Lock-Free Programming

Geoff Langdale

Desynchronization

- This is an interesting topic
- This will (may?) become even more relevant with near ubiquitous multi-processing
- Still: please don’t rewrite any Project 3s!

Synchronization

- We received notification via the web form that one group has passed the P3/P4 test suite. Congratulations!
- We will be releasing a version of the fork-wait bomb which doesn't make as many assumptions about task id's.
  - Please look for it today and let us know right away if it causes any trouble for you.
- Personal and group disk quotas have been grown in order to reduce the number of people running out over the weekend
  - if you try hard enough you'll still be able to do it.

Outline

- Problems with locking
- Definition of Lock-free programming
- Examples of Lock-free programming
- Linux OS uses of Lock-free data structures
- Miscellanea (higher-level constructs, ‘wait-freedom’)
- Conclusion

Problems with Locking

- **This list is more or less contentious, not equally relevant to all locking situations:**
  - **Deadlock**
  - **Priority Inversion**
  - **Convoying**
  - **“Async-signal-safety”**
  - **Kill-tolerant availability**
  - **Pre-emption tolerance**
  - **Overall performance**

Problems with Locking 2

- Deadlock
  - Processes that cannot proceed because they are waiting for resources that are held by processes that are waiting for…
- Priority inversion
  - Low-priority processes hold a lock required by a higher-priority process
  - Priority inheritance a possible solution

Problems with Locking 3

- Convoying
  - Like the 61-series buses on Forbes Avenue
    - Well, not exactly (overtaking stretches the metaphor?)
  - Several processes need locks in a roughly similar order
  - One slow process gets in first
  - All the other processes slow to the speed of the first one

Problems with Locking 4

- ‘Async-signal safety’
  - Signal handlers can’t use lock-based primitives
  - Especially malloc and free
  - Why?
    - Suppose a thread receives a signal while holding a user-level lock in the memory allocator
    - Signal handler executes, calls malloc, wants the lock
- Kill-tolerance
  - If threads are killed/crash while holding locks, what happens?

Problems with Locking 5

- Pre-emption tolerance
  - What happens if you’re pre-empted holding a lock?
- Overall performance
  - Arguable
  - Efficient lock-based algorithms exist
  - Constant struggle between simplicity and efficiency
  - Example. thread-safe linked list with lots of nodes
    - Lock the whole list for every operation?
    - Reader/writer locks?
    - Allow locking individual elements of the list?

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
- Example: Lamport's "Concurrent Reading and Writing"
  - CACM 20(11), 1977
  - describes a non-blocking buffer
  - limitations on number of concurrent writers
- Practical with hardware support
  - Odd history: lots of user-level music software uses lock-free data structures

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**
- **Often implemented in terms of simpler primitives**
  - **e.g. 'Multi-word Compare and Set' (MCAS, CAS2, CAS*N*)**
  - **Cannot implement lock-free algorithms in terms of lock-based data structures**
  - **What's going to be one of the scarier underlying lock-free, thread-safe primitive?**
    - **Hint: you usually need this for lists and stacks…**

Simple Lock-Free Example

- **Lock-free stack (aka LIFO queue)**
- **With integers! (wow…)**
- **Loosely adapted from example by Jean Gressmann**
  - **Basically ‘uglied up’ (C++ to C)**

Lock-free Stack Structures

```
class Node {
   Node * next;
   int data;
};
// stable ‘head of list’, not an real Node
Node * head;
```

- Not great style, just happens to fit on a slide
- Better to not gratuitously alias ‘whole data structure’ and ‘data structure element’ classes/structures, IMO

Lock-free Stack Push

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

Lock-Free Stack Pop

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

Lock-free Stack: ABA problem

- **‘ABA problem’**
  - **Thread 1 looks at some shared variable, finds that it is ‘A’**
  - **Thread 1 calculates some interesting thing based on the fact that the variable is ‘A’**
  - **Thread 2 executes, changes variable to B**
  - **(if Thread 1 wakes up now and tries to compare-and-set, all is well – compare and set fails and Thread 1 retries)**
  - **Instead, Thread 2 changes variable *back to A!***
  - **OK if the variable is just a value, but…**

