binary Search Trees

Properties

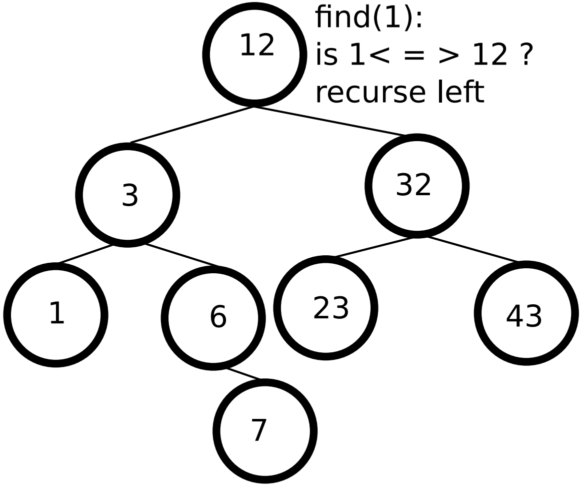
- Spatial partitioning tree
- Uses element partitioning
- Appropriate for 1D spaces
- Each node partition the object based on whether their location is less than or than the object in the node.
- A binary tree where recursively the following invariant holds: each node in the left (resp. right) subtree has a value less than (resp. greater than) the value of the root.



Binary Search Tree: *find*(*location*)

algorithm

- If $location = root.location$, then return the root object.
- If $location < root.location$, then $left.find(location)$
- If $location > root.location$, then $right.find(location)$

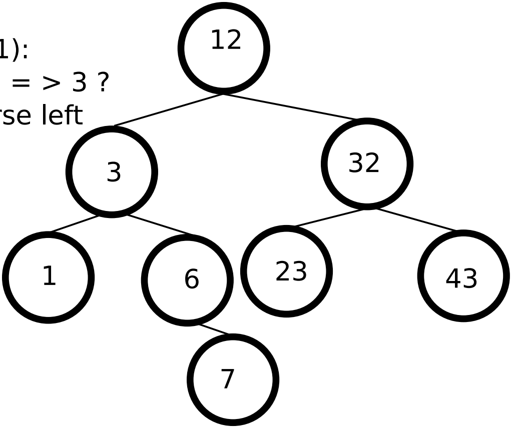


Binary Search Tree: *find*(*location*)

algorithm

- If $location = root.location$, then return the root object.
- If $location < root.location$, then $left.find(location)$
- If $location > root.location$, then $right.find(location)$

find(1):
is 1< = > 3 ?
recurse left

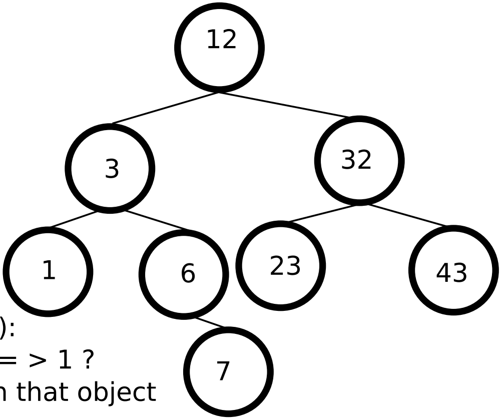


Binary Search Tree: *find*(*location*)

algorithm

- If $location = root.location$, then return the root object.
- If $location < root.location$, then $left.find(location)$
- If $location > root.location$, then $right.find(location)$

Complexity: $O(Depth)$



find(1):
is 1< = > 1 ?
Return that object

Binary Search Tree: *find*(*location*)

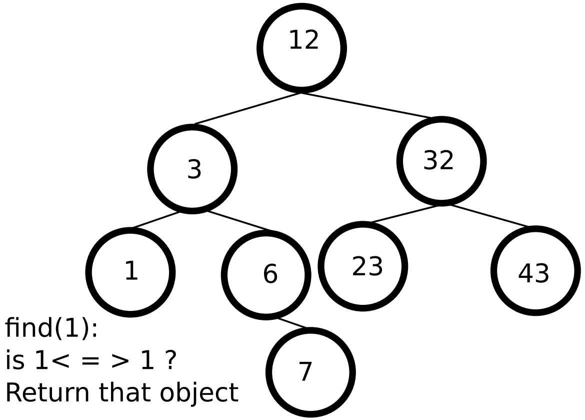
algorithm

- If $location = root.location$, then return the root object.
- If $location < root.location$, then $left.find(location)$
- If $location > root.location$, then $right.find(location)$

