Differentiated nonblocking: a new progress condition and a matching queue algorithm

David Y. C. Chan∗ Shucheng Chi† Vassos Hadzilacos‡ Sam Toueg§

March 23, 2021

Abstract

In this paper, we first propose a new liveness requirement for shared objects and data structures, we then give a shared queue algorithm that satisfies this requirement and we prove its correctness. We also implement this algorithm and compare it to a well-known shared queue algorithm that is used in practice [9]. In addition to having a stronger worst-case progress guarantee, our experimental results suggest that, at the cost of a marginal decrease in throughput, our algorithm is significantly fairer, by a natural definition of fairness that we introduce here.

1 Introduction

In this paper, we first propose a new liveness requirement for shared objects and data structures; we then give a shared queue algorithm that satisfies this requirement and we prove its correctness. We also implement this algorithm and compare it to a well-known shared queue algorithm that is used in practice [9]. In addition to having a stronger worst-case progress guarantee, our experimental results suggest that, at the cost of a marginal decrease in throughput, our algorithm is much more fair, by a natural definition of fairness that we introduce here. We now explain the paper’s motivation and contributions in more detail.

1.1 Background

Wait-freedom, a well-known liveness requirement for shared objects [6], guarantees that all the (non-faulty) processes make progress, i.e., every process completes every operation that it applies on the object. This is a very strong progress property but, unfortunately, it is also typically expensive to achieve: wait-free implementations often use an expensive “help” mechanism whereby processes that are fast must perform the operations of slower processes to prevent them from starving.

This is why many data structures and shared object implementations in practice are only non-blocking (also known as lock-free). Such implementations are often more efficient than wait-free ones, but they guarantee only that one process makes progress: if several processes apply operations to a nonblocking object, the object may reply only to the operations of a single process, while all the other processes can be stuck waiting for replies to their operations forever.

More recently, some papers [5, 4] considered a parametrized progress requirement, called k-nonblocking, that ranges from non-blocking (when k = 1) to wait-freedom (when k = n, the number of processes in the system): intuitively, when n processes apply operations, a k-nonblocking object is guaranteed to reply to the operations of at least k processes. A universal construction given in [4] shows that k-nonblocking can be achieved with a help mechanism whose worst-case cost is proportional to k rather than n. This suggests that for small k > 1, k-nonblocking implementations could be useful in practical settings as a compromise between the efficiency of non-blocking and the stronger progress guarantee of wait-freedom.

1.2 A new progress requirement

In this paper, we first identify a weakness in the non-blocking and k-nonblocking progress requirements, and propose a stronger versions of these two. To illustrate this weakness, consider a non-blocking queue that is

∗Department of Computer Science, University of Calgary.
†Institute for Interdisciplinary Information Sciences, Tsinghua University.
‡Department of Computer Science, University of Toronto.
§Department of Computer Science, University of Toronto.
used by two processes: a producer that repeatedly enqueues items (e.g., jobs to be executed) and a consumer that repeatedly dequeues items for processing. Here, the non-blocking progress guarantee of the queue could be useless: by definition, a non-blocking queue allows runs in which the producer succeeds in enqueueing every item (of an infinite sequence of items) while the consumer is not able to dequeue even one. So a non-blocking queue may prevent the producer from ever communicating any item to the consumer. Non-blocking also allows the symmetric case: the only process to ever succeed could be the (fast) consumer that keeps dequeuing \( \perp \) ("the queue is empty"), while the (slow) producer is not able to enqueue even one item.

It is easy to see that, for \( k < n \), the \( k \)-nonblocking liveness requirement, which guarantees that up to \( k \) processes make progress, suffers from the same problem. For example, if \( k \) producers enqueue items into a \( k \)-nonblocking queue, and one consumer dequeues items from that queue, it is possible that only the \( k \) enqueuers make progress while the consumer may fail to dequeue any item.

The above weakness of the non-blocking and \( k \)-nonblocking progress requirements extends to other shared objects. For example, consider a non-blocking or \( k \)-nonblocking dictionary, with the standard operations of \text{insert}, \text{delete} and \text{search}: the liveness requirement does not prevent executions where the only operations to complete are, say, \text{inserts} (which intuitively "write" the state of the dictionary), while every process that attempts to execute a \text{search} (which "reads" the state of the dictionary) may be prevented from completing its operation; this effectively renders the shared dictionary useless.

It is important to note that this weakness is not just permitted by the definition of non-blocking, but it is actually exhibited in some existing implementations of non-blocking objects. For example, consider the simple non-blocking version of the atomic snapshot algorithm described by Afek et al. in [1]. It is easy to see that this non-blocking algorithm allows the writers of the atomic snapshot to be the only ones to ever succeed, while all the readers (i.e., scanners) are stuck forever trying to read.

In general, the utility of an object is given by the set of operation types that it supports. By definition, the non-blocking or \( k \)-nonblocking progress requirements (for \( k < n \)) allow all but one of the operation types to be effectively suppressed; this may significantly degrade the object’s functionality.

In view of the above, we propose a natural strengthening of the \( k \)-nonblocking progress requirement (for \( k = 1 \) this is a strengthening of non-blocking) that ensures that no operation type is ever suppressed; we call it \emph{differentiated} \( k \)-nonblocking, or \( k \)-DNB, for short. Intuitively, \( k \)-DNB guarantees that, for every operation type \( T \), if a process is stuck while performing an operation of type \( T \) (i.e., it takes an infinite number of steps without completing this operation) then at least \( k \) processes complete infinitely many operations of type \( T \). Note that \( k \)-DNB guarantees that an object remains fully functional, in the sense that no operation type can be suppressed; as with \( k \)-nonblocking, \( k \)-DNB also guarantees that at least \( k \) processes make progress; in other words, \( k \)-DNB implies \( k \)-nonblocking. \footnote{Intuitively, in a non-blocking implementation, this may occur because the producer is faster than the consumer and its repeated enqueues interfere with every dequeuing attempt by the slower consumer.}

### 1.3 A corresponding algorithm and its proof

To demonstrate the feasibility of this new progress property, we derive and prove the correctness of a 2-DNB algorithm for a shared queue: this algorithm guarantees that at least two enqueuers and at least two dequeuers, all operating concurrently, make progress (we sometimes refer to this as the "2+2 non-blocking" property). We chose to design a queue because of its wide use in distributed computing, and we chose \( k = 2 \) because we wanted an algorithm that guarantees progress by more than one process while remaining efficient. Intuitively, our 2-DNB queue algorithm is efficient because its help mechanism is light-weight: (1) each process makes at most a single attempt to help one other process to do its operation before doing its own operation, and (2) dequeuers help only dequeuers, and enqueuers help only enqueuers. In fact, in our algorithm, enqueuers do not interfere (by helping or competing with) dequeuers, and vice versa.

