

Work in Progress: A Concurrent Priority Queue with Constant-Time Blocking

Anonymous

Abstract—In dynamic priority (DP) scheduling, kernels generally rely on priority queues to select the task to be executed. The choice of queue implementation introduces tradeoffs with respect to software overhead, memory usage and blocking times. A key consideration is thread-safety and memory safety. In this short paper, we sketch an unsorted, thread-safe in-place priority queue allowing an $\mathcal{O}(1)$ upper bound on inferred blocking, as well as $\mathcal{O}(1)$ insert, $\mathcal{O}(1)$ min and $\mathcal{O}(N)$ extractMin operations. The queue is implemented as a linked list backed by a fixed-size array, and can be allocated either statically, on the heap or on the stack. Potential applications include real-time scheduling, event management, and graph algorithms where predictable and minimal blocking times are paramount.

Index Terms—memory safety, priority queue, concurrency, blocking, defined behavior, real-time, data structures, critical section

I. INTRODUCTION

In embedded and real-time systems, DP scheduler kernel implementations typically rely on priority queues (PQs) to manage incoming task arrivals and retrieve the highest priority task to be executed. These data structures are challenging to implement correctly and efficiently in a concurrent environment; they have therefore been an area of extensive research.

One of the main challenges of such algorithms is limiting the blocking time. Indeed, synchronizing concurrent accesses to shared data structures often rely on mutual exclusion locks (*mutex*). On single-core systems, these locks are typically implemented as critical sections where the lock-region executes with interrupts disabled. However, schedulability criteria and task execution jitter are generally dependent on the length of the *longest* critical section in a given system; it is therefore of interest to limit worst-case lock duration to a strict minimum.

Some work has gone into implementing lock-free or concurrent PQs: the mound data structure presented in [1] achieves lock-free $\mathcal{O}(\log(\log(N)))$ insert and $\mathcal{O}(\log(N))$ extractMin operations. This PQ uses atomic compare-and-swap (CAS) operations which are assumed infallible; resource-limited embedded systems rarely implement truly infallible CAS operations, such as is the case for the ubiquitous ARM Cortex-M family of commercial, off-the-shelf (COTS) microcontrollers [2]. Other implementations use skip-lists and randomized access to amortize asymptotic time complexity [3]. Some work has also gone into limiting a PQ’s I/O operations between an internal cache and external memory, while retaining a favorable amortized time complexity for its operations [4]. Finally, while not a PQ, in [5], the authors propose a concurrent linked list, with node

manipulations also based on CAS operations. We however deem these approaches unsuitable for hard real-time kernel implementations targeting single-core COTS hardware, as the worst case blocking time is unbounded when accounting for retried operations.

In this paper we sketch a concurrent priority queue implementation, aiming for constant upper bounds on blocking times. Our approach is based on mutual-exclusion implemented as interrupt-free lock-regions, thus suitable for deployment on single-core COTS hardware.

A. Background and Motivation – Earliest Deadline First Scheduling

PQs are the cornerstone of Earliest Deadline First (EDF) kernel implementations, a DP scheduling paradigm. In common priority queues, elements are allowed to be extracted under some given ordering. Classical implementations include binary heaps, binomial heaps, Fibonacci heaps, and pairing heaps.

We consider an EDF kernel where arriving tasks J_i are each associated with two interrupt handlers:

- 1) They are first signalled to an arrival handler A_i . This handler captures the task’s arrival timestamp TS, and may then either dispatch the task to run on a lower priority handler, or enqueue the task in a priority queue for later retrieval and execution (Fig. 1 and Fig. 2 top).
- 2) As tasks are dispatched on their dispatch handlers D_i , their payload is executed when dispatch handler is executed by the interrupt controller. When the tasks completes, the dispatch handler take as scheduling decision. If $\text{min}(\text{PQ})$ has an absolute deadline which is shorter than the next task to execute’s deadline, then the highest priority task is extracted from $\text{extractMin}(\text{PQ})$ and dispatched (Fig. 2 bottom).
- 3) The priority of arrival and dispatch handlers is determined according to relative task deadlines, where the group of arrival handlers (Fig. 2 top) are assigned higher priority than the group of dispatch handlers (Fig. 2 bottom), to minimize time-stamp jitter.

