DATA CONSISTENCY MODELS, LOCKS, AND LOCK-FREE SYNCHRONIZATION

Ivan Jibaja

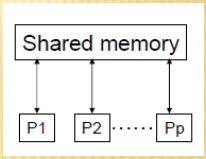
CS 395T – Topics in Multicore Programming

OUTLINE

- Recall: PRAM Model
- New Model
- Architecture Details
- Consistency Models (Relaxed, Release)
- Locks
  - Disadvantages
- Lock-free Synchronization
- Lock-free Data Structures

RECALL: PRAM MODEL

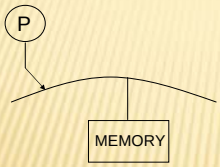
- Parallel Random Access Machine (PRAM)
  - Natural extension of RAM model
- Processors operate synchronously (in lockstep)
- Each processor has private memory
- Are PRAM Models any good?



RECALL: PRAM MODEL

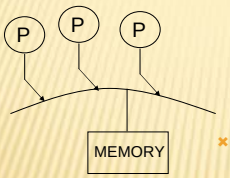
- PRAM Model:
  - No lock-step execution of global instructions
  - No global shared-memory for reads/writes by all of the processors in each cycle
- New Model:
  - Each processor executes its own program at its own speed
  - Only one processor can access memory at a time
  - How does a processor know what other processors are doing? We need **synchronization**
- Let's look at our new model in more detail

MULTITHREADED PROGRAM ON UNIPROCESSOR



- Assumptions for this example:
  - Processor executes all threads of a program:
    - Unspecified scheduling policies
  - Operations in each thread are executed in **order**
  - Atomic operations (lock/unlock) for synchronization between threads
  - Result is as if instructions from different threads were interleaved in some order
  - **Non-determinacy**: program might produce different outputs based on the scheduling of the threads (Can you come up with an example?)

MULTITHREADED ON MULTIPROCESSOR



- Assumptions for this model:
  - Each processors executes one thread
  - Operations in each thread are executed in order
  - One processor at a time can access global memory to perform load/store/atomic operations (no caching of global data)
- With these assumptions, you can show that running a multi-threaded program on a multiprocessor does not change possible output from the uniprocessor case

EXAMPLE (I)

Code:
Initially A = Flag = 0

P1	P2
A = 23;	**while (Flag != 1) {;}**
Flag = 1;	**… = A;**

Semantics:
- P1 writes data into A and sets Flag to tell P2 that data value can be read from A.
- P2 waits till Flag is set and then reads data from A.

EXAMPLE (II)

Code: (similar to Dekker's algorithm)
Initially Flag1 = Flag2 = 0

P1	P2
Flag1 = 1;	Flag2 = 1;
If (Flag2 == 0)	If (Flag1 == 0)
critical section	*critical section*

- What is the problem with our model? We are making unrealistic architectural assumptions.

Note: More complex synchronization problems require hardware support in the form of atomic instructions. We'll discuss this later in this lecture.

ARCHITECTURE DETAILS

- We have some architectural constrains with 2 of our assumptions:

1. Processors do not cache global data:
   - For execution efficiency, processors are allowed to cache global data:
     - Leads to cache coherence problems, which can be solved using coherent caches
2. Instructions within each thread are executed in order
   - For execution efficiency, processors are allowed to execute instructions out of order subject to data/control dependances:
     - Changes the semantics of the program
     - To prevent this requires attention to **memory consistency models**

CURRENT PROCESSORS

	Loads Reordered After Loads?	Loads Reordered After Stores?	Stores Reordered After Stores?	Stores Reordered After Loads?	Atomic Instructions Reordered With Loads?	Atomic Instructions Reordered With Stores?	Dependent Loads Reordered?	Incoherent Instruction Cache Pipeline?
Alpha	Y	Y	Y	Y	Y	Y	Y	Y
AMD64	Y			Y				
IA64	Y	Y	Y	Y	Y	Y		Y
(PA-RISC)	Y	Y	Y	Y				
PA-RISC CPUs								
POWER	Y	Y	Y	Y	Y	Y		Y
SPARC RMO	Y	Y	Y	Y	Y	Y		Y
(SPARC PSO)			Y	Y		Y		Y
SPARC TSO				Y				Y
x86	Y	Y		Y				Y
(x86 OOStore)	Y	Y	Y	Y				Y
zSeries				Y				Y

