Lock Free Concurrent Data Structures

K Hari Babu
Department of Computer Science & Information Systems

BITS Pilani
Pilani Campus

Problems with Locking

- Performance issues
  - Locks necessitate waits, context switching, CPU stalls, etc… effecting performance
- Undoing locks
  - If a thread dies while holding a lock, no other thread can proceed
- Can lead to deadlocks
- Priority inversion
  - If a higher priority thread needs to enter critical region but lock is held by low priority thread.

Lock-free Programming

- Thread-safe access to shared data without the use of synchronization primitives such as mutexes
- Possible but not practical in the absence of hardware support

General Approach to Lock-Free Algorithms

- Designing generalized lock-free algorithms is hard
- Design lock-free data structures instead
  - buffer, list, stack, queue, map etc.
- One can't implement lock free algorithms in terms of lock-based data structures.

Lock-free Data Structures

- Is it possible to build data structures that are thread-safe without locks?
  - Yes
- Lock-free data structures
  - Include no locks, but are thread safe
  - may introduce starvation
    - Due to retry loops
- Wait-free data structures
  - Include no locks, are thread safe, and avoid starvation
  - Wait-free implies lock-free. Wait-free is much stronger than lock-free
  - Wait-free structures are very hard to implement
  - Impossible to implement for many data structures
  - Often restricted to a fixed number of threads

Lock-free Data Structures

- Very few standard libraries/APIs implement these data structures
- Implementations are often platform-dependent
  - Rely on low-level assembly instructions
  - Many structures are very new, not widely known
- Lock free data structures make use of the <u>retry loop pattern</u>
  - Read some state
  - Do a useful operation
  - Attempt to modify global state if it hasn't changed (using Compare and Swap (CAS))
- This is similar to a spinlock
  - But, the assumption is that wait times will be small
  - However, retry loops may introduce starvation

Instruction-level Atomicity

- On x86
  - The INC instruction adds 1 to the destination operand. The destination operand can be a register or a memory location.
  - In case of memory address, this instruction loads the value, increments and stores it back. Meanwhile DMA or other cores may access the same memory location, and modify it.
- x86 provides LOCK prefix which ensures that the CPU has exclusive ownership of the appropriate cache line for the duration of the operation and by asserting a bus lock
- The lock prefix makes an instruction atomic and is legal only with some instructions.

Compare and Swap Instruction (CAS)

- On x86, known as compare and exchange

```
spin_lock:
        mov ecx, 1
        mov eax, 0
        lock cmpxchg ecx, [lock_addr]
        jnz spinlock
```

- cmpxchg compares eax and the value of lock_addr
- If eax == [lock_addr], swap ecx and [lock_addr]

The Price of Atomicity

- Atomic operations are very expensive on a multi-core system
  - Caches must be flushed
    - CPU cores may see different values for the same variable if they have out-of-date caches
    - Cache flush can be forced using a memory fence (sometimes called a memory barrier). MFENCE performs a serializing operation on all load-from-memory and store-to-memory instructions that were issued prior to the MFENCE instruction
  - Memory bus must be locked
    - No concurrent reading or writing
  - Other CPUs may stall
    - May block on the memory bus or atomic instructions

Implementation

- Many lock-free data structures can be built using compare and swap (CAS)

```
bool cas(int * addr, int oldval, int newval) {
    if (*addr == oldval) {
    *addr = newval;
    return true;
    }
    return false;
}
```

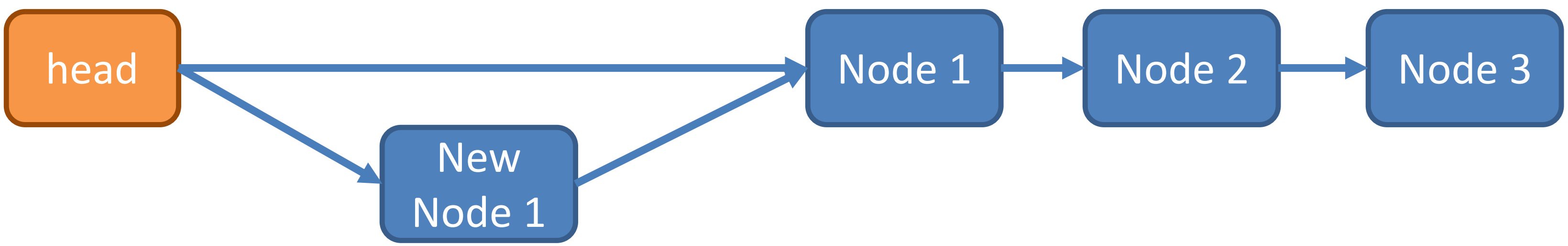
- This can be done atomically on x86 using the **cmpxchg** instruction
- Many compilers have built in atomic swap functions
  - GCC: __sync_bool_compare_and_swap(ptr, oldval, newval)
  - MSVC: InterlockedCompareExchange(ptr,oldval,newval)

