On the Nature of Progress

Maurice Herlihy
Brown University
and
Nir Shavit
Tel-Aviv University

We identify a simple relationship that unifies seemingly unrelated progress conditions ranging from the deadlock-free and starvation-free properties common to lock-based systems, to non-blocking conditions such as obstruction-freedom, lock-freedom, and wait-freedom.

Properties can be classified along two dimensions based on the demands they make on the operating system scheduler. A gap in the classification reveals a new non-blocking progress condition, weaker than obstruction-freedom, which we call clash-freedom.

The classification provides an intuitively-appealing explanation why programmers continue to devise data structures that mix both blocking and non-blocking progress conditions. It also explains why the wait-free property is a natural basis for the consensus hierarchy: a theory of shared-memory computation requires an independent progress condition, not one that makes demands of the operating system scheduler.

1. INTRODUCTION

The advent of multicore architectures has provoked a renewed interest in concurrent data structures and algorithms. The literature encompasses a bewildering array of progress conditions. Some ("non-blocking") conditions guarantee progress even if one or more threads halt, while others do not. Some blocking conditions guarantee that threads will not deadlock, and some go further and rule out starvation.

On modern multiprocessor machines, programmers often use a variety of lock-based and non-blocking algorithms, sometimes mixing and matching progress conditions within a single system. (For example, consider lock-free, obstruction-free, and lock-based software transactional memory systems [13]). How can these data structures and algorithms work well together when they make incomparable and incompatible progress guarantees?

This paper proposes a novel *grand unified explanation* that ties together these seemingly unrelated progress conditions, ranging from the deadlock-free and starvation-free properties common to lock-based data structures, to the obstruction-free, lock-free, and wait-free properties that have been the focus of so much recent research. We are deliberately not presenting a "grand unified *theory*", (even though our explanation is not difficult to formalize), because our primary goal is to provide a clear, simple, and intuitively-appealing explanation how these dissimilar properties actually fit together. These ideas may seem straightforward, perhaps even obvious, but we have never seen this formulation in any published work.

We show that progress conditions can be classified as shown in Figure 1. The horizontal line separates properties that ensure *maximal* progress, that is, progress for all threads, from properties that ensure *minimal* progress, progress for only

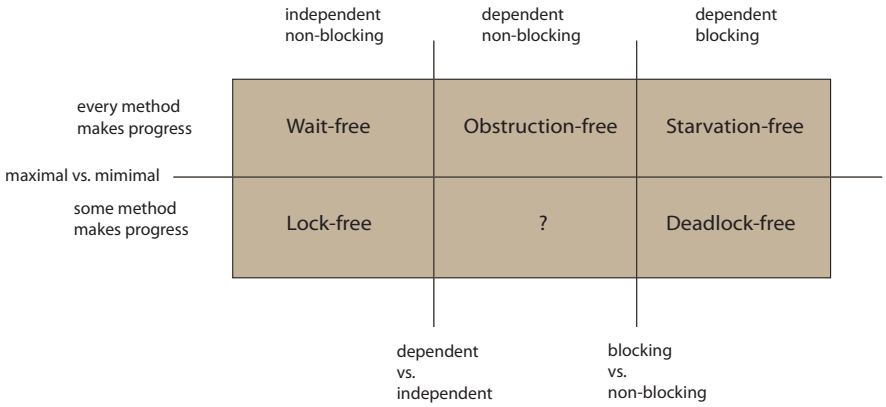

Fig. 1. The "Periodic Table" of Progress Conditions.

some threads. The vertical lines separate properties that depend on different kinds of guarantees provided by the operating system (OS) scheduler.

It is important to distinguish between *dependent* and *independent* progress conditions. At one extreme, the wait-free and lock-free properties are *independent* of the OS scheduler: they guarantee progress as long as threads are scheduled, but no matter how they are scheduled. The other properties are *dependent*: they rely on the OS scheduler to satisfy certain properties. The deadlock-free and starvation-free properties guarantee progress only if each thread eventually leaves each critical section, and the obstruction-free property [7] requires the scheduler to allow each thread to run in isolation for a sufficient duration.

If we further restrict our attention to schedulers that satisfy a *benevolent* property defined below, then the distinction between minimal and maximal progress along the horizontal axis vanishes: any algorithm that provides minimal progress provides maximal progress as long as the scheduler is benevolent. This is why algorithms that (in principle) permit starvation are so widely used in practice: programmers implicitly (and reasonably) assume that OS schedulers are benevolent in practice.

Here is how to unify the disparate progress conditions in the literature. Instead of analyzing each algorithm and its progress properties in isolation, focus on the interaction between the algorithm and the guarantees provided by the OS scheduler. Implicitly, programmers, whether they design starvation-free, deadlock-free, obstruction-free, lock-free, or wait-free data structures, all want the same thing: maximal progress[1]. They differ only in the assumptions they make about the OS scheduler.