In an experimental evaluation of our 2-DNB queue algorithm, we identified another desirable feature of this algorithm, which we call fairness and describe below.

### 1.4 Fairness

It has been observed that non-blocking objects often behave as if they were wait-free, and this is because the pathological scenarios that lead to blocking are rarely, if ever, encountered in practice [7]. Wait-freedom however, does not imply fairness—a property that may be desirable. To illustrate what we mean by fairness, suppose that two processes \( p \) and \( q \) repeatedly apply the same type of operation on a non-blocking object. If both processes have similar speed (i.e., steps per unit of time) then, by symmetry, we would expect both \( p \) and \( q \) to complete roughly the same number of operations. If, however, \( p \) and \( q \) has significantly different speeds, one of the processes, typically the faster one, may be able to complete many more operations (compared to

\footnote{Note that a \( k \)-DNB object with \( d \) types of operations is incomparable to a \( kd \)-nonblocking object.}
the slower process) than its greater speed actually justifies. For example, suppose \( p \) is twice as fast as \( q \). In this case the object is fair if it allows \( p \) to complete about twice as many operations as \( q \); but it is unfair if it allows \( p \) to do, say, four times as many operations as \( q \).

In general, we say that an object (implementation) is fair if the following holds: when \( m \) processes with speeds \( s_1, s_2, \ldots, s_m \), repeatedly apply operations of the same type, then, for each pair of processes \( i \) and \( j \), the ratio of the number of operations completed by each of the two processes is approximately the ratio of their speeds \( s_i / s_j \). Equivalently, each process \( i \) completes about \( s_i / (s_1 + s_2 + \ldots + s_m) \) of the total number of operations that all the \( m \) processes complete; this fraction is what we call the fair share of process \( i \).

As an example of the difference between achieving wait-freedom and achieving fairness, consider the Single CAS Universal (SCU) class of non-blocking algorithms \(^2\). In systems with a stochastic scheduler, these algorithms were shown to be wait-free with probability 1. This was proven for systems where processes have the same probability of taking the next step (intuitively, they have the same speed) \(^2\), and also for systems where processes have different probabilities of taking the next step (intuitively, they have different speeds) \(^3\). But these algorithms are not fair: even in systems with only two processes, they allow a process \( p \) that is \( x \) times as fast as a process \( q \) to complete about \( x^2 \) times as many operations as \( q \) \(^3\), whereas in a fair implementation \( p \) would complete about \( x \) times as many operations as \( q \).

### 1.5 Some experimental results

To evaluate the practicality of the 2-DNB queue algorithm, we implemented it and the well-known non-blocking queue algorithm by Michael and Scott (MS) \(^2\), and compared their performance under various metrics. The two algorithms differ in their liveness properties: the MS algorithm is non-blocking \(^4\) and thus does not incur the cost of a help mechanism; while our algorithm satisfies the stronger 2-DNB and employs a light-weight help mechanism.

Our experimental results are only preliminary, and they are only meant to give a rough idea on the potential performance of the 2-DNB algorithm compared to a commonly used algorithm. In particular, we did not try to optimize our implementation of the 2-DNB algorithm, and, similarly, we implemented the MS algorithm as written in the original paper without optimizations found in existing implementations.

These experimental results strongly suggest that, besides the stronger progress guarantee of the 2-DNB algorithm, the 2-DNB algorithm drastically increases fairness across a wide range of process speeds, at the cost of a marginal reduction in throughput. As expected, both the MS algorithm and our 2-DNB algorithm are fair when all processes have the same speed, but we observed that: (1) the fairness of the MS algorithm breaks down when processes have different speeds and it deteriorates rapidly as the differences in speeds increase, whereas (2) the 2-DNB algorithm maintains a good level of fairness throughout a wide range of speed differences.

We first considered a system with only two enqueuers and two dequeuers where we slowed one of the two enqueuers and one of the two dequeuers by a factor of \( k \), for each \( k \) in the range \( 2 \leq k < 20 \). With the MS algorithm, we observed that the slower enqueuer and dequeuer completed much less than their fair share of operations: for \( k = 2 \), they got about 45% and 26% of their fair share, respectively; for \( k = 3 \), they got about 23% and 8% of their fair share; for \( k = 5 \), they got about 8% and 1% of their fair share; and for \( k = 8 \), they got only about 2% and 0.22% of their fair share. When we reached \( k = 11 \), the slow dequeuer was prevented from completing any dequeue operation, while the other dequeuer completed about 49,000 operations; and the slow enqueuer managed to complete only 43 operations, while the other enqueuer completed about 59,000 operations. In sharp contrast, with the 2-DNB algorithm, even when the slow enqueuer and dequeuer were slowed down by a factor of \( k = 8 \), they still completed at least 60% of their fair share of operations.

While the fairness of the MS algorithm can be very poor, however, its throughput (i.e., the total number of completed operations) remained consistently higher than our 2-DNB algorithm: for \( k \) ranging from 1 (where all four processes have the same speed) to 19 (where slow processes are 19 times slower than fast ones), the “enqueue” throughput of the 2-DNB algorithm was about 67% to 62% of the throughput of the MS algorithm, while its “dequeue” throughput was about 88% to 76% of the MS algorithm.

Since a 2-DNB queue is, by definition, wait-free for two enqueuers and two dequeuers, one may wonder whether our algorithm’s good fairness behaviour holds up in systems with more processes. To check this, we also run experiments with a system with 8 enqueuers and 8 dequeuers. These experiments confirmed that the algorithm remains fair throughout a wide range of process speed differences. Moreover, its throughput is closer to the throughput of the MS algorithm in this larger system.