RECALL: UNIPROCESSOR EXECUTION

- Adding new constrains to our uniprocessor model:
  - Processors reorder operations to improve performance
  - Constraint on reordering: must respect dependences
    - data dependences must be respected: in particular, loads/stores to a given memory address must be executed in program order
    - control dependences must be respected
  - Reorderings can be performed either by the compiler or the processor

PERMITTED MEMORY-OP REORDERINGS

- Stores to different memory locations can be performed out of program order

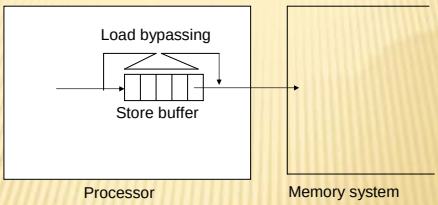
  store v1, data ←→ store b1, flag
  store b1, flag ←→ store v1, data

- Loads from different memory locations can be performed out of program order

  load flag, r1 ←→ load data,r2
  load data, r2 ←→ load flag, r1

- Load and store to different memory locations can be performed out of program order

EXAMPLE OF HARDWARE RE-ORDERING



- Store buffer holds store operations that need to be sent to memory
- Loads are higher priority operations than stores since their results are needed to keep processor busy, so they bypass the store buffer
- Load address is checked against addresses in store buffer, so store buffer satisfies load if there is an address match
- Result: load can bypass stores to other addresses

PROBLEM IN MULTIPROCESSOR CONTEXT

- Our first model (Canonical ordering)
  - Operations from given processor are executed in program order
  - Memory operations from different processors appear to be interleaved in some order at the memory
- Our revisited model:
  - If a processor is allowed to reorder independent operations in its **own** instruction stream, will the execution always produce the same results as the canonical model?
  - Answer: ?

EXAMPLE (I) REVISITED

Code:
Initially A = Flag = 0

P1
A = 23;
Flag = 1;

P2
while (Flag != 1) {;}
... = A;

Semantics:
- P1 writes data into A and sets Flag to tell P2 that data value can be read from A.
- P2 waits till Flag is set and then reads data from A.

EXECUTION SEQUENCE FOR (I)

Code:
Initially A = Flag = 0

P1
A = 23;
Flag = 1;

P2
while (Flag != 1) {;}
... = A;

Possible execution sequence on each processor:

P1	P2	
Write A 23	**Read Flag**	**//get 0**
Write Flag 1	**.....**	
	Read Flag	**//get 1**
	Read A	**//what do you get?**

Problem:
- If the two writes on processor P1 can be reordered, it is possible for processor P2 to read 0 from variable A.
- Can happen on most modern processors

EXAMPLE (II) REVISITED

Code: (similar to Dekker's algorithm)
Initially Flag1 = Flag2 = 0

P1	P2
Flag1 = 1;	**Flag2 = 1;**
If (Flag2 == 0)	**If (Flag1 == 0)**
critical section	***critical section***

Possible execution sequence on each processor:

P1	P2
Write Flag1, 1	**Write Flag2, 1**
Read Flag2 //get 0	**Read Flag1 //what could you get?**

EXECUTION SEQUENCE FOR (II)

Code: (similar to Dekker's algorithm)
Initially Flag1 = Flag2 = 0

P1	P2
Flag1 = 1;	**Flag2 = 1;**
If (Flag2 == 0)	**If (Flag1 == 0)**
critical section	***critical section***

Possible execution sequence on each processor:

P1	P2
Write Flag1, 1	**Write Flag2, 1**
Read Flag2 //get 0 →	**Read Flag1 //what could you get?**

- Most people would say that P2 will read 1 as the value of Flag1. Since P1 reads 0 as the value of Flag2, P1's read of Flag2 must happen before P2 writes to Flag2. Intuitively, we would expect P1's write of Flag to happen before P2's read of Flag1.
- However, this is true only if reads and writes on the same processor to different locations are not reordered by the compiler or the hardware. Unfortunately, this is very common on most processors (store-buffers with load-bypassing).