One way to test an ambitious hypothesis is by its predictive power. Figure 1 contains a hole: the obstruction-free property has no minimal counterpart. We define a new *clash-free* property to fill this gap, and show it is strictly weaker than the obstruction-free property (addressing an open question due to Herlihy, Luchangco, and Moir [7]).

Finally, we observe that our classification explains why the wait-free property is

[1] "Purity of heart is to will one thing" – Sören Kierkegaard

a natural basis for the consensus hierarchy [6]: a theory of shared-memory computation requires an independent progress condition, not one that makes demands of the OS scheduler.

The remainder of this paper expands these observations. It builds on many papers, and a comprehensive survey of relevant literature would take up too much space. Instead, we refer the reader to books by Attiya and Welch [4], Lynch [14], and Taubenfeld [19], and to references cited in the paper body.

2. CONVENTIONAL EXPLANATION PROGRESS CONDITIONS

We start with a review of the conventional view of progress conditions taken from the literature. We then reformulate these notions in our unified model.

An *object* is a container for data. Each object provides a set of *methods* which are the only way to manipulate that object. Each object has a *class*, which defines the object's methods and how they behave. An object has a well-defined *state* (for example, the FIFO queue's current sequence of items).

The simplest way to synchronize concurrent access to an object is to associate a mutual exclusion lock with the object. Each method acquires the lock when it is called, and releases the lock when it returns. (We postpone consideration of methods that need to block, waiting until a condition is satisfied.)

Perhaps the weakest progress condition one could demand of a method that employs locks is that the method be *deadlock-free*, meaning that some thread trying to acquire the lock eventually succeeds. This condition guarantees that the system as a whole makes progress, but does not guarantee progress to individual threads. For example, a test-and-set spin lock is deadlock-free, because some thread will acquire a free lock. Here is an important point that we will explore later on: a deadlock-free lock guarantees progress only if every thread that acquires the lock eventually releases it. This requirement constrains both the scheduler, which cannot halt a thread in a critical section, and the software, which must use the lock correctly.

Sometimes we would like locks to have an even stronger property. A lock is *starvation-free* if every attempt to acquire the lock eventually succeeds. For example, a test-and-set spin lock is not starvation-free, because it is possible (though unlikely) that some thread's attempts to acquire the lock repeatedly fail. By contrast, queue locks [16] are typically starvation-free because threads acquire locks in the order they are requested. Like deadlock-free locks, starvation-free locks make sense only if every thread that acquires a lock eventually releases it.

We described the deadlock and starvation-free properties directly in terms of classical mechanisms such as locks and critical sections because that is usually how these properties are used in the literature [3]. Later on, when we make these notions precise, we will see that this approach is unsatisfactory for several reasons. First, it is relatively easy to devise obfuscated object implementations where it is difficult to identify a particular field as a lock and particular statements as critical sections. Second, it is unclear how to compare such a property to non-blocking properties that, by definition, do not use locks and critical sections. Finally, progress should not be defined in terms of locks, which are low-level mechanisms, but in a more general way in terms of completed method calls.

While operating system schedulers rarely, if ever, halt threads holding locks, it is possible that preemption might well delay a thread holding a lock, effectively blocking progress by other threads. To address such issues, a number of *non-blocking* progress conditions have emerged. A non-blocking condition ensures that

an arbitrary and unexpected delay by any thread (say, one holding a lock) does not prevent other threads from making progress.

A method is *lock-free* if some thread that calls that method eventually returns. A method is *wait-free* if every thread that calls that method eventually returns.

There is another non-blocking progress condition. We say that a method call executes *in isolation* for a duration if no other threads take steps during that time. A method is *obstruction-free* if every thread that calls that method returns if that thread executes in isolation for long enough. This condition is non-blocking, and is strictly weaker than the lock-free condition. It rules out the use of locks and mutual exclusion, but does not guarantee progress when multiple threads execute concurrently. Obstruction-free algorithms typically rely on a *contention manager* [8] module to delay threads so that a given thread can make progress. For example, a contention manager might employ a *backoff* delay policy: a thread that is about to conflict with another pauses to give the earlier thread time to finish.

This concludes our brief, high-level overview of the conventional view of progress conditions. We will revisit these defintions in the context of our simpler, unifying model. For lack of space the more standard parts of the definitions are omitted.

3. MODELING PROGRESS

Our model is adapted from Herlihy and Wing [10] and can be readily recast using the IO-automata model [15]. We assume *linearizability* [10] as our basic correctness condition.

We are interested in progress conditions for methods of abstract objects. A given object has a set of different methods, each of which can be invoked many times during an execution.