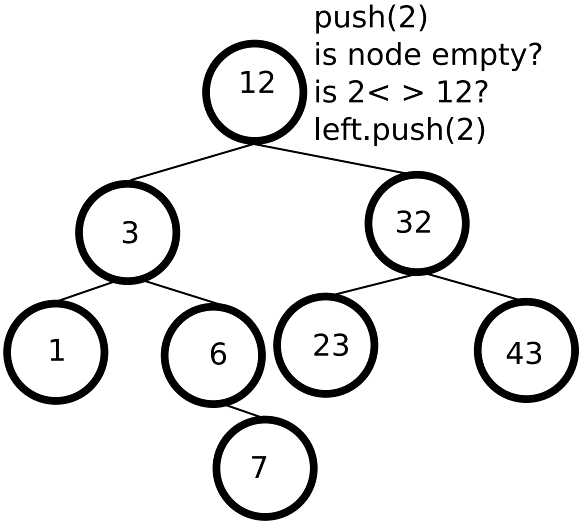
Complexity: $O(Depth)$

Fine print

- Handle cases where the root is empty, or a subtree is empty.



Binary Search Tree: *push*(*location*)



algorithm

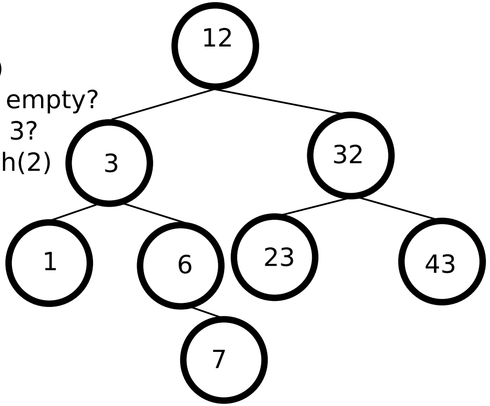
- If tree is empty, create it with that object
- If $location < root.location$, then $left.push(location)$
- If $location > root.location$, then $right.push(location)$

Binary Search Tree: *push*(*location*)

algorithm

- If tree is empty, create it with that object
- If $location < root.location$, then $left.push(location)$
- If $location > root.location$, then $right.push(location)$

push(2)
is node empty?
is 2< > 3?
left.push(2)

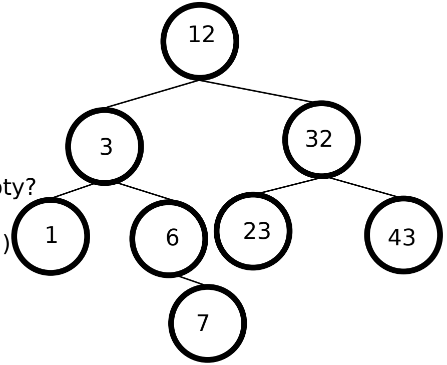


Binary Search Tree: *push*(*location*)

algorithm

- If tree is empty, create it with that object
- If $location < root.location$, then $left.push(location)$
- If $location > root.location$, then $right.push(location)$

push(2)
is node empty?
is 2< > 1?
right.push(2)

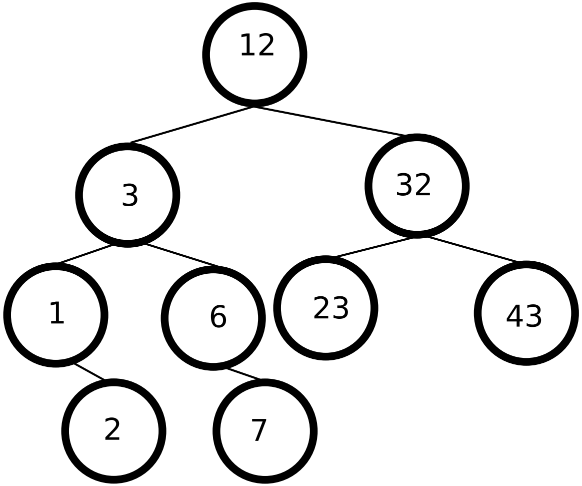


Binary Search Tree: *push*(*location*)

algorithm

- If tree is empty, create it with that object
- If $location < root.location$, then $left.push(location)$
- If $location > root.location$, then $right.push(location)$