SUMMARIZING THE PROBLEM

- Uniprocessors can reorder instructions subject only to control and data dependence constraints
- These constraints are not sufficient in shared-memory context
  - simple parallel programs may produce counter-intuitive results
- Question: what constraints must we put on uniprocessor instruction reordering so that:
  - shared-memory programming is intuitive?
  - but we do not lose uniprocessor performance?
- Many answers to this question:
  - answer is called **memory consistency model** supported by the processor

CONSISTENCY MODELS

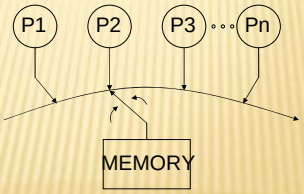
- Consistency models are **not** about memory operations **from different processors** .
- Consistency models are **not** about **dependent memory operations in a single processor's** instruction stream (these are respected even by processors that reorder instructions).

CONSISTENCY MODELS

- Consistency models are all about ordering constraints on **independent memory operations in a single processor's** instruction that should be respected to obtain intuitively reasonable results.

SIMPLEST MEMORY CONSISTENCY MODEL

- Sequential consistency (SC) [Lamport]
  - Our canonical model: processor is not allowed to reorder reads and writes to global memory



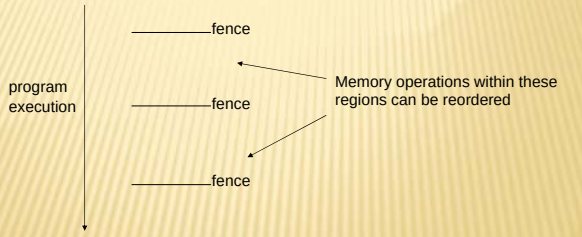
SEQUENTIAL CONSISTENCY

- SC constrains all memory operations:
  - Write → Read
  - Write → Write
  - Read → Read, Write
- Simple model for reasoning about parallel programs
  - You can verify that the examples considered earlier work correctly under sequential consistency.
- However, this simplicity comes at the cost of uniprocessor performance.
- Question: how do we reconcile sequential consistency model with the demands of performance?

RELAXED MODEL: WEAK CONSISTENCY

- Programmer specifies regions within which global memory operations can be reordered
- Processor has fence instruction:
  - all data operations before fence in program order must complete before fence is executed
  - all data operations after fence in program order must wait for fence to complete
  - fences are performed in program order
- Implementation of fence:
  - processor has counter that is incremented when data op is issued, and decremented when data op is completed
- Example: PowerPC has SYNC instruction
- Language constructs:
  - OpenMP: flush
  - All synchronization operations like lock and unlock act like a fenc

WEAK CONSISTENCY PICTURE



EXAMPLE (I) REVISITED

Code:
Initially A = Flag = 0

P1
A = 23;
flush;
Flag = 1;

P2
while (Flag != 1) {;}
flush;
... = A;

Execution:
- P1 writes data into A
- Flush waits till write to A is completed
- P1 then writes data to Flag
- Therefore, if P2 sees Flag = 1, it is guaranteed that it will read the correct value of A even if memory operations in P1 before flush and memory operations after flush are reordered by the hardware or compiler.
- Does P2 need a flush between the two statements?

RELAXED MODEL: RELEASE CONSISTENCY

- Further relaxation of weak consistency
- Synchronization accesses are divided into
  - Acquires: operations like lock
  - Release: operations like unlock
- Semantics of acquire:
  - Acquire must complete before all following memory accesses
- Semantics of release:
  - All memory operations (read/write) before release are complete
- However,
  - Acquire does not wait for accesses preceding it
  - Accesses after release in program order do not have to wait for release
    - Operations which follow release and which need to wait must be protected by an acquire

COMMENTS

- There are a lot of other consistency models out there:
  - Causal consistency
  - Processor consistency
  - Delta consistency….
- It is important to remember that these are concerned with reordering of independent memory operations **within** a processor.
- Easy to come up with shared-memory programs that behave differently for each consistency model.

CONSISTENCY VS. COHERENCE

MEMORY CONSISTENCY

- What instructions is compiler or hardware allowed to reorder?
- Nothing really to do with memory operations from different processors/threads
- Sequential consistency: perform global memory operations in program order
- Relaxed consistency models: all of them rely on some notion of a fence operation that demarcates regions within which reordering is permissible

MEMORY COHERENCE

- Preserve the illusion that there is a single logical memory location corresponding to each program variable even though there may be lots of physical memory locations where the variable is stored

OTHER TOOLS FOR SYNCHRONIZATION

LOCKS

- In a Uniprocessor, locking was achieved by disabling interrupts for the smallest possible number of instructions that will access shared data ("the critical section"):

```
while (true) {
        /* disable interrupts */
        /* critical section */
        /* enable interrupts */
        /* remainder */
}
```

- This approach does not work with multiprocessors. Why?