An execution of a concurrent object is modeled by a *history*, a finite sequence of method *invocation* and *response events*. A *subhistory* of a history H is a subsequence of the events of H. An *interval* is a subhistory consisting of contiguous events.

We model method invocations and responses using the standard terminology, which for lack of space we provide in Appendix A. We focus on two-level implementations that include an abstract object (the one being implemented) and concrete ones (the ones used in the implementation). Informally, each abstract method call is implemented by the sequence of concrete method calls it encompasses. We use this two-level approach because we care about the number of concrete steps needed to implement an abstract method call.

An abstract method call that never returns could happen in two ways: if it encompasses an infinite number of concrete steps, then the thread starved, but if it encompasses only a finite number of concrete steps, then the thread halted in the middle of the call. These situations are different, and must be distinguished.

A thread is *active* if it takes an infinite number of concrete steps (and is *suspended* if not), and an invocation is *active* if it is made by an active thread. To avoid clutter, we focus on implementation histories of a single abstract object with a single method, which is repeatedly called by all threads. It is easy to generalize these definitions to encompass multiple objects and methods, and to allow threads to shut down gracefully.

3.1 Minimal and Maximal Progress

In some sense, the weakest interesting notion of progress requires that the system as a whole continues to advance. Consider a fixed history H. An abstract method pro-

vides *minimal progress* in H if, in every suffix of H, some pending active invocation has a matching response. In other words, there is no point in the history where all threads that called the abstract method take an infinite number of concrete steps without returning. This condition might, for example, be useful for a thread pool, where we care about advancing the overall computation, but do not care whether individual threads are underutilized.

The strongest notion of progress, and arguably the one most programmers actually want, requires that each individual thread continues to advance. An abstract method provides *maximal progress* in a history H if in every suffix of H, every pending active invocation has a matching response. In other words, there is no point in the history where a thread that calls the abstract method takes an infinite number of concrete steps without returning. This condition might be useful for a web server, where each thread represents a customer request, and we care about advancing each individual computation.

3.2 The Scheduler's Role in Guaranteeing Progress

A history is *fair* if each thread takes an infinite number of concrete steps. A history is *uniformly isolating* if, for every $k > 0$, any thread that takes an infinite number of steps has an interval where it takes at least k concrete contiguous steps (that is, not interleaved with any other thread). Exponential back-off [1] is one possible mechanism to make schedules uniformly isolating (with high probability). Threads back off until all but one are inactive. Backoff durations can be controlled by the programmer.

We are now ready to reformulate the definitions of the progress properties surveyed in Section 2.

Definition 3.1. *A method implementation is* deadlock-free *if it guarantees minimal progress in every fair history, and maximal progress in some fair history.*

The restriction to fair histories captures the informal requirement that each thread eventually leaves its critical section. The definition does not mention locks or critical sections because progress should be defined in terms of completed method calls, not low-level mechanisms. Moreover, as noted, not all deadlock-free object implementations will have easily recognizable locks and critical sections.

The requirement that the implementation will provide maximal progress in some fair history is intended to rule out certain pathological cases. For example, the first thread to access an object might lock it and never release the lock. Such an implementation guarantees minimal progress (for the thread holding the lock) in every fair execution, but does not provide maximal progress in any execution. Clearly, such an implementation would not be considered acceptable in practice and is of no interest to us.

The starvation-free property is now straightforward:

Definition 3.2. *A method implementation is* starvation-free *if it guarantees maximal progress in every fair history.*

These properties are *dependent*: they are restricted to the subset of fair histories. Informally, these properties depend on a well-behaved operating system scheduler. We can capture the notion of dependency as follows:

Definition 3.3. *A progress condition is* dependent *if it does not guarantee minimal progress in every history, and is* independent *if it does.*

Here are the non-blocking properties.

DEFINITION 3.4. *A method implementation is* lock-free *if it guarantees minimal progress in every history, and maximal progress in some history.*

DEFINITION 3.5. *A method implementation is* wait-free *if it guarantees maximal progress in every history.*

The two properties above are independent: they apply to all histories. There is however a dependent non-blocking property:

DEFINITION 3.6. *A method implementation is* obstruction-free *if it guarantees maximal progress in every uniformly isolating history.*

4. A PERIODIC TABLE OF PROGRESS CONDITIONS

Although these progress conditions may have seemed quite different, each provides either minimal or maximal progress with respect to some set of histories. The result is a simple and regular structure illustrated in the "periodic table" shown in Figure 1 (and its more complete counterpart in Figure 2). These observations may appear so simple as to be obvious in retrospect, but we have never seen them described in this way.