It is also worth noting that our experiments confirmed that the 2-DNB algorithm has the following desirable feature: the throughput of the enqueuers depends only on their speed, it does not depend on the speed or
even the presence of dequeuers; symmetrically the throughput of dequeuers is independent of the speed or presence of enqueuers. For example, in one experiment, we considered a system with two groups, 8 enqueuers and 8 dequeuers, where all processes have the same speed. We first experimented with a run where every process in both groups applied operations, and we saw that each enqueuer completed about 12000 operations and each dequeuer completed about 14500 operations. We then experimented with two other runs: one with only the 8 enqueuers present, and one with only the 8 dequeuers present. In both cases, each participating process completed about the same number of operations as in the first run: namely, 12000 for the enqueuers and 14500 for the dequeuers. So the performance of the two groups are indeed independent of each other. As we noted earlier, this is because in our algorithm enqueuers do not interfere with dequeuers, and vice-versa.

The dramatic improvement of fairness exhibited by the 2-DNB algorithm (over the MS algorithm) is not accidental: it is due to the way we designed it to guarantee progress by at least two enqueuers and at least two dequeuers, as we now explain.

1.6 The help mechanism and fairness

At a high level, the help mechanism for enqueue operations works as follows (the help mechanism for dequeue operations is symmetric). There is a single register \( R_E \) that is shared by all enqueuers; \( R_E \) contains at most one call for help by some enqueuer. To execute an enqueue operation, a process \( p \) first tries once to help the process that it sees in \( R_E \) (if any). Whether \( p \) succeeds in this single attempt or not, it then tries to perform its own operation. If this attempt fails, \( p \) overwrites any existing call for help in \( R_E \) with its own call for help; then it tries to perform its own operation again. This attempt to perform its own operation, followed by overwriting \( R_E \) with its call for help if the attempt fails, continues until \( p \) succeeds in completing its operation, whether on its own or with the help of another process which saw \( p \)'s call for help in \( R_E \). Note that with this help mechanism, all the enqueuers that need help effectively compete with each other by (over-)writing their call for help in the same register \( R_E \). We now explain how this helps achieve fairness.

Suppose a process \( i \) is twice as fast as a process \( j \), and that both \( i \) and \( j \) repeatedly fail to perform their enqueue operations, and so they repeatedly call for help via the shared register \( R_E \). Then the periods when \( R_E \) contains the calls for help by \( i \) are approximately twice as long as the periods when \( R_E \) contains the calls for help by \( j \). So whenever a process reads \( R_E \) to see which process to help, it is twice as likely to see \( i \) as it is to see \( j \). Thus, \( i \) will be able to complete about twice as many operations as \( j \), which is fair since \( i \) is twice as fast as \( j \). More generally, the ratio of the periods of times that the calls of help of any two processes \( i \) and \( j \) remain in \( R_E \) is approximately the ratio of their speeds \( s_i/s_j \), so the ratio of the number of operations completed by \( i \) and \( j \) via the 2-DNB help mechanism is about the ratio of their speeds \( s_i/s_j \), as fairness requires.

This light-weight help mechanism to achieve the 2-nonblocking property is not limited to queues. In fact, in Appendix B we give a universal construction for 2-nonblocking objects that essentially employs the same help mechanism: A process first makes a single attempt to help one other process (the "altruistic" phase) and then it repeatedly tries to perform its own operation until it is done (the "selfish" phase). All the processes that need help compete by (over-)writing their call for help on the same shared register. This guarantees the 2-nonblocking property, and it also promotes fairness as explained above.

2 Model sketch

We consider a standard distributed system where asynchronous processes that may fail by crashing communicate via shared registers and other objects, including synchronization objects such as compare\&swap (CAS) \( \#3 \). In such systems, shared objects can be used to implement other shared objects such that: (1) the implemented objects are linearizable \( \#3 \) and (2) they satisfy some liveness requirement. In particular, we consider the \( k \)-nonblocking liveness requirement \( \#3 \ [4] \).

**Definition 1.** \( k \)-nonblocking (\( k \)-NB): if a process invokes an operation and takes infinitely many steps without completing it, then at least \( k \) processes complete infinitely many operations.

We also introduce a liveness requirement, called differentiated \( k \)-nonblocking or \( k \)-DNB for short, that takes into consideration the fact that a shared object may have several operation types (e.g., queues have enqueue and dequeue operations, atomic snapshot have write and scan operations, etc.). The differentiated \( k \)-nonblocking property of an object (or object implementation) is defined as follows:

**Definition 2.** Differentiated \( k \)-nonblocking (\( k \)-DNB): for every operation type \( T \), if a process invokes an operation of type \( T \) and takes infinitely many steps without completing it, then at least \( k \) processes complete infinitely many operations of type \( T \).
3 A 2-DNB algorithm for shared queue

3.1 Description of the algorithm

Data structures. We now describe the data structures used to represent the queue and to implement the helping mechanisms. (The algorithm incorporates two independent helping mechanisms to enforce the 2-nonblocking property, one for enqueuers and one for dequeuers.) The data structures are listed in Figure 1 and illustrated in Figure 1.

The queue consists of a linked list of nodes, containing the elements that have been enqueued but not yet dequeued, as well as the last element dequeued, in the order in which these elements were enqueued. Each node is a record with the following three fields:

- value, the actual element enqueued;
- next, a pointer to the next node in the queue (or NULL, if this node is the last one); and
- flag, a Boolean set to true to indicate that the node has been inserted into the queue, used by the helping mechanism for enqueuers.

The tail of the queue (the end to which elements are added) is identified through a pointer, called $g_Tail$, that points to the last node in the queue, unless a node is in the process of being added to the queue, in which case $g_Tail$ could be temporarily pointing to the penultimate node in the queue.

The head of the queue (the end from which elements are removed) is identified through a record, called $g_Head$, that contains the following information:

- ptr, a pointer to the last node that was dequeued. The queue is empty if and only if $g_Head.ptr$ and $g_Tail$ point to the same node. If the queue is not empty, the first element in the queue — i.e., the next element to be dequeued — is in the node pointed to by $g_Head.ptr \to next$.