LOCK IMPLEMENTATION IN HARDWARE

- Test and Set, here on, TS:
  - TS on a boolean variable flag
    - #atomic // The two lines below will be executed one after the other without interruption
    - If(flag == false)
    - flag = true;
    - #end atomic

```
bool lock = false; // shared lock variable
// Process i
Init i;
while(true) {
        while (lock==false){ // entry protocol
                TS(lock)};
Critical secion # i;
lock = false; // exit protocol
//Remainder of code;}
```

DISADVANTAGES OF LOCKS

- Notice the while loop in the algorithm
- If process 0 waits a lot of time to enter the critical section, it continually checks the flag and turn to see it can or not, while **not doing any useful work**
- This is termed **busy waiting**, and locking mechanisms have a major disadvantage in that regard.
- Locks that employ continuous checking mechanism for a flag are called **Spin-Locks**.
- Spin locks are good when the you know that the wait is not long enough.

EXAMPLE: A LOCK BASED STACK

- Stack: A list or an array based data structure that enforces last-in-first-out ordering of elements
- Operations
  - Void Push(T data) : pushes the variable data on to the stack
  - T Pop() : removes the last item that was pushed on to a stack. Throws a stackEmptyException if the stack is empty
  - Int Size() : returns the size of the stack
- All operations are synchronized using one common lock object.

CODE : JAVA

```
Class Stack<T> {

ArrayList<T> _container = new ArrayList<T>();
RentrantLock _lock = new ReentrantLock();

public void push(T data){ _lock.lock(); _container.add(data); _lock.unlock();}

public int size(){
        int retVal; _lock.Lock(); retVal = _container.size();
        _lock.unlock();
        return retVal;
}

public T pop(){
        _lock.lock();
        if(_container.empty()) {
          _lock.unlock();
                throw new Exception("Stack Empty");}
        T retVal _container.get(_container.size() – 1);
        _lock.unlock(); return retVal;
}
```


PROBLEMS WITH LOCKS

- Stack is simple enough. There is only one lock. The overhead isn't that much. But there are data structures that could have multiple locks
- Problems with locking
  - Deadlock
  - Priority inversion
  - Convoying
  - Preemption tolerance
  - Overall performance

PROBLEMS WITH LOCKING 2

- Priority inversion:
  - Assume two threads:
    - T2 with very low priority
    - T1 with very high priority
  - Both need to access a shared resource R but T2 holds the lock to R
    - T2 takes longer to complete the operation leaving the higher priority thread waiting, hence by extension T1 has achieved a lower priority

PROBLEMS WITH LOCKING 3

- **Deadlock:** Processes can't proceed because each of them is waiting for the other release a needed resource.
- Scenario:
  - There are two locks A and B
  - Process 1 needs A and B in that order to safely execute
  - Process 2 needs B and A in that order to safely execute
  - Process 1 acquires A and Process two acquires B
  - Now Process 1 is waiting for Process 2 to release B and Process 2 is waiting for process 1 to release A

PROBLEMS WITH LOCKING 4

- **Convoying**, all the processes need a lock **A** to proceed however, a lower priority process acquires **A** it first. Then all the other processes slow down to the speed of the lower priority process.
- Think of a freeway:
  - You are driving an Aston Martin but you are stuck behind a beat up old pick truck that is moving very slow and there is no way to overtake him.

PROBLEMS WITH LOCKING 5

- 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 SYNCHRONIZATION ("NON-BLOCKING")

LOCK-FREE DATA STRUCTURES

- A data structure wherein there are no explicit locks used for achieving synchronization between multiple threads, and the progress of one thread doesn't block/impede the progress of another.
- Doesn't imply **starvation freedom (*Meaning one thread could potentially wait forever).*** *But nobody starves in practice*
- **Advantages:**
  - You don't run into all the problems that you would with using locks
- **Disadvantages:** To be discussed later

LOCK-FREE SYNCHRONIZATION

- Think in terms of **Algorithms + Data Structure = Program**
- Thread safe access to shared data without the use of locks, mutexes etc.
- Possible but not practical/feasible in the absence of hardware support
- So what do we need?
  - Design algorithm to avoid critical sections
  - A compare and set primitive (**CAS)** from the hardware

LOCK-FREE SYNCHRONIZATION

- Compare and Set primitive
  - boolean cas(int * valueToChange, int * valueToSet To, int * ValueToCompareTo)
  - Sematics: The pseudocode below executes atomically without interruption

```
If( valueToChange == valueToCompareTo){
      valueToChange = valueToSetTo;
      return true;
   }
   else {
      return false;
   }
```

  - This function is exposed in Java through the atomic namespace, in C++ depending on the OS and architecture, you find libraries
- **CAS** is all you need for lock-free queues, stacks, linked-lists, and sets.