There are three dividing lines, two vertical and one horizontal, that split the five conditions. The leftmost vertical line separates dependent conditions from the rest. The lock-free and wait-free properties apply to any histories, while obstruction-freedom, starvation-freedom, and deadlock-freedom require some kind of external scheduler support to guarantee progress.

The rightmost vertical line separates the blocking and non-blocking conditions. The lock-free, wait-free, and obstruction-free conditions are non-blocking: if a suspended thread stops at an arbitrary point in a method call, at least some active threads can make progress. The deadlock-free and starvation-free conditions do not have this property.

Finally, the horizontal line separates the minimal and maximal progress conditions. The minimal conditions guarantee the system as a whole makes progress while the maximal conditions guarantee that each thread makes progress. For brevity, *minimal* progress properties encompass the lock-free and deadlock-free properties, while *maximal* properties encompass the wait-free, starvation-free, and obstruction-free properties. In the coming sections we will see several ways to cross this line: "helping" (Section) and benevolent schedulers (Section 8). Helping [6] is an algorithmic mechanism which has threads avoid being delayed by others by completing the delayed threads' work in their place. Benevolence is an assumption on the scheduler behavior that allows one to avoid the high communication costs associated with helping.

There is a hole in Figure 1: a conspicuous empty slot occupied by a dependent, non-blocking progress property that guarantees minimal progress in uniformly-isolating histories.

DEFINITION 4.1. *A method implementation is* clash-free *if it guarantees minimal progress in every uniformly isolating history, and maximal progress in some such history.*

In the next two sections we show that being clash-free is strictly weaker than being obstruction-free, answering the open question raised by Herlihy, Luchangco,

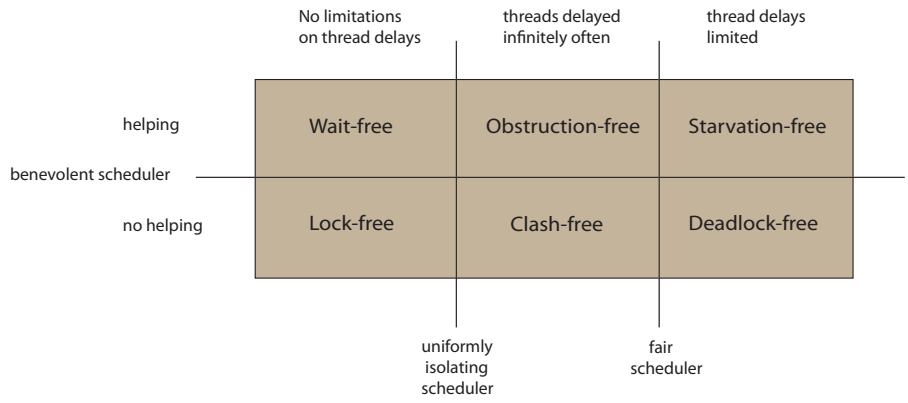

Fig. 2. Clash-freedom: the missing element.

```
1  public interface Consensus<T> {
2    T decide(T value);
3  }
```

Fig. 3. Consensus Object Interface

and Moir [7], whether obstruction-freedom is the weakest interesting non-blocking progress condition[2].

5. MINIMAL AND MAXIMAL-PROGRESS UNIVERSAL CONSTRUCTIONS

In this section we define two universal constructions. They are adapted from the standard lock-free and wait-free universal constructions in [9]. For lack of space we provide the code and explanations on how they operate in the appendix. The minimal-progress universal construction provides a linearizable implementation of any sequential object. The construction relies on a supply of one-time consensus objects (see Figure 3), ones in which each thread can call the decide() method at most once in any execution history.

It can be shown (The proofs follow easily from the proofs in [9] and we do not show them for lack of space.) that the *minimal progress universal construction* (a

[2]Clash-freedom is arguably the Einsteinium of progress conditions. Like Einsteinium, symbol *Es*, atomic number 99, it fills a vacant table slot, yet does not occur naturally in any measurable quantities and has no commercial value.

```
1  public interface SeqObject {
2    public abstract Response apply(Invocation invoc);
3  }
```

Fig. 4. A Generic Sequential Object: the apply() method applies the invocation and returns a response.

variation of the lock-free construction in [9]) provides the minimal form of whatever progress guarantee is provided by the consensus objects it uses: it is lock-free if the consensus objects are lock-free or wait-free, clash-free if they are clash-free or obstruction-free, and deadlock-free if they are deadlock-free or starvation-free.

The *maximal-progress universal construction* (a variation of the wait-free construction in [9]) does the same, except that it provides the maximal form of the consensus objects' progress guarantee: it is wait-free if the consensus objects are lock-free or wait-free, obstruction-free if they are clash-free or obstruction-free, and starvation-free if they are deadlock-free or starvation-free (notice that for one-time objects the minimal progress conditions are by definition equal to the maximal progress conditions).