Complexity: $O(Depth)$



Binary Search Tree: *push*(*location*)

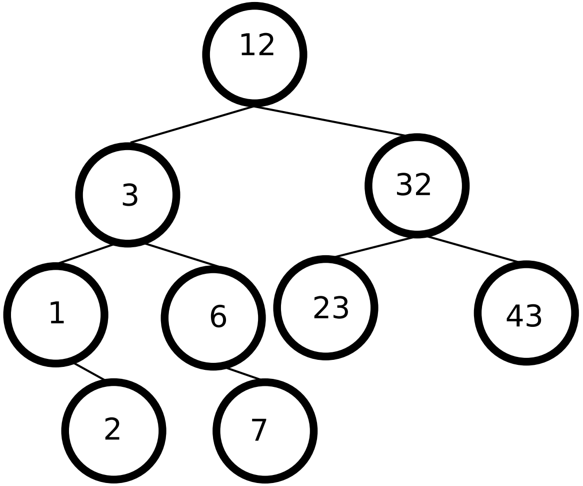
algorithm

- If tree is empty, create it with that object
- If $location < root.location$, then $left.push(location)$
- If $location > root.location$, then $right.push(location)$

Complexity: $O(Depth)$

Fine print

- Handle gracefully cases where the root is empty, or a subtree is empty.



Locking granularity for binary search trees

What if the set was implemented as a Binary Search Tree?

- One could lock per intermediate node
- The $\theta(log(n))$ locks would be held for $O(1)$ time instead of holding 1 lock for $\theta(log(n))$ time
- Could improve the execution

Locking granularity for binary search trees

What if the set was implemented as a Binary Search Tree?

- One could lock per intermediate node
- The $\theta(log(n))$ locks would be held for $O(1)$ time instead of holding 1 lock for $\theta(log(n))$ time
- Could improve the execution
- Maybe a different kind of tree, such as a k-ary tree, could reduce locking overhead.

Alternative strategies

What if each thread was working on its own set?

- Assume only add and no look-up during adds
- No locking overhead
- But you'll have a merging overhead once all adds are done

Alternative strategies

What if each thread was working on its own set?

- Assume only add and no look-up during adds
- No locking overhead
- But you'll have a merging overhead once all adds are done

Batched insertion semantic

Could insert element by batches instead of one at a time.

- Low locking overhead
- No need for final merge
- Assumes partial information in the structure is enough

Atomicity of reads and writes

How bad can things get when two threads execute `*p = 1;` or `v = *p;` ?

Atomicity of reads and writes

How bad can things get when two threads execute `*p = 1;` or `v = *p;` ?
Of course there is a data race.
But depending on the type of `*p` worse can happen.

- If `*p` is too small (less than 32-bit on x86), then 32-bit are read, `*p` is written at the correct location in a register and then the 32-bit are written.
- If `*p` is too large (depending on compiler and arch), then reads and writes happen per chunks.

Atomic operations

What is

Operation that is completed in a *single step*, without the possibility of interference from other operations or threads. It runs to completion without any other thread being able to observe it in a partially complete state. Critical for ensuring consistency and correctness in *concurrent programming* when multiple threads are accessing shared data.

Key features

- Indivisibility.
- Concurrency Safety.
- Hardware and Software Support.

Atomic operations

Many processors support atomic instructions.
The precise set depends on architecture.

Compare and Swap

CAS (A, val, newval) does atomically:

```
bool cnd = (*A == val);
if (cnd)
  *A = newval;
return cnd;
```

fetch-and-add

FAD(A, val) does atomically:

```
oldval = *A;
*A += val;
return oldval;
```

The volatile keyword

`volatile int c;` tells the compiler that the value of `c` might change without local code doing anything.

```
int c;

void waitOnC(){
  while (c != 0) {
    sleep(1);
  }
}
```

1. **Prevents Optimization:** Compilers often optimize code by keeping variables in registers for faster access. If a variable is declared as volatile, the compiler will read the variable from memory each time it is accessed and write to it immediately after modification, avoiding the use of potentially stale register values.
2. **Used in Multithreading and Hardware Interaction:**
   - **Multithreading:** If a variable is shared between threads, marking it as volatile ensures that one thread's changes are visible to other threads immediately.
   - **Hardware Interaction:** In embedded systems, variables connected to hardware registers (e.g., memory-mapped I/O) should be marked volatile because hardware can change these values asynchronously.
3. **Not a Thread Synchronization Mechanism:** volatile ensures visibility of changes, but it does not provide atomicity or memory ordering guarantees. For proper synchronization in multithreading, you often need additional mechanisms like mutexes or atomic operations.