Fig. 1 Shared objects for the 2-DNB queue algorithm

Structures:

```c
structure NODE {
  value: a register containing an integer.
  next: a compare&swap containing a pointer to a node.
  flag: a register containing a boolean. }
```

Shared Objects:

- $g_{Init\_Node}$: A node, initially with {value = arbitrary, next = NULL, and flag = 1}.
- $g_{Init\_DQ}$: A location containing an arbitrary non-NULL value.
- $g_Tail$: a compare&swap containing a pointer to a node, initially pointing to $g_{Init\_Node}$.
- $g_Head$: a compare&swap with fields:
  - ptr: a pointer to a node, initially pointing to $g_{Init\_Node}$.
  - value: an integer, initially an arbitrary non-NULL value.
  - addr: a pointer to a location that stores a dequeued value or NULL, initially pointing to $g_{Init\_DQ}$.
- $g_{Ann\_E}$: a register containing a pointer to a node, initially $g_{Init\_Node}$.
- $g_{Ann\_D}$: a register containing a pointer to a location that stores a dequeued value or NULL, initially pointing to $g_{Init\_DQ}$.
• Two fields with information pertaining to the last dequeue operation, used by the helping mechanism for dequeuers:
  – `value`, the last element dequeued. This is the same as `g.Head.ptr → value`, except if the queue was empty when the last element was dequeued, in which case it is ⊥.
  – `addr`, a pointer to a location reserved by the process that dequeued the last element; this location contains the element dequeued by that process or NULL.

Initially, the linked list representing the queue contains a single node `g.Init.Node`, with both `g.Tail` and `g.Head.ptr` pointing to it. The `value` field of `g.Init.Node` is arbitrary, `next` is NULL, and `flag` is 1. (This node can be thought of as representing a fictitious element that was enqueued and then dequeued before the algorithm starts.) The remaining two fields of `g.Head`, `value` and `addr`, are initialized to a non-NULL value and a pointer to `g.Init.DQ` (a location that contains an arbitrary non-NUL value).

To implement the helping mechanisms among enqueuers and among dequeuers, the algorithm uses two shared registers:
• `g.Ann.E` contains a pointer to a node that some process wishes to add to the linked list of nodes representing the queue (initially a pointer to `g.Init.Node` whose `flag` field is 1, indicating that no help is needed); and
• `g.Ann.D` contains a pointer to the location reserved by a process `p` that wishes to perform a dequeue operation; it is intended for a helper to store the value dequeued for `p` (initially a pointer to `g.Init.DQ`, a location that contains a non-NUL value, indicating that no help is needed).

The enqueue operation. Consider a process `p` that wishes to enqueue an element `v` (see procedure `enqueue(v)`, lines 1-12). Process `p` must add a node containing `v` to the tail end of the linked list of nodes representing the queue.

Roughly speaking, `enqueue(v)` consists of two phases: In the altruistic phase (lines 2-3) `p` tries once to help some process that has asked for help to enqueue an element. Whether it succeeds or fails, it then proceeds to the selfish phase (lines 8-10), where `p` keeps trying to enqueue its own element (a node it has prepared in lines 4-7 to append to the linked list) until it succeeds to do so. Each time it tries and fails, `p` asks for help by writing into the shared variable `g.Ann.E` a pointer to the node it wants to add to the linked list. It is possible that the enqueue operation does not terminate because a process is stuck forever.

---

**Algorithm 1 The 2-DNB queue algorithm (code for process `p`)**

```plaintext
1: procedure enqueue(v)  { v ∈ Z }
2: l.Ann.E := g.Ann.E.read()
3: trytoenqueue(l.Ann.E)
4: l.Node := a new node
5: l.Node → value.write(v)
6: l.Node → next.write(NULL)
7: l.Node → flag.write(0)
8: while trytoenqueue(l.Node) = failed do
9:   g.Ann.E.write(l.Node)
10: end while
11: return done
12: end procedure

13: procedure dequeue()
14: l.Ann.D := g.Ann.D.read()
15: if *l.Ann.D.read() = NULL then
16:   trytodequeue(l.Ann.D)
17: end if
18: l.addr := a pointer to a new location
19: *l.addr.write(NULL)
20: repeat
21:   t := trytodequeue(l.addr)
22:   if t = failed then
23:     g.Ann.D := l.addr
24:   end if
25: until t ≠ failed
26: return t
27: end procedure
```
in the selfish phase. As we will show, however, this can happen only if at least two other processes succeed in enqueuing elements infinitely often. Note that an enqueuer never helps a dequeuer.

Both p’s attempt to help another enqueuer and its attempt to enqueue its own element are carried out by the procedure \textsc{trytoenqueue}(r), discussed below, where \( r \) is a pointer to the node to be appended to the list. \textsc{trytoenqueue}(r) returns \textit{done} if it succeeds to append to the list the node pointed to by \( r \); otherwise it returns \textit{failed}. The latter happens only if some other concurrent enqueue operation succeeds in appending a node to the list.

We now describe the procedure \textsc{trytoenqueue}(r) (lines 28–51). The procedure first creates a local copy \( _\text{next} \) of the tail pointer \( g._\text{Tail} \), and a local copy \( _\text{next} \) of \( g._\text{Tail} \rightarrow \text{next} \) (lines 29–30). Ordinarily \( _\text{next} = \text{NULL} \), unless some call to \textsc{trytoenqueue} has appended a node to the linked list but has not (yet) updated the \( g._\text{Tail} \) pointer.

\textbf{Algorithm 1} The \textsc{trytoenqueue} and \textsc{trytodequeue} procedures