We use these constructions to demonstrate our separation result, that there exists a clash-free object implementation that is not obstruction-free.

Figure 4 shows a *generic* definition for a sequential object. Each object is created in a fixed initial state. The apply() method takes as argument an *invocation* which describes the method being called and its arguments, and returns a *response*, containing the call's termination condition (normal or exceptional) and the return value, if any. For example, a stack invocation might be push() with an argument, and the corresponding response would be normal and *void*.

Figures 6 and 7 show a universal construction that transforms any sequential object into a linearizable concurrent object satisfying the same minimal progress condition as its consensus objects.

For simplicity, this construction assumes that sequential objects are *deterministic*: if we apply a method to an object in a particular state, then there is only one possible response and one possible new object state. We can represent any object as a combination of a sequential object in its initial state and a *log*: a linked list of nodes representing the sequence of method calls applied to the object (and hence the object's sequence of state transitions). A thread executes a method call by scanning the log, starting at the oldest node (the tail), until it finds the newest node (the head). It then uses the node's consensus object to append its own node to the list. It then retraverses the log, applying the method calls to a private copy of the object. The thread finally returns the result of applying its own operation. It is important to understand that only the head of the log is mutable: the initial state and nodes following the head never change.

This algorithm works even when apply() calls are concurrent because the prefix of the log up to the thread's own node never changes. The losing threads, who failed to append their own nodes, must start over.

We can now make an interesting observation about Table 2: by adding the "helping" mechanism, one can implement any maximal (wait-free, obstruction-free, or deadlock-free) linearizable object from a minimal (lock-free, clash-free, or starvation-free) one-time consensus object. In other words, with an added implementation cost one can go from minimal to maximal progress.

6. SEPARATION RESULTS

THEOREM 6.1. *There exists a clash-free object implementation that is not obstruction-free.*

PROOF. Herlihy, Luchangco, and Moir [7] observe that one can implement an obstruction-free (and hence clash-free) one-time consensus object by derandomizing the randomized consensus protocol of Aspnes and Herlihy [2] (replacing the random

```
1  public class Queue<T> {
2    T items[];
3    int head, size;
4    int capacity;
5    public Queue(int capacity) {
6      items = (T[]) new Object[capacity];
7      head = size = 0;
8    }
9    public synchronized T deq() {
10     while (size == 0) {
11       wait();
12     }
13     notifyAll();
14     size--;
15     return items[head++];
16   }
17   public synchronized void enq(T x) {
18     while (size == capacity) {
19       wait();
20     }
21     notifyAll();
22     items[(head + size) % capacity] = x;
23     size++;
24   }
25 }
```

Fig. 5. A FIFO queue with partial methods.

coin by a deterministic one). Similarly, it is easy to implement a deadlock-free consensus object using a mutual exclusion lock.

We now give an example of a history in which the minimal-progress construction using an obstruction-free consensus object is clash-free but not obstruction-free.

Pick one favored thread A. Run each thread until it reaches Line 12. When all n threads have arrived, run each one through the entire loop between Lines 12 and 14. After all the threads have executed the loop, allow A to call and return from the consensus object, completing its own call. The others call the consensus object after A's call has returned, so A succeeds while the others fail.

If we repeat this interleaving, the result is a uniformly-isolating history, because each thread scans the log in isolation, and each time the log is longer.

It follows that being clash-free is a weaker condition than being obstruction-free. □

This closes the open question raised by Herlihy, Luchangco, and Moir, [7].

7. PARTIAL METHODS

So far we have considered only *total methods*, methods that are always capable of returning a response. Much of concurrent programming, however, makes use of *partial methods* that block when called in certain states. For example, Figure 5 shows how one might implement a partial FIFO queue in the Java^{TM} programming language. The deq() method is *synchronized*: it acquires an implicit lock when it is called and releases it when it returns. If it encounters an empty queue (Line 10) the method temporarily releases the lock and suspends itself. Later, if an enq() call adds an item to the queue, it calls notifyAll () to wake up any suspended dequeuers.

These threads reacquire the lock and retest whether the queue is empty. We say that an invocation is *enabled* if there is a response it could return.

It is not obvious how to define minimal and maximal progress for partial methods. For example, one might be tempted to say that a method provides maximal progress in a history if no pending invocation is infinitely often enabled. (In other words, any invocation enabled often enough will return.) The following example illustrates why this definition is problematic. Consider an empty FIFO queue, where thread A calls a blocking deq(). Because the queue is empty, the method cannot return, so A's invocation is disabled, and A blocks. Thread B then enqueues an item, enabling A's invocation, but then immediately dequeues that item, again disabling A's invocation. If B repeats this sequence forever, then A's invocation is infinitely often enabled, yet A never returns. Should we deem this history as not providing maximal progress?