Therefore, for the purpose of EDF scheduling, we seek a priority queue implementation with the following properties:

- Support for concurrent access from multiple execution contexts (e.g., threads or interrupts handlers).
- Bounded blocking times for concurrent access, with constant time $\mathcal{O}(1)$ upper bounds.
- Implementation should not depend on dynamic memory allocations, and should be resource efficient in terms of both memory and CPU usage.

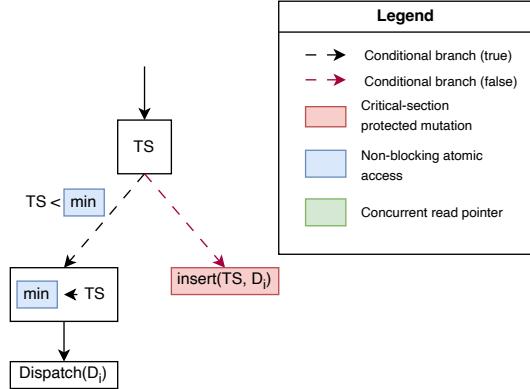


Fig. 1. Example implementation of an EDF arrival handler A_i .

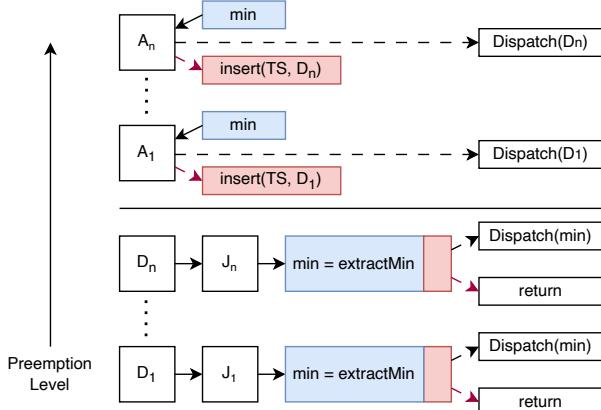


Fig. 2. Arrival and dispatch handlers sorted by preemption level. Arrival handlers are assigned higher priorities to minimize time-stamp jitter.

II. IN-PLACE PRIORITY QUEUE APPROACH

In the following we sketch the design and implementation of an in-place, concurrent priority queue, and discuss the design decisions in regard to the aforementioned requirements.

A. Array-based Linked List

For the sake of simplicity, we implement the priority queue as a linked list backed by a fixed-size array (Fig. 3). In-place operations are achieved by maintaining a free list of available nodes.

- *insert*: Insertion is unsorted: elements are appended at the tail of the list. Node updates are protected by a critical section, which is implemented by disabling

interrupts. This critical section is of constant time $\mathcal{O}(1)$, as it only involves mutating a single node.

- *min*: At all times, the data structure maintains a record of its minimum element separately from the main linked list, allowing a $\mathcal{O}(1)$ *min* operation. This record of the minimum element is updated at every list mutation (i.e., *insert* and on *extractMin*), guaranteeing it remains synchronized with the main data structure.
- *extractMin*: Extraction of the minimum element is performed by traversing the list from head to tail to find the minimum element, and then removing it from the list. This operation has a time complexity of $\mathcal{O}(N)$, where N is the number of elements in the queue. However, since all insertions are performed exclusively and atomically—via a critical section—at the queue tail, inspecting all nodes guarantees that the minimum element of the list is found, since no node can be inserted at a location already traversed by the reader pointer. Moreover, critical sections can be limited to the length of inspecting or mutating a single node—and are thus constant-time ($\mathcal{O}(1)$).

The implementation is thread-safe, thus allows for concurrent access from multiple execution contexts (the arrival and dispatch handlers, for the EDF case under study).