Out-of-order execution and fences

Compilers reorder instructions

- Compilers optimize code for coherent sequential execution.
- It may change the order of the instruction.

```
data = x;
dataavailable = true;
```

What if these are swapped? Unintended consequence for multiple threads.

Out-of-order execution and fences

Compilers reorder instructions

- Compilers optimize code for coherent sequential execution.
- It may change the order of the instruction.

```
data = x;
dataavailable = true;
```

What if these are swapped? Unintended consequence for multiple threads.
`asm volatile("" ::: "memory");`
prevents GCC to reorder the instruction.

Out-of-order execution and fences

Compilers reorder instructions

- Compilers optimize code for coherent sequential execution.
- It may change the order of the instruction.

```
data = x;
dataavailable = true;
```

What if these are swapped? Unintended consequence for multiple threads.
`asm volatile("" ::: "memory");` prevents GCC to reorder the instruction.

Processors reorder instructions

Some processors (e.g., all x86) can execute instructions out-of-order.
Assembly instructions can be executed in an order different than the order they appear in the compiled code.
Of course, the processor guarantees the sequential correctness of these operations.
But one needs to be careful of memory operation reordering as well.
Different architectures have different memory models, so be careful for portability.

non-blocking, lock-free and wait-free operations

Blocking

A function is said to be blocking if it calls an operating system function that waits for an event to occur or a time period to elapse. Mutexes, Thread Join, I/O operations

non-blocking, lock-free and wait-free operations

Blocking

A function is said to be blocking if it calls an operating system function that waits for an event to occur or a time period to elapse. Mutexes, Thread Join, I/O operations

Non-blocking

The suspension of one thread can not indefinitely block others.
So if there is a mutex, the operation is not non-blocking.

non-blocking, lock-free and wait-free operations

Blocking

A function is said to be blocking if it calls an operating system function that waits for an event to occur or a time period to elapse. Mutexes, Thread Join, I/O operations

Non-blocking

The suspension of one thread can not indefinitely block others.
So if there is a mutex, the operation is not non-blocking.

Lock-free

Non-blocking and for any ordering execution and suspension, some global progress eventually happens. Spin-locks are not lock-free, why?

non-blocking, lock-free and wait-free operations

Blocking

A function is said to be blocking if it calls an operating system function that waits for an event to occur or a time period to elapse. Mutexes, Thread Join, I/O operations

Non-blocking

The suspension of one thread can not indefinitely block others.
So if there is a mutex, the operation is not non-blocking.

Lock-free

Non-blocking and for any ordering execution and suspension, some global progress eventually happens. Spin-locks are not lock-free, why?

Wait-freedom

Lock-free and guarantees that each operation completes in a bounded number of steps.

Examples

blocking

pthread_mutex_lock();
condition_wait();
fread/fwrite();
sleep();

Examples

blocking

pthread_mutex_lock();
condition_wait();
fread/fwrite();
sleep();

Non-blocking

spinlock()
busy-wait loops

Examples

lock-free

```
int curr;
...
for(;;){
  int old = cur;
  int next = curr + 1;
  if(cas(curr, old, next))
    return next;
}
```

Examples

lock-free

```
int curr;
...
for(;;){
  int old = cur;
  int next = curr + 1;
  if(cas(curr, old, next))
    return next;
}
```

wait-free

Very hard to implement.
wait-free queue and stack

General Approach to Lock-Free Algorithms

- Designing generalized lock-free algorithms is hard
- Design lock-free data structures instead (Buffer, list, stack, queue, map, deque, snapshot)
- Cannot implement lock-free algorithms in terms of lock-based data structures

Appending to a buffer

```
char* buffer;
char* buffer_end;

void append (char* data) {
  size_t len = strlen(data);
  char* pos = fetch_and_add (buffer_end, len);
  memcpy (pos, data, len);
}
```

Pay attention to having enough memory.