The problem with rejecting such a history is that it is permitted by all threads packages of which we are aware. For example, in the Queue implementation of Figure 5, A's deq() call releases the lock and waits. B's enq() call notifies A asynchronously, but before the operating system reschedules A, B's deq() removes the item. (Similar behavior can occur also with the *Pthreads* and *.Net* threads libraries.) We should avoid any definition of maximal progress that cannot be implemented.

A pending invocation is *continually enabled* in H if it is enabled at every step in some suffix of H. Once an invocation becomes continually enabled, then when its thread is awakened and resumed, however asynchronously, it is certain to discover a response.

We are ready to propose another definition. A method provides *minimal progress* in H if, in every suffix of H where some active invocation is continually enabled, some pending active invocation has a matching response. In other words, at no point in H does the method have continually-enabled active invocations, none of which ever returns.

Similarly, a method provides *maximal progress* in H if it has no continually-enabled active invocations ever.

The condition we rejected is essentially *strong fairness*, while one we adapted is *weak fairness* [12; 15].

8. BENEVOLENT SCHEDULERS

In practice, programmers often use implementations that guarantee only minimal progress, not because they do not care about lack of progress by individual threads, but because such lack of progress almost never happens under normal circumstances. For example, while programs that use spin locks are deadlock free, they are not starvation-free because the scheduler might schedule one particular thread only when the lock is held by another thread. In practice, few programmers worry about this prospect because they do not expect schedulers to persecute individual threads.

Let us make this notion more precise. Consider an algorithm that guarantees a minimal progress condition. A scheduler is *benevolent* for that algorithm if it guarantees maximal progress for that algorithm in every history it permits. Such a guarantee can also be probabilistic in nature.

For example, an *oblivious scheduler* is a fair scheduler that chooses the next thread to take a step uniformly at random. Now consider a deadlock-free spin lock algorithm where each thread repeatedly acquires the lock (by spinning), executes an operation, and releases the lock.

THEOREM 8.1. *An oblivious scheduler is benevolent (with probability one) for any deadlock-free spin-lock algorithm.*

PROOF. Because the scheduler is fair, the lock must become free an infinite number of times. Each time the lock becomes free, that thread is chosen with probability at least $1/n$, implying that the thread starves with probability measure zero. □

Along the same lines, we can use exponential backoff [1] to make lock-free algorithms wait-free.

We have barely scratched the surface with these theorems, and we leave it as an open question to derive more theorems of this nature. In particular, this approach provides a new way to think about *contention managers* [8], application-specific modules that modify the behavior of schedulers.

9. FOUNDATIONS OF SHARED-MEMORY COMPUTABILITY

Our classification of dependent progress conditions has implications for the foundations of shared-memory computability. Lamport's register-based approach [11] to read-write memory computability is based on wait-free implementations of one register type from another. Similarly, Herlihy's consensus hierarchy [6] applies to wait-free or lock-free object implementations. Combined, these structures form the basis of a *theory of concurrent shared-memory computability* [9] that explains what objects can be used to implement other objects in an asynchronous shared memory multiprocessor environment.

One might ask, however, why such a theory should rest on non-blocking progress conditions (that is, wait-free or lock-free) and not on locks. After all, locking implementations are common in practice. Moreover, the obstruction-free condition is a non-blocking progress condition where read-write registers are universal [7], effectively leveling the consensus hierarchy.

We are now in a position to address this question. Perhaps surprisingly, Figure 2 suggests that the reason to use the lock-free and wait-free conditions as a basis for a computability theory is not because they are non-blocking. Rather, it is because they are *independent* progress conditions that do not rely on the good behavior of the operating system scheduler. A theory based on a dependent condition would require strong assumptions about the environment in which programs were executed.

By analogy with sequential Church-Turing computability, using a dependent condition is like relying on an oracle to recognize languages. One could easily devise a finite automaton that together with a sufficiently powerful oracle could identify any context-free language. Such an automaton would by construction be weaker than a Turing machine, and so its real computing power would be masked by the oracle.

By analogy, when studying the computational power of synchronization primitives, it is unsatisfactory to rely on the operating system to ensure progress, both because it obscures the inherent synchronization power of the primitives, and because we might want to use such primitives in the construction of the operating system itself. For these reasons, a satisfactory theory of shared-memory computability should rely on independent progress conditions such as the wait-free or lock-free properties, and not on the other, dependent properties.

We have discussed progress properties of individual methods, not of entire objects, because it is often useful for objects to provide different methods that satisfy different progress properties. For example, Heller et al. [18]) describe a linked-list

that supports starvation-free lock-based insertion and removal, but with a wait-free search.