\begin{verbatim}
28: procedure \textsc{trytoenqueue}(_\text{Node})
29:   _\text{Tail} := g._\text{Tail}.read()
30:   _\text{next} := g._\text{Tail} \rightarrow \text{next}.read()
31: if _\text{Node} \rightarrow \text{flag}.read() = 1 then
32:   _\text{Tail} := g._\text{Tail}.read()
33:   _\text{next} := g._\text{Tail} \rightarrow \text{next}.read()
34:   if _\text{next} \neq \text{NULL} then
35:     _\text{next} \rightarrow \text{flag}.write(1)
36:     g._\text{Tail}.CAS(_\text{Tail}, _\text{next})
37:   else
38:     return done
39:   end if
40: if _\text{next} \neq \text{NULL} then
41:   _\text{next} \rightarrow \text{flag}.write(1)
42:   g._\text{Tail}.CAS(_\text{Tail}, _\text{next})
43: else
44:   _\text{next} \rightarrow \text{next}.CAS(\text{NULL}, _\text{Node})
45:   _\text{Node} \rightarrow \text{flag}.write(1)
46:   g._\text{Tail}.CAS(_\text{Tail}, _\text{Node})
47:   return done
48: end if
49: end if
50: return failed
51: end procedure
52: procedure \textsc{trytodequeue}(\text{Laddr})
53:   _\text{Head} := g._\text{Head}.read()
54:   _\text{Tail} := g._\text{Tail}.read()
55: "\text{Laddr}.write(_\text{Head}.value)
56: if "\text{Laddr}.read() \neq \text{NULL} then
57:   return "\text{Laddr}.read()
58: end if
59: if _\text{Head}.ptr = _\text{Tail} then
60:   if g._\text{Head}.CAS(_\text{Head}, (_\text{Head}.ptr, \perp, \text{Laddr})) then
61:     return \perp
62: end if
63: else
64:   _\text{next} := g._\text{Head}.\text{ptr} \rightarrow \text{next}.read()
65:   \text{v} := g._\text{next} \rightarrow \text{value}.read()
66: if g._\text{Head}.CAS(_\text{Head}, (_\text{next}, \text{v}, \text{Laddr})) then
67:   return \text{v}
68: end if
69: end if
70: return failed
71: end procedure
\end{verbatim}
TRYTOENQUEUE(r) checks whether the node to which r points has flag = 1, indicating that the node has already been threaded to the list (line 31). If so, TRYTOENQUEUE refreshes lTail and lnext (lines 32–33). It then ensures that the node pointed to by lnext, if any, is fully incorporated into the data structures: its flag is set to 1 and g_Tail points to it, and returns done (lines 34–35).

If the node to which r points does not have flag = 1, then the TRYTOENQUEUE(r) procedure tests whether lnext = NULL (line 40). If not, a concurrent ENQUEUE operation succeeded in appending to its node to the list. So TRYTOENQUEUE(r) ensures that the appended node is fully incorporated into the data structures: its flag is set to 1 and g_Tail points to it (lines 11–12); and returns failed (line 50). If, on the other hand, TRYTOENQUEUE(r) found that lnext = NULL, it tries to append to the list the node pointed to by r by applying a CAS(NULL, r) operation on lTail → next (line 14). If this CAS is successful, TRYTOENQUEUE incorporates into the list the new node pointed to by r by setting its flag to 1 and ensuring that g_Tail points to it, and then returns done (lines 15–17). If the CAS is unsuccessful, then another enqueue operation succeeded in appending to the linked list a node other than r. Thus, TRYTOENQUEUE returns failed (line 50).

The dequeue operation. Consider a process p that wishes to perform a dequeue operation (see procedure DEQUEUE, lines 13–27). Process p must return the first element in the queue (or ⊥, if the queue is empty) after updating the information in g_Head to reflect the removal of that element.

Similar to ENQUEUE, the dequeue operation consists of an altruistic and a selfish phase: In the altruistic phase (lines 14–17) p tries once to help some process that has asked for help to dequeue an element. Whether it succeeds or fails, p then proceeds to the selfish phase (lines 18–25), where it keeps trying to dequeue an element for itself until it succeeds to do so; each time it tries and fails, p asks for help by writing into the shared variable g_Ann.D a pointer to a location it has reserved (line 18), in which the helper process will place the dequeued element for p to retrieve. This location is initialized to the value NULL (line 19), which indicates that p (the process requesting help) has not yet received help; when this location contains a non-NUL value, p’s dequeue operation has been helped, and p can retrieve the dequeued element from that location. So, the altruistic loop starts with p making a local copy lAnn.D of the g_Ann.D shared variable used to ask for help (line 14). If the location to which lAnn.D points contains NULL, p tries to help a dequeuer (lines 15–17).

It is possible that the dequeue operation does not terminate: A process may be stuck forever in the selfish phase. As we will show this can happen only if at least two other processes succeed in dequeuing elements infinitely often. Note that a dequeuer never helps an enquirer.

Both p’s attempt to help another dequeuer and its attempt to dequeue its own element are carried out by the procedure TRYTODEQUEUE(r), discussed below, where r is a pointer to a location reserved by the process on behalf of which this attempt to dequeue is being made. If it is successful, TRYTODEQUEUE(r) returns the element dequeued (or the special value ⊥, if the queue is empty); otherwise it returns failed. The latter happens only if some other concurrent dequeue operation succeeds in dequeuing an element.

We now describe the procedure TRYTODEQUEUE(lAddr) (lines 52–71). This procedure first makes a local copy lHead of the g_Head record and a local copy lTail of the g_Tail pointer (lines 53–54). Recall that the fields value and addr of g_Head contain information about the last dequeue operation performed. As this information is about to be over-written, and the value dequeued by that operation (namely, g_Head.value) may be needed to help the process that issued that dequeue operation (namely, the process that reserved the location pointed to by g_Head.addr), the dequeued element is copied to that location (line 55).

Once the information in g_Head has been safely stored, TRYTODEQUEUE(lAddr) examines the value in the location pointed to by lAddr. If this is non-NUL, the element dequeued for this operation (or ⊥, if the queue was empty) has already been stored there by a helper. Thus, in this case, TRYTODEQUEUE(lAddr) merely returns the content of the location pointed to by lAddr (lines 56–58). Otherwise, TRYTODEQUEUE(lAddr) tries to dequeue an element by attempting to update g_Head. This is done by means of a CAS operation that changes g_Head if it has not been changed since TRYTODEQUEUE(lAddr) made a local copy of that variable in lHead. The new information written into g_Head depends on whether TRYTODEQUEUE(lAddr) found the queue to be empty (lines 59–62) or not (lines 63–69). If the CAS operation succeeds, TRYTODEQUEUE(lAddr) returns the dequeued element (or ⊥, if the queue is empty). If the CAS operation fails, then some concurrent dequeuer succeeded in effecting its dequeue operation, and this call to TRYTODEQUEUE returns failed (line 70).

---

5Pointer g_Tail is updated using a CAS operation to ensure that a late application does not over-write an earlier update.