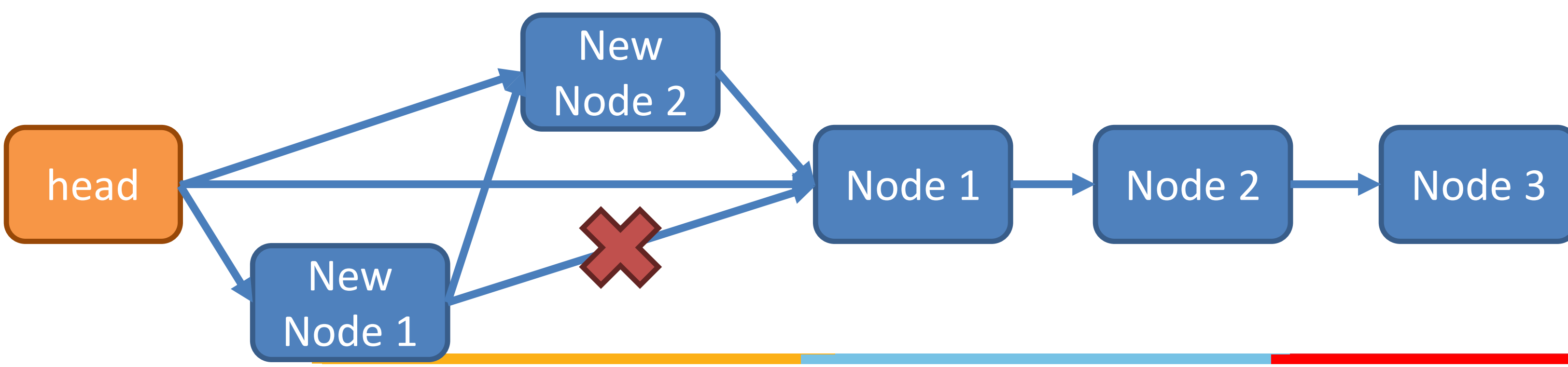
Lock-free Stack Example: Push

```
class Node {
   Node * next;
   int data;
};

// Root of the stack
volatile Node * head;
```

```
void push(int t) {
  Node* node = new Node(t);
  do {
    node->next = head->next;
  } while (!cas(head->next, node->next, node));
}
```

```
bool cas(int * addr, int oldval, int newval) {
      if (*addr == oldval) {
      *addr = newval;
      return true;
```



