WIKIPEDIA

ABA problem

In multithreaded computing, the **ABA problem** occurs during synchronization, when a location is read twice, has the same value for both reads, and "value is the same" is used to indicate "nothing has changed". However, another thread can execute between the two reads and change the value, do other work, then change the value back, thus fooling the first thread into thinking "nothing has changed" even though the second thread did work that violates that assumption.

The ABA problem occurs when multiple threads (or processes) accessing shared data interleave. Below is the sequence of events that will result in the ABA problem:

- Process P_1 reads value A from shared memory,
- P_1 is preempted, allowing process P_2 to run,
- P_2 modifies the shared memory value A to value B and back to A before preemption,
- P_1 begins execution again, sees that the shared memory value has not changed and continues.

Although P_1 can continue executing, it is possible that the behavior will not be correct due to the "hidden" modification in shared memory.

A common case of the ABA problem is encountered when implementing a lock-free data structure. If an item is removed from the list, deleted, and then a new item is allocated and added to the list, it is common for the allocated object to be at the same location as the deleted object due to optimization. A pointer to the new item is thus sometimes equal to a pointer to the old item which is an ABA problem.

Contents

Examples

Consider a software example of ABA using a lock-free stack:

```
/* Naive lock-free stack which suffers from ABA problem.*/
class Stack {
  std::atomic<Obj*> top_ptr;
  //
  // Pops the top object and returns a pointer to it.
  //
  Obj* Pop() {
    while(1) {
      Obj* ret_ptr = top_ptr;
      if (!ret_ptr) return nullptr;
      // For simplicity, suppose that we can ensure that this dereference is safe
      // (i.e., that no other thread has popped the stack in the meantime).
      Obj* next_ptr = ret_ptr->next;
      // If the top node is still ret, then assume no one has changed the stack.
      // (That statement is not always true because of the ABA problem)
      // Atomically replace top with next.
      if (top_ptr.compare_exchange_weak(ret_ptr, next_ptr)) {
        return ret_ptr;
      }
      // The stack has changed, start over.
    }
  }
  //
  // Pushes the object specified by obj_ptr to stack.
  //
  void Push(Obj* obj_ptr) {
    while(1) {
      Obj* next_ptr = top_ptr;
      obj_ptr->next = next_ptr;
      // If the top node is still next, then assume no one has changed the stack.
      // (That statement is not always true because of the ABA problem)
      // Atomically replace top with obj.
      if (top_ptr.compare_exchange_weak(next_ptr, obj_ptr)) {
        return;
      }
      // The stack has changed, start over.
    }
  }
};
```

This code can normally prevent problems from concurrent access, but suffers from ABA problems. Consider the following sequence:

Stack initially contains *top* → A → B → C

Thread 1 starts running pop:

```
ret = A;
next = B;
```

Thread 1 gets interrupted just before the `compare_exchange_weak`...

```
{ // Thread 2 runs pop:
  ret = A;
  next = B;
  compare_exchange_weak(A, B)  // Success, top = B
  return A;
} // Now the stack is top → B → C
{ // Thread 2 runs pop again:
  ret = B;
  next = C;
  compare_exchange_weak(B, C)  // Success, top = C
  return B;
} // Now the stack is top → C
delete B;
{ // Thread 2 now pushes A back onto the stack:
  A->next = C;
  compare_exchange_weak(C, A)  // Success, top = A
}
```

Now the stack is *top* → A → C

When Thread 1 resumes:

```
compare_exchange_weak(A, B)
```

This instruction succeeds because it finds *top* == ret (both are A), so it sets top to next (which is B). As B has been deleted the program will access freed memory when it tries to look at the first element on the stack. In C++, as shown here, accessing freed memory is undefined behavior: this may result in crashes, data corruption or even just silently appear to work correctly. ABA bugs such as this can be difficult to debug.

Workarounds

Tagged state reference

A common workaround is to add extra "tag" or "stamp" bits to the quantity being considered. For example, an algorithm using compare and swap on a pointer might use the low bits of the address to indicate how many times the pointer has been successfully modified. Because of this, the next compare-and-swap will fail, even if the addresses are the same, because the tag bits will not match. This is sometimes called ABA′ since the second A is made slightly different from the first. Such tagged state references are also used in transactional memory.

If "tag" field wraps around, guarantees against ABA do not stand anymore. However, it has been observed that on currently existing CPUs, and using 60-bit tags, no wraparound is possible as long as the program lifetime (that is, without restarting the program) is limited to 10 years; in addition, it was argued that for practical purposes it is usually sufficient to have 40-48 bits of tag to guarantee against wrapping around. As modern CPUs (in particular, all modern x64 CPUs) tend to support 128-bit CAS operations, in certain use cases it allows to provide firm guarantees against ABA when using tagging. [1]

Intermediate nodes

A correct but expensive approach is to use intermediate nodes that are not data elements and thus assure invariants as elements are inserted and removed [Valois].

Deferred reclamation

Another approach is to defer reclamation of removed data elements. One way to defer reclamation is to run the algorithm in an environment featuring an automatic garbage collector; a problem here however is that if the GC is not lock-free, then the overall system is not lock-free, even though the data structure itself is.

Another way to defer reclamation is to use one or more hazard pointers, which are pointers to locations that otherwise cannot appear in the list. Each hazard pointer represents an intermediate state of an in-progress change; the presence of the pointer assures further synchronization [Doug Lea]. Hazard pointers are lock-free, but can only track at most two elements for being in-use.

Yet another way to defer reclamation is to use read-copy update (RCU), which involves enclosing the update in an RCU read-side critical section and then waiting for an RCU grace period before freeing any removed data elements. Using RCU in this way guarantees that any data element removed cannot reappear until all currently executing operations have completed. RCU is lock-free, but isn't suitable for all workloads.

Some architectures provide "larger" atomic operations such that, as example, both forward and backward links in a doubly linked list can be updated atomically; while this feature is architecture-dependent, it, in particular, is available for x86/x64 architectures (x86 allows for 64-bit CAS, and all modern x64 CPUs allow for 128-bit CAS) and IBM's z/Architecture (which allows for up to 128-bit CAS).

Some architectures provide a load linked, store conditional instruction in which the store is performed only when there are no other stores of the indicated location. This effectively separates the notion of "storage contains value" from "storage has been changed". Examples include DEC Alpha, MIPS, PowerPC and ARM (v6 and later).

See also

- Readers–writers problem

References

- Dechev, Damian; Pirkelbauer, Peter; Stroustrup, Bjarne. "Lock-free Dynamically Resizable Arrays". CiteSeerX 10.1.1.86.2680 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.86.2680).

- Dechev, Damian; Pirkelbauer, Peter; Stroustrup, Bjarne. "Understanding and Effectively Preventing the ABA Problem in Descriptor-based Lock-free Designs" (http://www.stroustrup.com/isorc2010.pdf) (PDF).

1. 'No Bugs' Hare, CAS (Re)Actor for Non-Blocking Multithreaded Primitives (http://ithare.com/cas-reactor-for-non-blocking-multithreaded-primitives/), Overload #142

Retrieved from "https://en.wikipedia.org/w/index.php?title=ABA_problem&oldid=885039905"