Our definitions are easily generalized to *collections* of methods ranging from a single method to all of an object's methods. For example, the Harris-Michael lock-free list [5; 17] provides add(), remove(), and contains() methods. Each method on its own is obstruction-free, but the collection of all methods taken together is lock-free.

10. CONCLUSIONS AND FURTHER RESEARCH

This paper proposes a novel way to impose order on the previously unstructured world of progress conditions for algorithms on multicore machines. Much, however, remains to be done.

For example, it would be of great interest to identify new classes of benevolent schedulers. It could be of practical importance to understand how *contention managers* [8], application-specific modules that modify the behavior of schedulers, serve in making them benevolent. It would be interesting to better understand the role of "helping" in overcoming scheduler limitations, possibly finding lower bounds on the cost of universal helping, a cost that perhaps captures the value of the benevolence scheduling property.

Finally, our approach implies that real-world operating system designers should be aware of the progress guarantees that their systems and services provide. Perhaps it is time that these criteria be formally stated and made available to the user in a manner similar to how memory models are defined with respect to correctness.

11. ACKNOWLEDGMENTS

We are grateful to Ori Shalev and Victor Luchangco for their help in formulating and crystallizing some of the concepts presented here.

REFERENCES

[1] Agarwal, A., and Cherian, M. Adaptive backoff synchronization techniques. In *Proceedings of the 16th International Symposium on Computer Architecture* (May 1989), pp. 396–406.

[2] Aspnes, J., and Herlihy, M. Fast randomized consensus using shared memory. *J. Algorithms 11*, 3 (1990), 441–461.

[3] Attiya, H., and Welch, J. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics.* John Wiley and Sons, 2004.

[4] Attiya, H., and Welch, J. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics, 2nd Edition.* John Wiley & Sons, Inc., New York, NY, 2004.

[5] Harris, T. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of 15th International Symposium on Distributed Computing (DISC 2001), Lisbon, Portugal* (Oct. 2001), vol. 2180 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 300—314.

[6] Herlihy, M. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems 13*, 1 (January 1991), 124–149.

[7] Herlihy, M., Luchangco, V., and Moir, M. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems* (2003), IEEE, pp. 522–529.

[8] Herlihy, M., Luchangco, V., Moir, M., and William N. Scherer, I. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing* (New York, NY, USA, 2003), ACM, pp. 92–101.

[9] Herlihy, M., and Shavit, N. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers, San Mateo, CA, 2008.

[10] Herlihy, M. P., and Wing, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS) 12*, 3 (1990), 463–492.

[11] Lamport, L. On interprocess communication. part ii: Algorithms. *Distributed Computing 1*, 2 (1986), 86–101.

[12] Lamport, L. Fairness and hyperfairness. *Distributed Computing 13*, 4 (2000), 239–245.

[13] Larus, J. R., and Rajwar, R. *Transactional Memory*. Morgan and Claypool, 2006.

[14] Lynch, N. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 2008.

[15] Lynch, N., and M.Tuttle. An introduction to input/output automata. *CWI Quarterly 2* (1989), 219–246.

[16] Mellor-Crummey, J., and Scott, M. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems 9*, 1 (1991), 21–65.

[17] Michael, M. M. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures* (2002), ACM Press, pp. 73–82.

[18] S. Heller, M. Herlihy, V. L. M. M. W. S., and Shavit, N. A lazy concurrent list-based set algorithm. In *Proc. of the 9th International Conference On Principles Of Distributed Systems (OPODIS 2005)* (2005), pp. 3–16.

[19] Taubenfeld, G. *Synchronization Algorithms and Concurrent Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

APPENDIX

A. BASIC DEFINITIONS

This section provides the missing standard definitions used by our model. An execution of a concurrent system is modeled by a *history*, a finite sequence of method *invocation* and *response events*. A *subhistory* of a history H is a subsequence of the events of H. An *interval* is a subhistory consisting of contiguous events. We model method invocations and responses using the standard terminology, which for lack of space we provide in Appendix A.

A method invocation is labeled with an object, an method, its arguments, and the calling thread's ID. A response is labeled with an object, a condition (either *Ok* or an exception), a result value, and the calling thread's ID. Sometimes we refer to an event labeled with thread A as a *step* of A.

A response *matches* an invocation if they have the same object and thread. A *method call* in a history H is a pair consisting of an invocation and the next matching response in H. We need to distinguish calls that have returned from those that have not: an invocation is *pending* in H if no matching response follows. An *extension* of H is a history constructed by appending responses to zero or more pending invocations of H. Sometimes we ignore all pending invocations: $complete(H)$ is the subsequence of H consisting of all matching invocations and responses.

In some histories, method calls do not overlap: A history H is *sequential* if the first event of H is an invocation, and each invocation, except possibly the last, is immediately followed by a matching response.