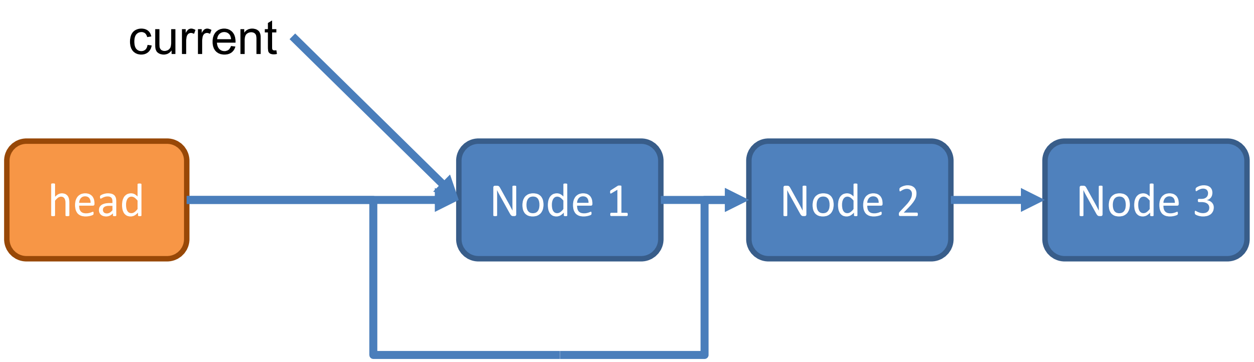


Lock-free Stack Example: Pop

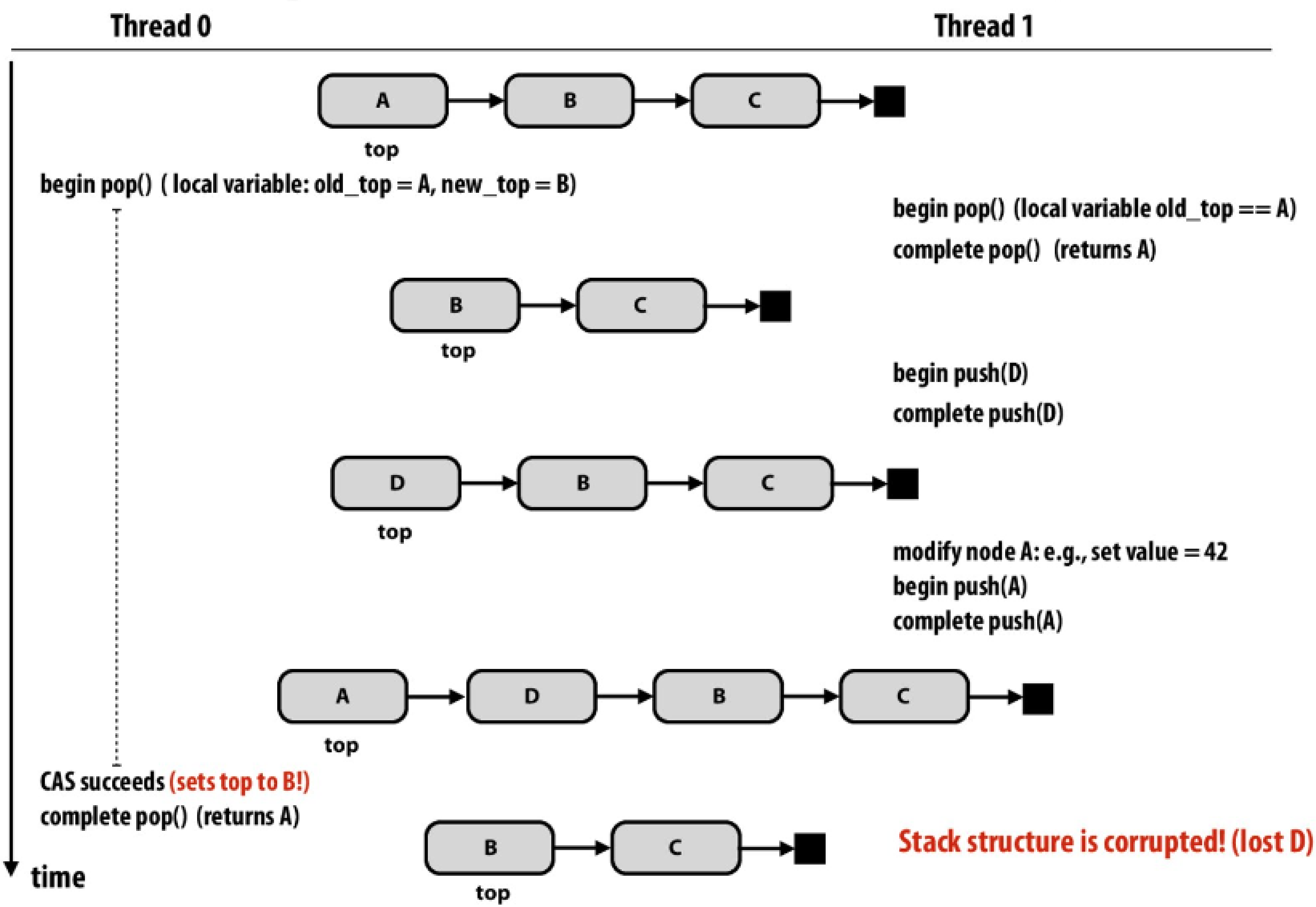
```
class Node {
   Node * next;
   int data;
};

// Root of the stack
volatile Node * head;
```

```
bool pop(int& t) {
   Node* current = head->next;
   while(current) {
      if(cas(head->next, current, current->next)) {
         t = current->data;
         delete current;
         return true;
      }
      current = head->next;
   }
   return false;
}
```

```
bool cas(int * addr, int oldval, int newval) {
      if (*addr == oldval) {
      *addr = newval;
      return true;
```



Lock-free Stack: ABA problem



http://15418.courses.cs.cmu.edu/spring2017/lecture/lockfree/slide_027

ABA Workarounds

- Workarounds
  - Keep a ‘update count’ (needs ‘doubleword CAS’)
    - Kind of version counter that will not match though value matches
  - Don’t recycle the memory ‘too soon’

Applications of Lock-Free Structures

- Stack
- Queue
- Linked list
- Doubly linked list
- Hash table
- Many variations on each
  - Lock free vs. wait free
- Memory managers
  - Lock free malloc() and free()
- The Linux kernel
  - Many key structures are lock-free

References

- Geoff Langdale, Lock-free Programming

http://www.cs.cmu.edu/~410-s05/lectures/L31_LockFree.pdf

Q&A

Thank You