BUILDING LOCK-FREE DATA STRUCTURES

- What do we need to build these lock-free data structures?
- Limit the scope of changes to a single atomic variable
  - Stack : head
  - Queue: head or tail depending on enque or deque

A SIMPLE LOCK-FREE EXAMPLE

- A lock-free Stack
- Adopted from **Geoff Langdale** at CMU
  - Intended to illustrate the design of lock-free data structures and problems with lock-free synchronization
  - There is a primitive operation we need:
    - Compare and Set (CAS)
    - Available on most modern machines
      - X86 assembly: xchg
      - PowerPC assembly: LL(load linked), SC (Store Conditional)

LOCK-FREE STACK WITH INTS IN C

- A stack based on a singly linked list. Not particularly good design!

```
struct NodeEle {
    int data;
    Node *next;
};

typedef NodeEle Node;

Node* head; // The head of the list
```

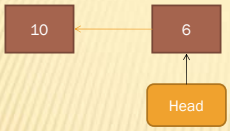
- Now that we have the nodes let us proceed to body of the stack

LOCK-FREE STACK PUSH

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

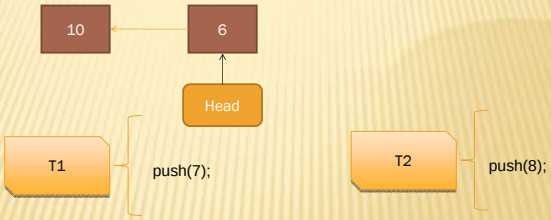
Let us see how this works!

PUSH IN ACTION



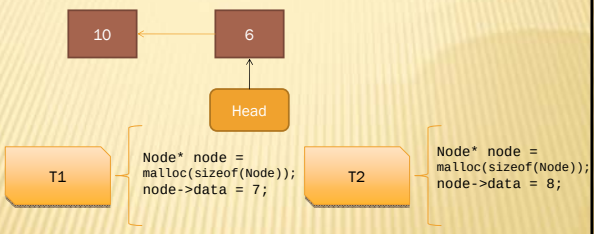
- Currently Head points to the Node containing data 6

PUSH IN ACTION



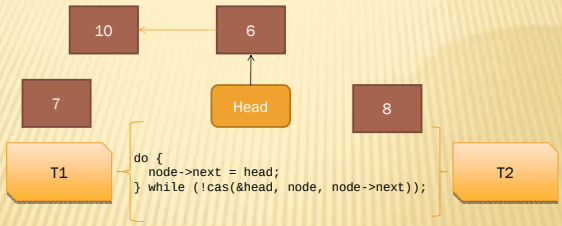
- Two threads T1 and T2 comes along wanting to push 7 and 8 respectively, by calling the push function

PUSH IN ACTION



- Two new node structs on the heap will be created on the heap in parallel after the execution of the code shown

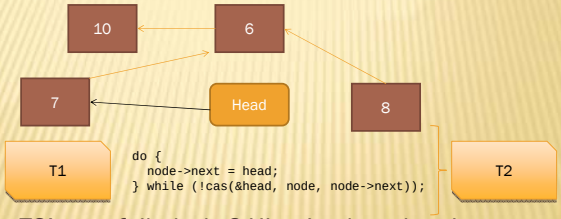
PUSH IN ACTION



- The above code means set the newly created Nodes next to head, if the head is still points to 6 then change head pointer to point to the new Node
- Both of them try to execute this portion of the code on their respective threads. But only one will succeed.

