An introduction to Wait-Free and Lock-Free data structures

Eric Shenck
Eric.Shenck@dna.lth.se
Mail from the smpdev mailing list

June 25, 2001

David has mentioned the possibility of using lock-free and wait-free algorithms in the Linux SMP code. Since not everyone is familiar with these concepts, this note is an attempt at an introduction to the ideas.

Please bear with me, this note is rather long, and in some parts the writing is a bit rough, but I wanted to get at least a preliminary document into circulation so we have a starting point for discussion. I will update this document as discussion evolves to clarify points that are confusing and to include observations that I have overlooked. Note that this document is by no means complete. One particularly glaring omission at the moment is the lack of bibliographic references. This will be fixed as soon as I have a chance to backtrack on all the trails I have been following while preparing this.

A word on the unintentional bias you will find in this note. In real life I do computational complexity research, mostly centered in parallel or distributed settings. Portions of this note are loosely based on my lecture notes for a graduate course in theoretical distributed that I am teaching this semester. This means that my explanations may tend toward the mathematical, just because this is the language I am most comfortable with. It also means that I may use terms you are familiar with to mean things completely different than what you think. I know this can be annoying, but I can't do much about the choice of terms in the existing literature, and I think it will be better if we are familiar with the terms being used in the literature.

Now, also, a warning. Wait-free algorithms can be quite expensive, and although there is reason to think lock-free algorithms can compete with locking algorithms in some cases, this is not always so. Remember, when we are thinking about applying this stuff to SMP in Linux, the holy grail is not wait-free or lock-free data structures, the holy grail is _speed_.

Ok now, lets roll on a little light theory.

1 Distributed Computing Versus Parallel Computing

The literature on concurrent algorithms is generally divided into two major camps, which don't seem to interact all that much. These camps are the "distributed computing" camp and the "parallel computing" camp. Of course, within these two camps there is the usual division between theoreticians and practitioners, with neither side paying much attention to the other.

Distributed computing tends to be concerned with more loosely coupled computing, and parallel computing tends toward the type of programming where you have a large collection of processors all attempting to work together to solve the same task. Generally we also find that distributed computing is very concerned with fault tolerance, and parallel computing work tends to assume that all participating processors will get their tasks down in a timely manner and without fail.

In this note I will mostly be concerned with discussing results from the distributed computing end of things, although I will introduce some practical tricks that strictly speaking violate the rules of the game that we play when discussing the theory of these things.

2 Basic Model

A shared memory distributed computation is defined over a collection of processes which communicate via some collection of atomic shared objects.

A "shared object" should be thought of as a hardware black box. The black box can perform some set of operations at the request of processors (read, write, etc...) and it guarantees that these operations will be atomic. That is, every operation on an object can be seen as occurring at some distinct instant of time between the time the operation was requested and the time the object responded with the result of the operation. Note that there are be a delay between the time an operation is requested and the time it is seen to have occurred, and there can also be a delay between the time an operation occurs and a processor obtains the result of this operation.

An important practical point that has to been noted here is that the assumption that objects behave atomically is rather strong. In fact, it is generally stronger than what any real machine will offer. The best we can hope for in practice is "sequential consistency", although this is generally good enough for what we want. I'll have a bit more to say about this issue later.

The most basic type of shared object is the "register". This is just a shared memory cell that supports reads and writes. More complex objects include the "test and set" object, and the "compare and swap" object.

Another point that should be noted here is that the theoretical model of having a collection of shared objects each of which supports some private set of operations does not exactly match the design of real machines, where we have a collection of memory cells, all of which support some set of operations. For the most part this distinction should not be important, but it should be kept in mind when reading the literature.

3 Memory consistency conditions

The technical name given the consistency condition guaranteed by fully atomic objects is "linearizability". As far as I know, no machines currently in use actually guarantee linearizability. Sequential consistency is usually the best we can hope for on a real machine, and often we won't even get that.