Sometimes we focus on a single thread or object: a *thread subhistory*, $H|P$ ("H at P"), of a history H is the subsequence of all events in H whose thread names are P. An *object subhistory* $H|x$ is similarly defined for an object x. In the end, all that matters is how each thread views what happened: two histories H and H' are *equivalent* if for every thread A, $H|A = H'|A$. Finally, we need to rule out histories that make no sense: A history H is *well-formed* if each thread subhistory is sequential. All histories we consider here are well-formed. Notice that thread subhistories of a well-formed history are always sequential, but object subhistories need not be.

We assume *linearizability* [10] as our basic correctness condition. For brevity, we omit the technical definition, but informally linearizability states that any concurrent history is equivalent to a sequential history in which each method call "takes effect" instantaneously in sometime between the method's invocation and its response. Linearizability does not imply any notion of progress.

An object *implementation* is a set of histories in which events of two objects, a *concrete* object c and an *abstract* object a are interleaved in a constrained way: for each history H in the implementation, (1) the subhistories $H|c$ and $H|a$ satisfy the usual well-formedness conditions; and (2) for each thread A, each concrete method call $H|A$ lies within an abstract method call in $H|A$. Informally, each abstract method call of is implemented by the sequence of concrete method calls it encompasses. An implementation is *correct* if for every history H in the implementation, $H|a$ is linearizable.

B. UNIVERSAL CONSTRUCTION CODE

The maximal-progress universal construction appears in Figure 8. We must guarantee that every thread completes an apply() call within a finite number of steps,

```
1  public class Node {
2    public Invoc invoc;                      // method name and args
3    public Consensus<Node> decideNext; // decide next Node in list
4    public Node next;                        // the next node
5    public int seq;                          // sequence number
6    public Node(Invoc invoc) {
7      invoc = invoc;
8      decideNext = new Consensus<Node>()
9      seq = 0;
10   }
11   public static Node max(Node[] array) {
12     Node max = array[0];
13     for (int i = 1; i < array.length; i++)
14       if (max.seq < array[i].seq)
15         max = array[i];
16     return max;
17   }
18 }
```

Fig. 6. The Node class

```
1  public class MinUniversal {
2    private Node tail;
3    public MinUniversal() {
4      tail = new Node();
5      tail.seq = 1;
6    }
7    public Response apply(Invoc invoc) {
8      int i = ThreadID.get();
9      Node prefer = new Node(invoc);
10     Node head = tail;
11     while (prefer.seq == 0) {
12       while (head.next != null) {
13         head = head.next;
14       }
15       Node after = head.decideNext.decide(prefer);
16       head.next = after;
17       after.seq = head.seq + 1;
18     }
19     SeqObject myObject = new SeqObject();
20     current = tail.next;
21     while (current != prefer){
22       myObject.apply(current.invoc);
23       current = current.next;
24     }
25     return myObject.apply(current.invoc);
26   }
27 }
```

Fig. 7. The minimal-progress universal construction

that is, no thread starves. To guarantee this property, threads making progress must help less fortunate threads to complete their calls.

To allow helping, each thread shares with other threads the apply() call that it

```
1  public class MaxUniversal {
2    private Node[] announce; // array added to coordinate helping
3    private Node[] head;
4    private Node tail = new node(); tail.seq = 1;
5    for (int j=0; j < n; j++){head[j] = tail; announce[j] = tail};
6    public Response apply(Invoc invoc) {
7      int i = ThreadID.get();
8      announce[i] = new Node(invoc);
9      head[i] = Node.max(head);
10     while (announce[i].seq == 0) {
11       Node before = head[i];
12       Node help = announce[(before.seq + 1 % n)];
13       if (help.seq == 0)
14         prefer = help;
15       else
16         prefer = announce[i];
17       after = before.decideNext.decide( prefer );
18       before.next = after;
19       after.seq = before.seq + 1;
20       head[i] = after;
21     }
22     SeqObject MyObject = new SeqObject();
23     current = tail.next;
24     while (current != announce[i]){
25       MyObject.apply(current.invoc);
26       current = current.next;
27     }
28    head[i] = announce[i];
29    return MyObject.apply(current.invoc);
30   }
31 }
```

Fig. 8. The maximal-progress universal construction.

is trying to complete. We add an n-element announce[] array, where announce[i] is the node thread i is currently trying to append to the list. Initially all entries refer to the sentinel node, which has a sequence number 1. A thread i *announces* a node when it stores the node in announce[i].

To execute apply(), a thread first announces its new node. This step ensures that if the thread itself does not succeed in appending its node onto the list, some other thread will append that node on it's behalf. It then proceeds as before, attempting to append the node into the log.