Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms*

Maged M. Michael Michael L. Scott

Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
{michael,scott}@cs.rochester.edu

Abstract

Drawing ideas from previous authors, we present a new non-blocking concurrent queue algorithm and a new two-lock queue algorithm in which one enqueue and one dequeue can proceed concurrently. Both algorithms are simple, fast, and practical; we were surprised not to find them in the literature. Experiments on a 12-node SGI Challenge multiprocessor indicate that the new non-blocking queue consistently outperforms the best known alternatives; it is the clear algorithm of choice for machines that provide a universal atomic primitive (e.g. `compare_and_swap` or `load_linked/store_conditional`). The two-lock concurrent queue outperforms a single lock when several processes are competing simultaneously for access; it appears to be the algorithm of choice for busy queues on machines with non-universal atomic primitives (e.g. `test_and_set`). Since much of the motivation for non-blocking algorithms is rooted in their immunity to large, unpredictable delays in process execution, we report experimental results both for systems with dedicated processors and for systems with several processes multiprogrammed on each processor.

Keywords: concurrent queue, lock-free, non-blocking, `compare_and_swap`, multiprogramming.

*This work was supported in part by NSF grants nos. CDA–94–01142 and CCR–93–19445, and by ONR research grant no. N00014–92–J–1801 (in conjunction with the DARPA Research in Information Science and Technology—High Performance Computing, Software Science and Technology program, ARPA Order no. 8930).

1 Introduction

Concurrent FIFO queues are widely used in parallel applications and operating systems. To ensure correctness, concurrent access to shared queues has to be synchronized. Generally, algorithms for concurrent data structures, including FIFO queues, fall into two categories: *blocking* and *non-blocking*. Blocking algorithms allow a slow or delayed process to prevent faster processes from completing operations on the shared data structure indefinitely. Non-blocking algorithms guarantee that if there are one or more active processes trying to perform operations on a shared data structure, some operation will complete within a finite number of time steps. On asynchronous (especially multiprogrammed) multiprocessor systems, blocking algorithms suffer significant performance degradation when a process is halted or delayed at an inopportune moment. Possible sources of delay include processor scheduling preemption, page faults, and cache misses. Non-blocking algorithms are more robust in the face of these events.

Many researchers have proposed lock-free algorithms for concurrent FIFO queues. Hwang and Briggs [7], Sites [17], and Stone [20] present lock-free algorithms based on `compare_and_swap`. [1] These algorithms are incompletely specified; they omit details such as the handling of empty or single-item queues, or concurrent enqueues and dequeues. Lamport [9] presents a wait-free algorithm that restricts concurrency to a single enqueuer and a single dequeuer.[2]

Gottlieb *et al.* [3] and Mellor-Crummey [11] present algorithms that are lock-free but not non-blocking: they do not use locking mechanisms, but they allow a slow process to delay faster processes indefinitely.

[1] `Compare_and_swap`, introduced on the IBM System 370, takes as arguments the address of a shared memory location, an expected value, and a new value. If the shared location currently holds the expected value, it is assigned the new value atomically. A Boolean return value indicates whether the replacement occurred.

[2] A *wait-free* algorithm is both non-blocking and starvation free: it guarantees that every active process will make progress within a bounded number of time steps.