B. Work Stealing

Dispatch handlers execute concurrently, where a higher priority dispatch handler may preempt an ongoing *extractMin* operation. The higher priority handler steals the read cursor and the current minimum value encountered, continuing the traversal on behalf of the preempted *extractMin* operation.

Once the traversal is complete, the minimum element, if any, is removed from the list, protected by a critical section. The critical section is of constant time $\mathcal{O}(1)$, as it only involves a constant number of node updates. The stolen read cursor is set to indicate that the steal is complete, thus the resumed *extractMin* can immediately return without additional traversal. This queue is therefore intended for single-core systems, where only a single task may execute at any given time, and it is therefore unnecessary to attempt to dispatch multiple tasks simultaneously.

The restart-free implementation ensures that the amortized work for *extractMin* of each enqueued element is $\mathcal{O}(N)$.

C. Dispatcher Design

By performing the *extractMin* operation at the level of the currently highest priority task, we ensure that the task dispatch latency is free of priority inversion, and the currently most urgent task isn't blocked by queue operations from lower priority dispatch handlers.

III. CONCLUSIONS

In this short paper we have sketched a concurrent priority queue implementation, and argued constant time blocking times for all operations. The in-place designs allows for efficient memory usage and static allocation, meeting our requirements for hard real-time scheduling applications.

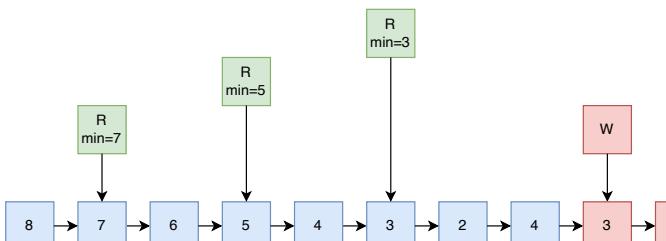


Fig. 3. Extraction of the minimum element from the priority queue, with 3 concurrent readers and a writer protected by a (global)critical section.

While priority queues using unsorted in-place array-based linked lists are well understood, the novelty here resides with the simplistic concurrent design, matching concrete requirements for hard-real time scheduling on single-core COTS hardware.

A. Future work

In future work, we plan to implement and evaluate the proposed design in a Stack Resource Policy based EDF scheduler. For the implementation, we intend to leverage on the Rust language for zero-cost abstractions, and provide safe APIs for inherently unsafe operations.

REFERENCES

- [1] Y. Liu and M. Spear, “A Lock-Free, Array-Based Priority Queue,” LU-CSE-11-004, 2011. Accessed: Nov. 19, 2025. [Online]. Available: https://engineering.lehigh.edu/sites/engineering.lehigh.edu/files/_DEPARTMENTS/cse/research/tech-reports/2011/lu-cse-11-004.pdf
- [2] Arm Limited, “Armv7-M Architecture Reference Manual.” [Online]. Available: <https://developer.arm.com/documentation/ddi0403/latest/>
- [3] H. Sundell and P. Tsigas, “Fast and Lock-Free Concurrent Priority Queues for Multi-Thread Systems,” in *Proceedings International Parallel and Distributed Processing Symposium*, Apr. 2003, p. 11 pp.–. doi: 10.1109/IPDPS.2003.1213189.
- [4] G. S. Brodal *et al.*, “External-Memory Priority Queues with Optimal Insertions,” in *33rd Annual European Symposium on Algorithms (ESA 2025)*, A. Benoit, H. Kaplan, S. Wild, and G. Herman, Eds., in Leibniz International Proceedings in Informatics (LIPIcs), vol. 351. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025, pp. 5:1–5:14. doi: 10.4230/LIPIcs.ESA.2025.5.
- [5] T. L. Harris, “A Pragmatic Implementation of Non-blocking Linked-lists,” in *Distributed Computing*, J. Welch, Ed., Berlin, Heidelberg: Springer, 2001, pp. 300–314. doi: 10.1007/3-540-45414-4_21.