PUSH IN ACTION

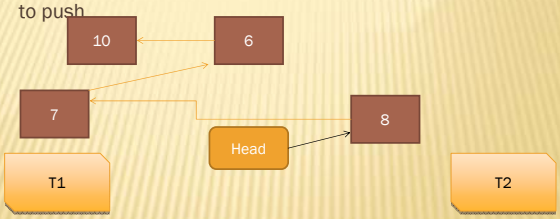
- Let us Assume T1 Succeeds, therefore T1 exits out of the while and consequently the push()



- T2's cas failed why? Hint: Look at the picture.
  - T2 has no choice but to try again

PUSH IN ACTION

- Assume T2 Succeeds this time because no one else trying to push



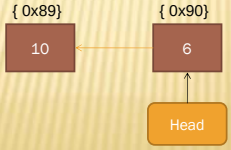
POP()

```
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;
}
```

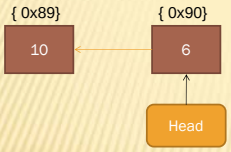
There is something wrong this code. It is very subtle. Can you figure it out? Most of the time this piece of code will work.

IT IS CALLED THE ABA PROBLEM

- While a thread tries to modify A, what happens if A gets changed to B then back to A?
- Malloc recycles addresses. It has to eventually.
- Now Imagine this scenario.
- Curly braces contain addresses for each node



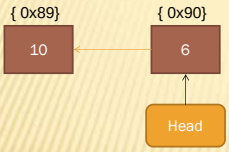
ABA PROBLEM ILLUSTRATION STEP 1



- Assume two threads T1 and T2.
- T1 calls pop() to delete Node at 0x90 but before it has a change and CAS, there is a context switch and T1 goes to sleep.

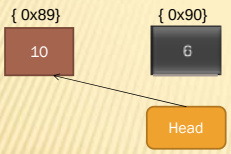
```
bool pop(int& t) {
        Node* current = del = head;
        while(current) {
        if(cas(&head, current->next, current)) {
                t = current->data; // problem?
                delete del;
                return true;
        }
        current = head;
        }
return false;
}
```


ABA PROBLEM ILLUSTRATION STEP 2



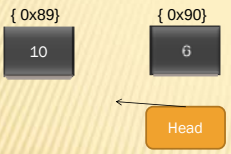
- The following happens while T1 is asleep

ABA PROBLEM ILLUSTRATION STEP 3



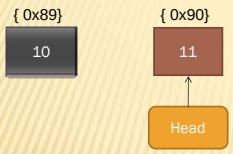
- The following happens while T1 is asleep
  - T2 calls Pop(), Node at 0x90 is deleted

ABA PROBLEM ILLUSTRATION STEP 4



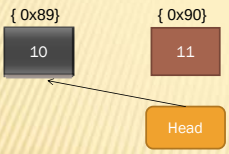
- The following happens while T1 is asleep
  - T2 calls Pop(), Node at 0x90 is deleted
  - T2 calls Pop(), Node at 0x89 is deleted

ABA PROBLEM ILLUSTRATION STEP 5



- The following happens while T1 is asleep
  - T2 calls Pop(), Node at 0x90 is deleted
  - T2 calls Pop(), Node at 0x89 is deleted
  - T2 calls push(11) but malloc has recycled the memory 0x90 while allocating space for the new Node

ABA PROBLEM ILLUSTRATION STEP 6



Head is now pointing to illegal memory!!!!

Replace **10** and **6** with **B** and **A**

Now you know where the name (ABA) comes from

- The following happens while T1 is asleep
  - T2 calls Pop(), Node at 0x90 is deleted
  - T2 calls Pop(), Node at 0x89 is deleted
  - T2 calls push(11) but malloc has recycled the memory 0x90 while allocating space for the new Node
  - **T1 now wakes up** and the CAS operation succeeds

SOLUTIONS:

- **Double word compare and set.**
  - One 32 bit word for the address
  - One 32 bit word for the update count which is incremented every time a node is updated
  - Compare and Set **iff both of the above match**
  - Java provides **AtomicStampedReference**
- **Use the lower address bits of the pointer** (if the memory is 4/8 byte Aligned) to keep a counter to update
  - But the probability of a false positive is still greater than doubleword compareandset because instead of 2^32 choices for the counter you have 2^2 or 2^3 choices for the counter

DISADV. OF LOCK-FREE DATA STRUCTURES

- Current hardware limits the amount of bits available in CAS operation to 32/64 bits.
- Imagine the implementation of data structures like BST's pose a problem
  - When you need to balance a tree you need update several nodes all at once.
- Way to get around it
  - Transactional memory based systems

CONCLUSIONS