Lock-free Stack: ABA problem

- **In our example, variable in question is the stack head**
  - **It’s a pointer, not a plain value!**

```
Thread 1: pop()

read A from head

store A.next `somewhere’

cas with A suceeds
```

```
Thread 2:

pop()

pops A, discards it

First element becomes B

memory manager recycles
‘A’ into new variable

Pop(): pops B

Push(head, A)
```

ABA problem notes

- Work-arounds
  - Keep a ‘update count’ (needs ‘doubleword CAS’)
  - Don’t recycle the memory ‘too soon’
- Theoretically not a problem for LL/SC-based approaches
  - ‘Ideal’ semantics of Load-linked/Store-conditional don’t suffer from this problem
  - No ‘ideal’ implementation of load-linked/store-conditional exists (so all new problems instead of ABA)
    - Spurious failures
    - Limited or no access to other shared variables between LL/SC pairs

Lock-Free Stack Caveats

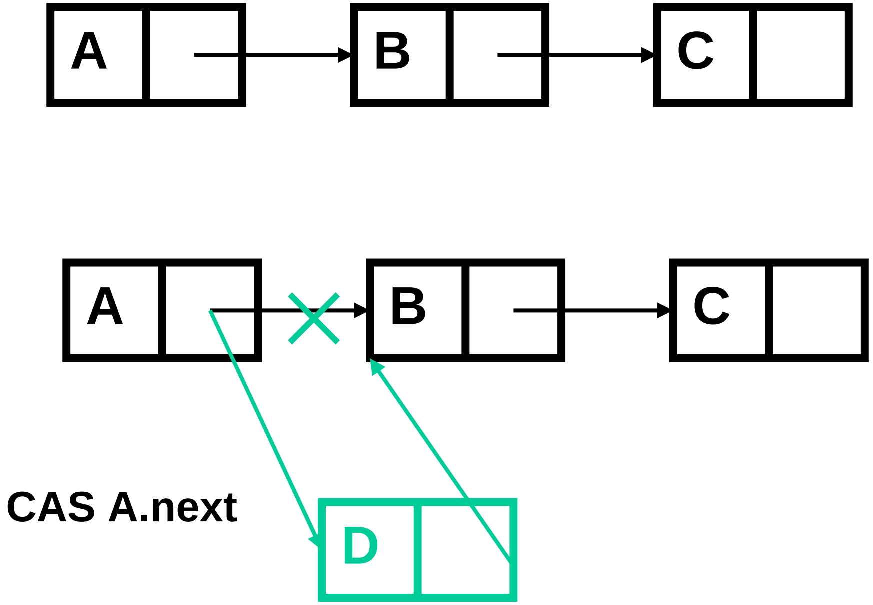
- This is not an especially wonderful example
  - Could implement with a single mutex and expose only push() and pop()
  - Overhead of a single lock is not prohibitive
- Still illustrates some important ideas
  - No overhead
  - Common lock-free technique: atomically switching *pointers*
  - No API possible to ‘hold lock’
  - Illustrates ABA problem

Lock-free Linked Lists

- Better example: lock-free linked lists
- Potentially a long traversal
- Unpleasant to lock list during whole traversal
- High overhead to festoon entire list with locks
- Readers-writers locks only solve part of the problem
  - P2 demonstrated all the difficulties with rwlocks…