For our purposes sequential consistency and atomicity are identical, since sequentially consistent shared objects can only be distinguished from atomic shared objects by an outside observer capable of noting the exact times of all events on all processes. For the curious, sequential consistency allows the result of an operation to be obtained from the "past" or the "future" relative to the times at which its requests are made. This is disallowed in truly atomic systems. In practice it can be useful to be allowed to shift operations back in time when implementing consistency hardware, as far as I know forward time anomalies can't happen on real hardware.

The algorithms I describe work on truly atomic systems, and should work on systems that offer sequential consistency (there are some subtle possible problems, but they are unlikely). [READ AS: I know there is something fishy about the difference, but I can't remember exactly what it is, and I can't pin it down right now. Obviously I have to come back to this.]

The algorithms I describe may not work on systems that do not offer sequential consistency. Careful analysis will be necessary to be sure one way or the other. From the documentation I have been able to find, this will be a problem on everything but the x86 and SPARC architectures, since nothing else seems to offer memory consistency without synchronizing barriers, and we seem to have different flavors of synchronizing barriers across different machines. MIPS seems to be the worst of the lot, but AXP and PPC processors aren't much friendlier to SMP programmers. I'll be posting a separate message to start a thread about this issue.

4 Lock-Freedom and Wait-Freedom

If accesses to memory are atomic, then each access to shared memory by a process can be viewed as occurring at a unique instant in time. The order in which accesses to memory occur is called the *schedule*.

Formally, an algorithm for accessing a data structure is called *non-blocking* if under every infinite schedule an infinite number of operations on the data structure are completed. Less formally, this means that no process can block the application of further operations to the data structure, although some process can fail to apply any operations to the data structure because it always gets beat out by a faster process.

An algorithm is called *wait-free* if every process is guaranteed to complete any operation in a finite number of its own steps, regardless of the schedule. Observe that this gives a very strong fault tolerance, since any number of other processes can stop at arbitrary points in their executions without stopping the execution of other processes. (Stronger notions of fault tolerance exist, but they are harder to deal with, and beyond the scope of this discussion.)

5 Important primitives

5.1 Compare and swap

The compare and swap primitive (CAS) atomically checks the contents of a memory location and swaps it with a replacement value if it matches. In either case the instruction returns a code indicating success or failure. In pseudo C, it performs the following atomic action:

```
CAS(*X,old,new) {
        if (*X == old) {
                *X = new;
                return 1;
        } else return 0;
}
```

The CAS instruction or an equivalent instruction is necessary to construct lock-free and wait-free data structures for any more than two processes (I'm simplifying a bit, but this is good enough for what we want to talk about here.) All the architectures we are dealing with either have a CAS or the equivalent Load Linked / Store Conditional pair.

5.2 Double Compare and Swap

The double compare and swap (DCAS) is only available on the MC680x0 series processors as far as I know. It is, however, a very powerful primitive, which can greatly simplify the construction of lock-free and wait-free data structures. In words, the DCAS instruction atomically compares two separate words against test values, and if both words match the test values, they are replaces with new values. In pseudo C, it the following code is performed atomically:

```
DCAS(*X1,old1,new1,*X2,old2,new2) {
        if (*X1 == old1 && *X2 == old2) {
                *X1 = new1;
                *X2 = new2;
                return 1;
        } else return 0;
}
```

5.3 Load Linked / Store Conditionnal

Load Linked / Store Conditional is actually a pair of instructions, which together implement an atomic Read/Modify/Write cycle. The idea is that the Load Linked (LL) instruction marks a memory location as "reserved" (but does not lock it!) and if no processor has tried to change the contents of that memory location when the store conditional takes place, the store will succeed, otherwise it will fail. This pair of instructions can be used to implement Compare and Swap in an obvious way:

```
CAS(*X,old,new)
        tmp = LL(X);
        if (old == tmp) return SC(X,new);
        else return;
```

Most modern processors seem to be offering LL/SC as the only atomic operations. This is a bit of a shame, as a wider range of operations support the efficient construction of lock-free and wait-free data structures better.

