Lock-free programming

Concurrent programs are difficult to develop correctly, particularly for large-scale systems. Problems such as priority inversion, deadlock and convoying have been highlighted.

Lock-free programming became established as a research area from the late 1990s

We'll use a set implemented as a non-blocking linked list as an example, from Tim Harris's paper:
 "A Pragmatic Implementation of Non-Blocking Linked Lists"
 DISC 2001, pp. 300-314, LNCS 2180, Springer 2001

Further reading:

Keir Fraser,
Practical Lock Freedom, 2004.
PhD thesis (UK-Distinguished Dissertation winner), UCAM-CL-TR-579

Keir Fraser and Tim Harris
Concurrent programming without locks
ACM Transactions on Computer Systems (TOCS) 25 (2), 146-196, May 2007

Lock-free programming using CAS

Build on atomic hardware instructions such as Compare-and-Swap (CAS)
Atomic machine (assembler) instructions include:

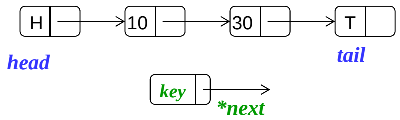
read (addr, register)	// atomic read from memory into register
write (addr, register)	// atomic write to memory from register
CAS (addr, old, new)	// atomic compare-and-swap

CAS (addr, old, new)
as a single atomic instruction: reads value in addr (a memory address)
compares this value with old
updates addr to new iff old == value

Lock-free programming - 1

Example: a set of integers represented as a sorted linked list

set operations: *find (int) -> bool*
insert (int) -> bool
delete (int) -> bool



node.key contains integer value *key*
node.next contains pointer to successor node

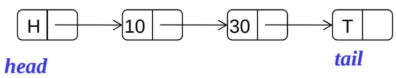
list operations: *read (node.key) -> int*
write (node.key, int)
CAS (node.key, old-int, new-int) -> bool "Compare and Swap"
CAS atomically compares the contents of address *node.key* with the *old-int* value and, iff they match, writes the *new-int* value into *node.key*
CAS returns a boolean to indicate success/failure.

Lock-free programming - 2

Example: a set of integers represented as a sorted linked list

set operations: *find (int) -> bool*

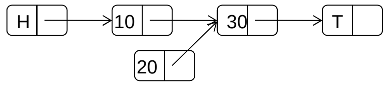
find (20) -> false

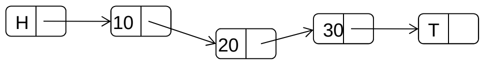


exercise:
write a program to traverse a list,
comparing the integer key in each node with search-key = 20

Lock-free programming 3

Insertion is straightforward. First, the list is traversed until the correct position is found. Then a new cell is created, and inserted atomically using ***CAS*** (compare and swap)



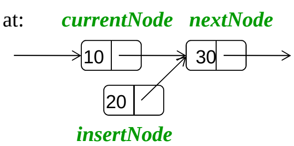


Note that if the ***CAS*** fails, this means that the list has been updated concurrently by other thread(s) and the traversal must start again to find the correct place to insert.
See next slide for more detail.

Lock-free programming 4

boolean = CAS (address, old-value, new-value)

Traverse the list to find where to insert 20, arriving at:



Create ***insertNode*** with ***.next*** pointing to the node with key 30
insertNode.next = * nextNode

done = CAS (currentNode.next, insertNode.next, *insert Node)

If a concurrent insert has been done these will not be equal and ***done*** will be returned false
Restart the traversal .

Consider the correctness of concurrent ***insert*** and ***find***.

Lock-free programming 5

Compare ***CAS*** with the similar "spin lock" approach for claiming a semaphore (locking a mutex).

boolean = CAS (address-in-memory, old-value, new-value)

e.g. ***claimed : boolean = false***
claimed = CAS (flag, 1, 0) with ***flag = 0*** (busy/claimed)
flag = 1 (free/unclaimed)

If the flag indicates the resource is free, atomically set it to busy and return true,
otherwise return false, in which case, repeat the ***CAS*** until it succeeds

Recall a simpler atomic compare and swap RISC instruction :

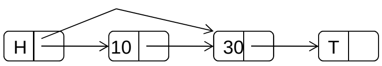
read-and-clear, register flag
if ***register = 0*** repeat ***read-and-clear*** instruction
(spin-lock or busy-wait)
if ***register = 1*** (flag was 1, now = 0) continue into critical region

Here, the ***read-and-clear*** is repeated until it succeeds.

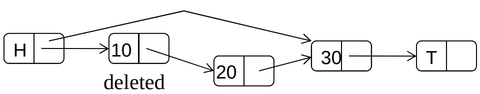
How would you program this ordered list using semaphores? Lock the whole list?

Lock-free programming 6

Correct deletion is more difficult, consider:

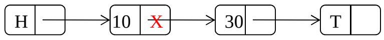


CAS (address, old, new) could be used to change ***node.next*** in the head to point to 30,
after checking the old value points to 10 (so there were no concurrent inserts between H and 10)
But concurrent threads could have inserted values between 10 and 30, after 30 was selected
for the new pointer from H.
Those inserts would be lost:

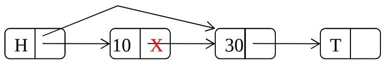


Lock-free programming 7

Correct deletion:



atomically **mark** node for deletion (X)
The node is "*logically deleted*" and this can be detected by concurrent threads that must cooperate to avoid concurrent insertions/deletions at this point
A marked node can still be traversed.



The node is "*physically deleted*"

The algorithms are given in C^{++}-like pseudo-code in the paper, as is a proof of correctness

Exercise:
Consider concurrent executions of any combinations of ***find, insert*** and ***delete***.

Lock-free programming – Transactional memory

The research continued to develop Transactional Memory
hiding the complexity of using CAS under concurrency from the programmer.

Idea: define data structures with atomic operations implemented without locks.
These operations can be called like monitor operations.

This topic is not examinable in 2015.