

# Summary

In modern computers, many processes run concurrently, e.g. the window manager, the mailer, the word processor, the clock, etc. All these processes need access to the shared memory, but if they would read and modify the shared memory in unrestricted ways, the data would become inconsistent. The classical way to avoid this is to enforce synchronization by means of locks. This may lead to blocking.

Due to blocking, the classical synchronization paradigms using locks can incur many problems such as convoying, priority inversion and deadlock. Over the past two decades a number of researchers have proposed techniques for designing lock-free implementations. These techniques allow concurrent update of shared data structures without resorting to critical sections protected by locks. Essential for such implementations are using advanced machine instructions such as *LL/SC* or *CAS*. Lock-free implementations guarantee that within a finite number of steps always some process trying to perform an operation on the object will complete its task. They can avoid many problems arising due to failures and priority inversion. Lock-free synchronization is very important especially in real-time systems since if the blocked process is performing a high-priority or real-time task, it is highly undesirable to halt its progress.

Ensuring the correctness of the design at the earliest possible stage is a major challenge in any responsible system development. Lock-free algorithms are in general very complex and hard to design correctly. The only technique we see is to specify the programming model of the behavior of the system in a formal language, and to mathematically verify that the system design and implementation satisfy certain properties such as safety and liveness. In general there are two verification methods for system design: model checking and theorem proving. Model checking relies on automatic exhaustive exploration of the reachable state space of the system. The lock-free algorithms presented in this thesis are too data-intensive and complicated for model checking. Theorem proving can avoid the so-called

state explosion by a compact (or logical) representation of states and state transformations. Therefore, we have chosen the interactive theorem prover PVS for mechanical support. Except informal proofs for a few liveness properties, all the algorithms and proofs presented in this thesis have been formalized and checked with PVS.

It is very difficult to fairly compare the performance of different concurrent implementations of one algorithm, since the performance of parallel processing is very much influenced by the machine architecture, the relative sizes of data structures compared to sizes of caches, and even the scheduling of processes on processors. Even a good comparison might be disputable, and become outdated by the introduction of other architectures. Therefore, we cannot offer full empirical support for the algorithms presented.

Chapter 2 presents an efficient lock-free algorithm for hash tables with open addressing. The algorithm is dynamic in the sense that it allows the hash table to grow and shrink as needed. Experiments indicate that the algorithm scales up linearly with the number of processes. It seems to require on average only constant time for insertion, deletion or accessing of elements. An apparent weakness of our algorithm is the worst-case space complexity proportional to the product of the number of processes and the size of the hash table. However, when all processes make ordinary progress and the hash table is not too small, the actual memory requirement is proportional to the size of the table.

Though PVS provided great help for managing and reusing the proofs, we have to admit that the verification for the algorithm was very complicated due to the complexity of the algorithm. The whole correctness proof of the algorithm contains around 200 invariants. The total verification effort can roughly be estimated to consist of two man years.

Chapter 3 formalizes Herlihy's general methodology for transferring a sequential implementation of any data structure into a lock-free synchronization, and presents a lock-free pattern as a reduction theorem. The reduction theorem enables us to reason about a lock-free program to be designed on a higher level than the synchronization primitives *LL/SC*. It is based on refinement mappings as described by Lamport. Application of this theorem simplifies the verification effort for lock-free algorithms since fewer invariants are required and some invariants are easier to discover and formulate without considering the internal structure of the final implementation. Moreover, two enhanced alternative algorithms are presented that avoid unnecessary copying for large objects in cases where only small part of the objects are modified.

Many machines provide either *CAS* or *LL/SC*, but not both. Chapter 4 presents a

similar lock-free pattern based on the weaker atomic primitive *CAS* without causing the so-called *ABA* problem or problems with wrap around. It is a variation of Herlihy’s general methodology for lock-free transformation.

Chapter 5 presents a lock-free parallel algorithm for mark&sweep garbage collection (GC) in a realistic model using synchronization primitives *LL/SC* or *CAS*. A number of mutators and collectors can simultaneously operate on the data structure. In particular no strict alternation between usage and cleaning up is necessary contrary to what is common in most other garbage collection algorithms. To simplify the proof, we first extend the specification to a high-level implementation, then verify the correctness of the high-level implementation, and finally apply the reduction theorem developed in chapter 3 to implement the higher-level atomic steps by means of the low-level primitives.

The algorithm employs a procedure *Mark\_stack*, which is mainly a form of graph search, and was initially designed as a recursive procedure. It is a surprise that the elimination of the recursion in favor of an explicit stack makes the proof possible. It would be much more difficult (if possible at all) to prove the correctness of the recursive procedure where we have to rely on the fixed point semantics of recursive procedures or some other denotational semantics.

Apart from safety properties, we have also considered the important problem of verifying liveness properties using the strong fairness assumption. Liveness properties are often expressed using the “*leads-to*” relation. They are widely thought to be harder to verify than safety properties. We found that the “*steps-to*” ( $\triangleright$ ) relation and the “*unless*” ( $\mathcal{U}$ ) relation are quite useful to prove the “*leads-to*” ( $\circ\rightarrow$ ) relation, since these two relations only involve a single step, and they can be checked directly by PVS with the help of invariants.

A main observation is that the PVS proof is surprisingly complex compared to the size of the algorithm proved. In [26], Havelund and Shankar admit that their reduction did not make the proof simpler because the major effort has gone to show the refinement relation. We have the impression that, in their case, the gap between the abstraction and the implementation is too big. In our case of the reduction theorem, there are only six invariants and it is not a burden to show the refinement relation between the abstraction and the implementation in the lock-free pattern. Using the reduction theorem, we only postulate 72 invariants in the whole correctness proof of the high-level implementation of lock-free GC since substantial pieces of the concrete program can be dealt with as atomic statements on the higher level. The total verification effort can roughly be estimated to

consist of half a man year. This is significantly less than what we afforded for the correctness of the lock-free hash tables.