5.4 Fetch and Add

Fetch and Add is an old standby in the atomic pile. Many new processors don't bother implementing it, since fetch and add isn't sufficiently general to wait-free synchronize more than two processors, but it's a nice instruction to have anyway, and if you don't have it you usually end up building it out of something else with a major efficiency hit. As you might imagine, it atomically reads a memory location, adds a value to it, and writes the result back to memory. This instruction is particularly nice for building wait-free bounded size queues and stacks.

5.5 Test and Set (TAS)

Another old standby, kind of a limited version of CAS. This primitive is also limited to wait-free synchronizing at most two processes, and has gone out of favor because of this. You can always build this one out of CAS or LL/SC without any trouble, so if it's missing it's not a big deal. It mostly gets used to construct locking mechanisms.

6 Basic lock free techniques

The simplest technique for constructing lock-free data structures is to use CAS or LL/SC to grab the contents of a memory word (which happens to represent the entire state of the data structure) and make the appropriate changes to it, then write it back if the state hasn't changed. As a simple example, consider the following implementation of a fetch and add instruction using CAS:

```
fetchnadd(int *X, int x)
{
    do {
        tmp = *X;
    } while (!CAS(X,tmp,tmp+x));
    return tmp+x;
}
```

This implementation is lock-free, since some processor will always be able to increment the counter. Of course, a processor might always loose to some faster processor, but this is often unlikely in practice.

If a data structure does not fit into the maximum sized memory unit that we can perform CAS or LL/SC on (I believe this is 8 bytes on all the architectures Linux currently supports), then we need to use more complex techniques. The conceptually simplest of these techniques is to make a copy of the entire data structure, change the copy, and substitute the changed copy for the original. The substitution is accomplished by having all references to the data structure go through a globally stored pointer to the current "active" data structure.

```
big_atomic_op (*object, operation)
{
  do {
     tmp = *object;
     make a complete copy of the data structure
     pointed to by tmp,
     where new is a pointer to the copy.
     perform the changes specified by "operation"
     on the copy.
  } while (!CAS(object,tmp,new));
  return the result of the operation
}
```

Note that I have carefully ignored the issue of reclaiming copies of the data structure that are no longer in use. This can be quite complex. In particular, when a process goes to sleep it can hold a pointer to an old copy of the data structure, and we must not delete that copy while the process sleeps. There are a variety of ways of getting around this, mostly involving reference counts or timing tricks, but I don't want to get into to much detail in this posting. For now, assume that some magic method does garbage collection for us, or, if you like, that we have unbounded memory available. (The problem of reclaiming memory in lock-free and wait-free algorithms is one of the most difficult issues that has to be dealt with, and this comes up again and again in different contexts).

Aside from the problem of reclaiming memory chunks, this technique is quite inefficient on several counts. First, it requires us to make complete copies of the object each time we change it. For a large object this may be excessive, especially if we only want to make a small change. Second, even non-conflicting parallel modifications to data structure are disallowed. This later point is somewhat subtle. The data structure may be lock-free, but changes to it will never take place faster than if we run each modification sequentially, one after another. Ignoring the cost of constructing a lock, this is accomplished by a locking approach. In practice I think the cost of constructing a lock will be no worse than the cost of an attempt at doing a lock-free update.

The first point can be addressed by recording changes to the data structure and using a clever protocol to perform the changes that get accepted in a strict ordering. Roughly speaking we proceed by using compare&swap to chain together a queue of changes and accesses, and perform the changes in sequence by having processors all attempt to make the changes independently. To prevent changes from being reversed or performed more than once every changeable word in the data structure has a version number associated with it that always increases after each change. Whenever a processor succeeds in making a change, the change gets dequeued. Eventually every change gets done, regardless of the actions of other processors. (Obviously I have skipped a huge amount of detail. I can point people to papers if they want to read up on this. At the moment, I don't believe this approach is practical).