Appending to a linked list

```
struct node {
  int val;
  node* next;
};

void append (int val, node* n) {
  node* newnode = new node;
  newnode->val = val;
  newnode->next = NULL;
  bool done = false;
  while (!done) {
    while (n->next != NULL)
      n = n->next;
    done = compare_and_swap(&(n->next), NULL, newnode);
  }
}
```

Appending to a linked list

```
struct node {
  int val;
  node* next;
};

void append (int val, node* n) {
  node* newnode = new node;
  newnode->val = val;
  newnode->next = NULL;
  bool done = false;
  while (!done) {
    while (n->next != NULL)
      n = n->next;
    done = compare_and_swap(&(n->next), NULL, newnode);
  }
}
```

Works great for concurrent add and concurrent traversals.
Be careful with deletion.

Lock-free Stack

Data Structure

```
struct node {
  int val;
  node *next
} Node;
...
Node *head;
```

Lock-free Stack

Data Structure

```
struct node {
  int val;
  node *next
} Node;
...
Node *head;
```

Push

```
void push(Node * node){
  while (1){
    node->next = head;
    if (cas(&head,
            node->next,
            node))
      return;
} }
```

Lock-free Stack

Data Structure

```
struct node {
  int val;
  node *next
} Node;
...
Node *head;
```

Push

```
void push(Node * node){
  while (1){
    node->next = head;
    if (cas(&head,
          node->next,
          node))
      return;
} }
```

Any issue with this function?

Lock-free Stack

The key issue lies in how the CAS operation is used. The CAS operation compares the current value of the head with node→next and only succeeds if they are equal. However, node→next may no longer be the current head at the time of the CAS operation because another thread could have modified the stack (i.e., changed head) between the time you set node→next and the CAS operation.

Lock-free Stack

The key issue lies in how the CAS operation is used. The CAS operation compares the current value of the head with node→next and only succeeds if they are equal. However, node→next may no longer be the current head at the time of the CAS operation because another thread could have modified the stack (i.e., changed head) between the time you set node→next and the CAS operation.

```
void push(Node* node) {
  Node* old_head;
  while (1) {
    old_head = head;        // Capture the current head
    node->next = old_head;  // Set the new node's next pointer to the current head
    // Try to atomically swap the head with the new node
    if (cas(&head, old_head, node)) {
      return;  // If CAS succeeds, the push is complete
    }
    // If CAS fails, another thread modified the head, so retry
  }
}
```

Lock-free Stack

Pop

```
Node* pop() {
  Node* old_head;
  Node* new_head;
  while (1) {
    old_head = head;              // Capture the current head (top of the stack)

    if (old_head == nullptr) {    // Check if the stack is empty
      return nullptr;            // If empty, return nullptr (nothing to pop)
    }
    new_head = old_head->next;    // The new head will be the next node
    // Try to atomically update the head to the new head
    if (cas(&head, old_head, new_head)) {
      break;  // CAS succeeded, exit the loop
    }
    // If CAS failed, another thread modified the head, so retry
  }
  return old_head;  // Return the popped node (old head)
}
```

ABA Problem

The ABA problem is a common issue in concurrent programming, particularly in lock-free algorithms that use atomic operations like Compare-And-Swap (CAS). It occurs when a memory location is read twice, and in between the two reads, its value changes and then changes back to the original value. The CAS operation detects that the value is the same as it was originally, so it assumes no changes were made, even though the value has changed and potentially invalid actions have occurred.

ABA Problem

The ABA problem is a common issue in concurrent programming, particularly in lock-free algorithms that use atomic operations like Compare-And-Swap (CAS). It occurs when a memory location is read twice, and in between the two reads, its value changes and then changes back to the original value. The CAS operation detects that the value is the same as it was originally, so it assumes no changes were made, even though the value has changed and potentially invalid actions have occurred.

For example

- Thread 0 begins a pop and sees "A" as the top, followed by "B".
- Thread 1 begins and completes a pop, returning "A".
- Thread 1 begins and completes a push of "D".
- Thread 1 pushes "A" back onto the stack and completes.
- Thread 0 sees that "A" is on top and returns "A", setting the new top to "B".
- Node D is lost.

Why ABA probelm is dangerous

The ABA problem is particularly problematic in lock-free data structures (like stacks or queues), where changes in pointer values or states might make the system inconsistent. If a thread assumes that a value hasn't changed simply because it's back to A, it could lead to:

- Corrupted data: If changes were made and then undone, subsequent operations may behave incorrectly.
- Memory safety issues: If nodes in a linked list or stack were freed and reallocated, the ABA problem could lead to accessing invalid memory.