6The process on behalf of which p’s call to TRYTODEQUEUE(r) is trying to dequeue an element is a process other than p, if this call is made in the altruistic phase; or p itself, if this call is made in the selfish phase.
3.2 Proof of linearizability

Let $H$ be any history of the algorithm. Then we construct a completion $H'$ of $H$ as follows:

- For each incomplete ENQUEUE operation in $H$, if it allocates a new node on line 4 and this node has been pointed to by the $g_{Tail}$ pointer, then it is completed in $H'$ by returning done; otherwise, it is removed from $H'$.

- For each incomplete DEQUEUE operation in $H$, if it allocates a new location on line 18 and this location has been pointed to by $g_{Head}.addr$, then it is completed in $H'$ by returning the value of $g_{Head.value}$ at the earliest time when $g_{Head.addr}$ points to the new location it allocated; otherwise, it is removed from $H'$.

We then construct a linearization $L$ of $H'$ as follows:

- Each ENQUEUE operation in $H'$ is linearized when $g_{Tail}$ first points to the new node it allocates on line 4.

- Each DEQUEUE operation is linearized when $g_{Head.addr}$ first points to the new location it allocates in line 18 if the return value of the DEQUEUE is not $\perp$; otherwise it is linearized when the first process that sets $g_{Head.addr}$ to point to the new location last reads $g_{Tail}$ in line 54.

The proof that this linearization is correct with respect to the specification of a queue can be found in Appendix A.

3.3 Proof of liveness: the 2-DNB property

**Definition 3.** A shared queue is called non-blocking for ENQUEUE operations, if it satisfies the following property: if a process invokes an ENQUEUE operation and takes infinitely many steps without completing it, then there must be another process completing infinitely many ENQUEUE operations. Nonblocking for DEQUEUE operations is defined similarly.

**Definition 4.** A shared queue is called 2-nonblocking for ENQUEUE operations, if it satisfies the following property: if a process invokes an ENQUEUE operation and takes infinitely many steps without completing it, then there must be at least two other processes completing infinitely many ENQUEUE operations. 2-nonblocking for DEQUEUE operations is defined similarly.

We first show that the algorithm is non-blocking, and then use this to prove that it is also 2-nonblocking.

**Theorem 5.** The algorithm is non-blocking for both ENQUEUE and DEQUEUE operations.

**Proof.** First consider ENQUEUE operations. Suppose, for contradiction, that some process $p$ takes infinitely many steps for an ENQUEUE operation but no other process completes an infinite number of ENQUEUE operations. Thus, eventually $p$ makes continuously calls to TRYTOENQUEUE, all of which return failed; and every other process ceases to execute any operations, or executes infinitely many DEQUEUE operations, or is stuck in an ENQUEUE or DEQUEUE operation forever. Note that TRYTOENQUEUE returns failed either because the condition in line 10 is true, or because the CAS in line 44 fails. We now prove that each of these can happen only a finite number of times, contradicting that $p$ is stuck forever in an ENQUEUE operation.

**Case 1.** Process $p$ finds the condition in line 10 to be true infinitely many times. Whenever $p$ finds the condition in line 10 to be true it performs a CAS on $g_{Tail}$ to move it forward through the list of nodes, and this CAS fails only if another process has already moved the tail forward. This means that infinitely many nodes are added to the list, i.e., infinitely many ENQUEUE operations complete, contrary to our supposition.

**Case 2.** The CAS in line 44 fails infinitely many times. Whenever this CAS fails, some node is added to the list. This implies that an infinite number of ENQUEUE operations complete, contrary to our supposition.

Next we prove that the algorithm is also non-blocking for DEQUEUE operations. By a similar argument, eventually some process $p$ makes continuously calls to TRYTODEQUEUE, all of which return failed; and every other process ceases to execute any operations, or executes infinitely many ENQUEUE operations, or is stuck in an ENQUEUE or DEQUEUE operation forever. Note that a call to TRYTODEQUEUE returns failed only if the CAS is line 60 or 66 fails. Thus, $p$ performs an infinite number of unsuccessful CAS operations on $g_{Head}$. This means that there are infinitely many successful CAS operations on $g_{Head}$. Therefore, an infinite number of DEQUEUE operations complete, contrary to our assumption.

\[\square\]
Theorem 6. The algorithm is 2-nonblocking for ENQUEUE operations.

Proof. Suppose, for contradiction, that the algorithm is not 2-nonblocking for ENQUEUE operations. Thus there is an execution where a process takes infinitely many steps but is stuck in an ENQUEUE operation forever, and during which fewer than two processes complete infinitely many ENQUEUE operations. Since we have proved the algorithm is non-blocking for ENQUEUE operations, the only possibility is that one process \( p \) completes infinitely many ENQUEUE operations. Since other processes complete ENQUEUE operations for only finitely many times, eventually (from some time \( T \) onwards) only \( p \) can complete ENQUEUE operations. We should notice that our algorithm has a help mechanism: by *only* \( p \) can complete operations we mean that only the requests of \( p \) are fulfilled. This does not necessarily means that process \( p \) wins every competition of setting \( g_{\text{Tail}.next} \). Other process can also win that competition, but they can only succeed when they are helping \( p \). When process \( p \) fails one such competition, it will put its node in \( g_{\text{Ann}.E} \). We show this can only happen finitely many times.

Claim 6.1 After time \( T \), process \( p \) can fail on line 40 or 44 for at most \( 2n \) times.

Proof of Claim 6.1 If process \( p \) fails one such competition or finds that tail is not pointing to the last node, then at least one other process has succeeded in linking one node to the queue. However, we have assumed that after time \( T \), only process \( p \) can return from an ENQUEUE operation. Every other process either gets stuck in a loop forever, or is able to carry on DEQUEUE operation but never enqueues. They can succeed the CAS of line 44 only when helping process \( p \), or when helping itself but crashing before returning. Either case only happens at most once for one process, so \( p \) can fail no more than \( 2n \) times.

Claim 6.2 Eventually the value in register \( g_{\text{Ann}.E} \) is never \( p \).

Proof of Claim 6.2 From Claim 6.1 we know that process \( p \) only sets \( g_{\text{Ann}.E} \) to its node(s) for finitely many times, because setting \( g_{\text{Ann}.E} \) to a node of \( p \) means \( p \) failed one competition on line 40 or 44 which happens no more than \( 2n \) times. However, by our assumption, some other process is taking infinitely many steps in an ENQUEUE operation and never returns. Thus it will set \( g_{\text{Ann}.E} \) to its node infinitely many times. As a result, eventually the register \( g_{\text{Ann}.E} \) points to a node that does not belong to \( p \) and is never rewritten by \( p \).