Addressing the second point in full requires even more elaborate protocols. An explanation of these protocols is quite a bit beyond the scope of this note, but I will say that I am quite sure we will never want to implement them as they are quite expensive in both time and space. (The best known protocol requires O(n $m\hat{2}$) time and space in the worst case, where n is the number of processors, and m is the number of shared memory locations.)

One thing we *can* likely do, is construct efficient protocols that allow parallel reads of the data structure to proceed, even during modifications to the data structure. We would only need to serialize modifications. This could be done either with or without locking.

7 Basic Wait-Freedom

Consider the problem of implementing fetch and increment. In the previous section, we showed a lock-free implementation of the more general fetch and add. To make the algorithm of the previous section to wait-free we need to introduce the idea of "helping" unfinished tasks. Although the basic idea is fairly simply, it turns out to be non-trivial to implement. To simplify presentation I will show how it can be done using the DCAS instruction (compare and swap 2 locations). In the following procedure the variable i is the index of the process running the code.

```
fetchninc(int *X)
{
```

```
/* Get some place to store the result.
 * This must be initially 0,
 * and must be new for each operation.
 */
  result[i] = malloc(sizeof(int));

  do {
    tmp = X;
    DCAS(&X,tmp,tmp+1,result[i],0,tmp+1);
    /* stop when we or some helper gets
     * us a result
     */
  } while (*result[i] == 0);

  /* Help other processes finish their
   * tasks
   */
  for (j = 0; j < n; j++) {
    tmp2 = result[j];
    if (j != i && tmp2 != 0) {
      do {
        tmp = X;
        DCAS(&X,tmp,tmp+1,tmp2,0,tmp+1));
      } while (*result[j] == 0);
    }
  }
  return *result[i];
}
```

The idea here is that after successfully completing a fetch and add operation, a process tries to help every other process that has an unfinished operation. Processes that need help are exactly those that have a result entry pointing to a memory location containing 0. Of course, some other process may manage to help first. The key is that whenever a process fails, some process makes progress, and processes that are "helping" never finish helping until every process that started an operation before them has been successfully helped. As with the lock-free construction, note that this algorithm does not make any attempt to free up the memory allocated to store a result each time the counter is incremented.

8 Pratical Games

Although large scale wait-free and lock-free algorithms are often not practical, we may be able to take advantage of some practical "tricks" to help us get good algorithms.

The algorithms I have been talking about were all originally designed for a more hostile environment than we have to deal with. In particular, they have to be able to deal with arbitrary asynchronous scheduling of the next instruction in any process, and with an environment where it is impossible to tell a dead process from a slow process.

Inside the kernel we have some advantages:

- If we disable interrupts on a processor, we known that the segment of code we run while the interrupts are disabled will complete in a fixed amount of time (assuming no page faults, which in the case of Linux is true from inside the kernel).
- The underlying physical processors in a system have known execution speeds, and we can measure the difference in speed between different processors. This will give us timing information that we can potentially take advantage of.
- If we must stop what the kernel is doing in the middle of some wait-free computation in order to execute some user code or run an interrupt handler, we can arrange to roll-back changes in progress if we take to long to return to the original thread of execution.

This information can sometimes be used to construct better algorithms than are possible in a strictly asynchronous system.

For example, a major problem with allowing unlocked concurrent readers on a data structure that we are modifying is that some process may hold a pointer to a piece of memory that we want to free. To be more specific, consider searching a linked list for a structure matching some key, and suppose that we serialize accesses to items that we find by locking the items themselves. If we want to be able to free a piece of memory belonging to a structure that was deleted from the linked list, we must not do so if there is a risk that there is still some processor holding a pointer to the deleted data. A simple way to do this is to just wait to free the memory until we _know_ enough time has passed that no process can still have a pointer to the memory we want to free.

That's all for now folks!

The author


```
Eric Schenk
Dept. of Comp. Sci., Lund University
Box 118, S-221 00 LUND, Sweden
ph: +46-46 222 96 38
fax: +46-46 13 10 21
email: Eric.Schenk@dna.lth.se
www: http://www.dna.lth.se/~erics
```

Contents