Addressing the ABA Problem:

Double CAS (DCAS)

Some architectures support double compare-and-swap, which allows the CAS operation to check and update two memory locations simultaneously (e.g., a value and its version tag). This helps detect changes more reliably.

Addressing the ABA Problem:

Double CAS (DCAS)

Some architectures support double compare-and-swap, which allows the CAS operation to check and update two memory locations simultaneously (e.g., a value and its version tag). This helps detect changes more reliably.

Solution 1:

```
Node * pop(){
  while (1){
    Node *curr = head;
    int pop_count = stack->pop_count;
    if (curr == NULL)
      return NULL;
    if (double_cas(&head, curr, curr->next,
        &stack->pop_count, pop_count, pop_count + 1))
      return curr;
  }
}
```

Addressing the ABA Problem:

Double CAS (DCAS)

Some architectures support double compare-and-swap, which allows the CAS operation to check and update two memory locations simultaneously (e.g., a value and its version tag). This helps detect changes more reliably.

Solution 2:

```
Node* pop() {
  TaggedPointer oldHead = head.load();
  while (true) {
    if (oldHead.ptr == nullptr) { return nullptr;} // Stack is empty
    Node* nextNode = oldHead.ptr->next;  // Next node in the stack
    TaggedPointer newHead = {nextNode, oldHead.version + 1};  // New head with
        updated version
    // Perform double CAS: both head pointer and version must match
    if (head.compare_exchange_weak(oldHead, newHead)) { // Successfully popped the
        node
      return oldHead.ptr;  // Return the popped node
    }
  }
```

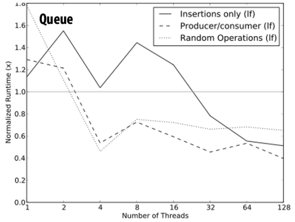
Performance

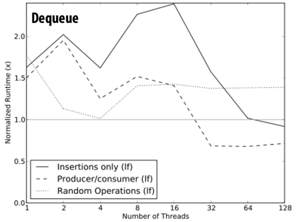
lockfree data structures don't necessarily show better performance, despite the apparent benefits.
In cases where a CAS operation will fail many times before succeeding, you end up doing a lot of extra busy waiting, similar to a spin lock.

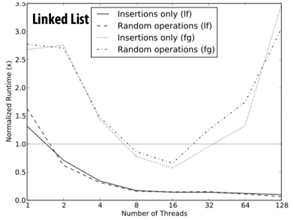
Performance

Lock-free vs. locks performance comparison

Lock-free algorithm run time normalized to run time of using pthread mutex locks







lf = "lock free"
fg = "fine grained lock"

Source: Hunt 2011. Characterizing the Performance and Energy Efficiency of Lock-Free Data Structures

CMU 15-418/618, Spring 2018

https://www.cs.cmu.edu/afs/cs/academic/class/15418-s18/www/lectures/17_lockfree.pdf

External

C++ threading and atomic support:

- `http://en.cppreference.com/w/cpp/thread`
- `http://en.cppreference.com/w/cpp/atomic`

Concurrent data structure:

- in TBB `https://software.intel.com/en-us/node/506169`
- lib lfds `http://liblfds.org/`

Atomics:

- `https://en.wikipedia.org/wiki/Compare-and-swap`
- `https://en.wikipedia.org/wiki/Fetch-and-add`
- `https://en.wikipedia.org/wiki/Test-and-set`

Compiler:

- volatile for embedded programming: `http://www.barrgroup.com/Embedded-Systems/How-To/C-Volatile-Keyword`

Ordering:

- On compiler reordering: `http://preshing.com/20120625/memory-ordering-at-compile-time/`
- On cpu reordering: `http://preshing.com/20120515/memory-reordering-caught-in-the-act/`
- C++ on memory ordering: `http://en.cppreference.com/w/cpp/atomic/memory_order`
- Memory ordering of different architectures: `https://en.wikipedia.org/wiki/Memory_ordering`

Lock-free:

- Andrei Alexandrescu. Lock-Free Data Structures. 2007.
- Keir Fraser, Tim Harris. Concurrent Programming Without Locks. ACM Transactions on Computer Systems, Vol. 25 (2), May 2007
- Chris Purcell and Tim Harris. Non-blocking hashtables with open addressing. 19th International Symposium on Distributed Computing (DISC), September 2005