Now return to the proof of Theorem 6. By Claim 6.2 eventually \( g_{\text{Ann}.E} \) never points to a node of \( p \). Since process \( p \) completes infinitely many ENQUEUE operations, after this time it will invoke an ENQUEUE operation and see \( g_{\text{Ann}.E} \) contains a node from another process. It will then work for that process on line 6. By Claim 6.1 eventually process \( p \) never fails the competitions on line 40 or 44, so it succeeds in helping one other process. Eventually the node in register \( g_{\text{Ann}.E} \) is from a process that takes infinitely many steps, (because if a node is written only finitely many times, it will eventually be covered by those writing infinitely many times), and that process will complete its ENQUEUE operation after being helped by process \( p \), a contradiction. In conclusion, the assumption that only process \( p \) can complete operations after time \( T \) is wrong, so the algorithm is 2-nonblocking for the ENQUEUE function.

Theorem 7. The algorithm is 2-nonblocking for the DEQUEUE operations.

Proof. The proof is similar to that of Theorem 6. Eventually (from some time \( T \) onwards) only a process \( p \) can complete infinitely many DEQUEUE operations, and every other process is either stuck in a loop, or never dequeues. After time \( T \), process \( p \) can fail the CAS operations on lines 50 or 54 or at most \( 2n \) times. This is because that when process \( p \) fails one such CAS operation, at least one other process succeeds to dequeue a node (or \( \perp \)). These events can happen at most twice for one process, so the total number of times is no more than \( 2n \). Eventually \( g_{\text{Ann}.D} \) never stores a pointer allocated by \( p \), because \( p \) never writes into it, whereas some other process writes into it infinitely many times. As a result, process \( p \) works for that process on line 16 and successfully dequeues a node (or \( \perp \)) for it.

From Theorems 6 and 7 we have:

Corollary 8. The algorithm is 2-DNB.
4 Experimental results

4.1 Experimental setting

We implemented two shared queue algorithms: our 2-DNB algorithm (Algorithm 1), and the non-blocking MS algorithm as described in [9]. The two algorithms were implemented “as is”, without any optimization: our goal was to get a rough idea of the 2-DNB algorithm’s potential compared to a commonly used non-blocking algorithm. In particular, we wanted to explore the tradeoff between the cost and benefits of the non-blocking property of the MS algorithm (which does not need any help mechanism) compared to the stronger 2+2 non-blocking property of 2-DNB (which uses a light-weight help mechanism).

We implemented both algorithms in Java, and executed them on an Intel i7-9750H CPU with 6 cores (12 threads) and 16 GBs of RAM. These are only preliminary results as we intend to re-run more extensive experiments on a dedicated multiprocessor cluster in the future. The experimental results that we obtained so far indicate that, compared to the MS algorithm, the fairness of the 2-DNB algorithm is much higher across a wide range of process speeds, albeit at the cost of a marginal reduction in throughput.

To evaluate the fairness of the two algorithms under various process speeds, we had to control the speed of individual processes. To control the speed of a process \( p \), we added a delay immediately after every shared memory access (read, write or CAS) by \( p \). In our experiments, these delays follow an exponential distribution \( \text{Exp}(\mu) \), where \( \mu \) is the average delay. So the number of shared-memory steps that \( p \) executes by some time \( t \) follows a Poisson distribution with parameter \( \lambda = 1/\mu \), and the simulated speed of a process \( p \) is (proportional) to \( 1/\mu \). Thus, in our experiments we controlled the speed of each process by setting its corresponding average delay \( \mu \).

4.2 Experiments with two enqueuers and two dequeuers

One of our experiments considered a system with only two enqueuers and two dequeuers where we slowed one of the two enqueuers and one of the two dequeuers by a factor of \( k \), for each \( k \) in the range \( 2 \leq k < 20 \). To do so, we set the delay parameter \( \mu \) of the slower enqueuer [dequeuer] to be \( k \) times as large as the delay parameter \( \mu \) of the faster enqueuer [dequeuer].

With the MS algorithm, we observed that the slower enqueuer and the slower dequeuer completed much less than their fair shares of operations as \( k \) increased (see Figures 2 and 3). And when we reached \( k = 11 \), the slow dequeuer was not able to complete any dequeue operation, while the other dequeuer completed about 49,000 operations; and the slow enqueuer completed only 43 operations, while the other enqueuer completed almost 59,000 operations.

In contrast, the fairness of the 2-DNB algorithm remained reasonable (55% or more) for both enqueuers and dequeuers, and for all the values of \( k \) in the range \( 2 \leq k < 20 \).

The throughput of the 2-DNB algorithm, however, was lower than that of the MS algorithm: for \( k \) in the range \( 2 \leq k < 20 \), the enqueuers’ throughput of the 2-DNB algorithm was about 67% to 62% of the throughput.

---

7We did not add delays after local steps, e.g., after reading a local variable, because the time to perform a local step is negligible compared to the time it takes to access shared memory.

8This is also because the smallest delay \( \mu \) that we chose in our experiments, 1ms, is orders of magnitude greater than the time that a process actually takes to execute any line of code.
of the MS algorithm, while the dequeuers’ throughput was about 88% to 76% of the MS algorithm.

4.3 Experiments with eight enqueuers and eight dequeuers

The experimental results that we observed for a system with two enqueuers and two dequeuers were very encouraging, but since a 2-DNB queue is, by definition, wait-free for this special case, one may wonder whether the good fairness behaviour of our algorithm would continue to hold in systems with more processes. To check this, we considered a system with 8 enqueuers and 8 dequeuers.

As a baseline, we first executed the two algorithms in a setting $S_0$ where all 16 processes have exactly the same speed (every process has $k = 1$). These runs confirmed that, as expected, both algorithms are indeed fair in this case: every process got very close to 100% of its fair share.

We then considered what happens when, in each group, processes have different speeds. To do so, we experimented with two different settings:

- a setting $S_1$ where the differences in speeds are relatively small; specifically, process $i$, for $1 \leq i \leq 8$, is slowed down by a factor of $i$ (e.g., the speeds of processes 2, 3 and 4, are 1/2, 1/3 and 1/4 of the speed of process 1, respectively).