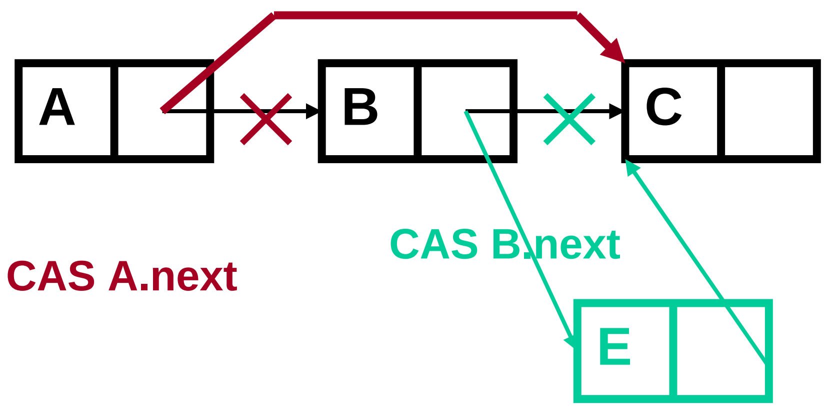
Lock-free Linked Lists

- Example operation: append
- Search for the right spot in the list
- Append using same CAS pointer trick



Lock-free Linked Lists: Deletion

- Problem
  - A thread deleting of B requires an atomic action on node's predecessor
  - Suppose another thread tries to insert E after B (concurrently)
  - B.next -> E
  - B no longer on list, E 'somewhere'



L-F Linked Lists: Deletion Solutions

- **A myriad of solutions, for example:**
- **Harris, "A pragmatic implementation of non-blocking linked-lists", 2001 (15th International Symposium on Distributed Computing)**
  - **Place a 'mark' in the next pointer of the soon-to-be-deleted node**
    - **Easy on aligned architectures (free couple of low-order bits in most pointers)**
  - **Always fail if we try to CAS this (doesn't look like a real pointer)**
  - **If we detect this problem, restart**
    - **Have to go back to the start of the list (we've 'lost our place')**

Lock-free OS Examples

- ACENIC Gigabit Ethernet driver
  - Circular receive buffers with no requirement for spin-lock
- Various schemes proposed for Linux lock-free list traversal
  - "Read-copy-update" (RCU) in 2.5 kernel
  - Yet Another type of Lock-free programming
  - Summary
    - To modify a data structure, put a copy in place
    - Wait until it's known all threads have given up all of the locks that they held (easy in non-preemptive kernel)
    - Then, delete the original
    - Requires memory barriers but no CAS or LL/SC.

Lock-Free Memory Allocation

- **Michael (PLDI 2004), "Scalable Lock-Free Dynamic Memory Allocation"**
- **Thread-safe malloc() and free() with no locks**
- **Claim:**
  - **Near-perfect scalability with added processors under a range of contention levels**
  - **Lower latency than other highly tuned malloc implementations (even with low contention)**

Higher-Level Concepts

- Difficulties with lock-free programming
  - Have to make sure that everyone behaves
    - True of mutexes too; C/C++ can't force you to acquire the right mutex for a given structure
    - Although they can *try*
  - Hard to generalize to arbitrary sets of complex operations
- Object-based Software Transactional Memory
  - Uses object-based programming
  - Uses underlying lock-free data-structures
  - Group operations and commit/fail them atomically
  - Not really a OS-level concept (yet?)

Lock-Free Warnings

- Not a cure for contention
  - It's still possible to have too many threads competing for a lock free data structure
  - Starvation is still a possibility
- Requires the same hardware support as mutexes do
- Not a magic bullet
- Requires:
  - A fairly simple problem (e.g. basic data structure), or
  - Roll your own lock-free algorithm (fun!)

Wait-Freedom

- **Don’t confuse this!**
- **Wait-Free definition: Each operation completes in a finite number of steps**
- **Wait-free implies lock-free**
- **Lock-free algorithms does not imply wait-free**
  - **Note while loops in our lock-free algorithms…**
- **Wait-free synchronization much harder**
  - **Impossible in many cases**
  - **Usually specifiable only given a fixed number of threads**
- **Generally appear only in ‘hard’ real time systems**

Conclusion

- Lock-free programming can produce good performance
- Difficult to get right
  - Performance and correctness (ABA problem)
- Well-established, tested, tuned implementations of common data structures are available
- Good starting points
  - Google: “lock-free programming”
  - http://www.audiomulch.com/~rossb/code/lockfree/ is a good summary