- a setting $S_2$ where the differences in speeds are large; specifically, process $i$, for $1 \leq i \leq 8$, is slowed down by a factor of $2^{i-1}$ (e.g., the speeds of processes 2, 3 and 4, are 1/2, 1/4 and 1/8 of the speed of process 1, respectively).

In each of the two groups (of 8 enqueuers and 8 dequeuers) each process completed about 12.5% of the total number of operations completed by its group.
In both settings, we observed that the fairness of the 2-DNB algorithm is even better than in the system with two enqueuers and two dequeuers: in $S_1$, even the enqueuer and dequeuer that were 8 times slower than the fastest enqueuer and dequeuer managed to complete 67% and 77% of their fair shares of operations, respectively (Figures 4 and 5); in $S_2$, even the enqueuer and dequeuer that were 128 times slower (!) than the fastest enqueuer and dequeuer completed 65% and 76% of their fair shares of operations, respectively (Figures 6 and 7).

In sharp contrast, we saw that the fairness of MS algorithm declines rapidly when the differences in speeds increase. As shown in Figures 4 and 5, enqueuers with a slowdown factor of 2, 3, 5 and 8 completed only 74%, 33%, 11% and 3% of their fair shares of enqueue operations, respectively; and dequeuers with a slowdown factor of 2, 3, 5 and 8 completed only 45%, 13%, 3% and 0% of their fair shares of dequeue operations, respectively. As shown in Figures 6 and 7, enqueuers (dequeuers) with a slowdown factor of 2, 4, 8 and 16 completed only 59%, 15%, 3% and 0% (35%, 4%, 0% and 0%) of their fair shares of enqueue (dequeue) operations, respectively.

We also compared the throughput of the two algorithms, in terms of the total number of completed operations. As we can see in Table 1 the throughput of the MS algorithm was marginally higher, but by less than in the experiment with two enqueuers and two dequeuers: In setting $S_0$, the 16 processes completed a total of 223089 and 206605 operations with the MS algorithm and the 2-DNB algorithm, respectively; so the throughput of the 2-DNB algorithm was 92.6% of the MS algorithm. In setting $S_1$, they completed a total of 112147 and 100490 operations with the MS algorithm and the 2-DNB algorithm, respectively. In setting $S_2$, they completed a total of 111710 and 93357 operations with the MS algorithm and the 2-DNB algorithm, respectively. So in $S_1$ and $S_2$, the throughput of the 2-DNB algorithm was 89.6% and 83.6% of the MS algorithm, respectively.

| Setting | MS NQ throughput | MS DQ throughput | 2-DNB NQ throughput | 2-DNB DQ throughput | Total | 2-DNB/MS |
|---------|------------------|------------------|---------------------|---------------------|-------|-----------|
| $S_0$   | 128167           | 94922            | 93177               | 113428              | 220309| 92.61%    |
| $S_1$   | 61522            | 50625            | 44445               | 56045               | 100690| 89.61%    |
| $S_2$   | 61003            | 50707            | 42856               | 50501               | 93357 | 83.57%    |

Table 1: Throughputs of the MS and 2-DNB algorithms in settings $S_0$, $S_1$, $S_2$

References

[1] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Mike Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, 1993.

[2] Dan Alistarh, Keren Censor-Hillel, and Nir Shavit. Are lock-free concurrent algorithms practically wait-free? *Journal of the ACM (JACM)*, 63(4):1–20, 2016.

[3] Dan Alistarh, Thomas Sauerwald, and Milan Vojnović. Lock-free algorithms under stochastic schedulers. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 251–260, 2015.

[4] Naama Ben-David, David Yu Cheng Chan, Vassos Hadzilacos, and Sam Toueg. k-abortable objects: Progress under high contention. In Cyril Gavoille and David Ilcinkas, editors, *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*, volume 9888 of *Lecture Notes in Computer Science*, pages 298–312. Springer, 2016.

[5] Victor Bushkov and Rachid Guerraoui. Safety-liveness exclusion in distributed computing. In *Proceedings of the twenty fourth annual ACM symposium on Principles of Distributed Computing*. ACM, 2015.

10Dequeuer 8 managed to complete only 5 dequeue operations, while the group of dequeuers completed about 50000 dequeues.
A Proof of linearizability of the 2-DNB queue algorithm

Recall that $H$ is any history of the algorithm, and we constructed a completion $H'$ of $H$ as follows:

- For each incomplete ENQUEUE operation in $H$, if it allocates a new node on line 4 and this node has been pointed to by the $g.Tail$ pointer, then it is completed in $H'$ by returning done; otherwise, it is removed from $H'$.

- For each incomplete DEQUEUE operation in $H$, if it allocates a new location on line 18 and this location has been pointed to by $g.Head.addr$, then it is completed in $H'$ by returning the value of $g.Head.value$ at the earliest time when $g.Head.addr$ points to the new location it allocated; otherwise, it is removed from $H'$.

We then constructed a linearization $L$ of $H'$ as follows:

- Each ENQUEUE operation in $H'$ is linearized when $g.Tail$ first points to the new node it allocates on line 4.

- Each DEQUEUE operation is linearized when $g.Head.addr$ first points to the new location it allocates in line 18 if the return value of the DEQUEUE is not ⊥; otherwise it is linearized when the first process that sets $g.Head.addr$ to point to the new location last reads $g.Tail$ in line 54.

Note that by this construction, no operation can be linearized before it is invoked. This is easy to see for ENQUEUE operations and DEQUEUE operations that do not return ⊥, because these operations are linearized when their newly allocated node or location is pointed to by $g.Tail$ or $g.Head.addr$. For each DEQUEUE operation that returns ⊥, the process that first sets $g.Head.addr$ to point to the new location on line 60 must have previously read a pointer to that new location on line 14 before executing line 54, which is the linearization point of the operation. Clearly, a pointer to the new location cannot be created before the location is allocated, so the operation is linearized after it is invoked.

Thus we observe that:

Observation 9. The linearization points of the linearization $L$ of $H'$ are well defined and within their execution intervals if the following properties hold:

- For each ENQUEUE operation, a pointer to the new node it allocates on line 4 can be written into $g.Tail$ at most once, and has occurred exactly once when the operation completes.

- For each DEQUEUE operation, a pointer to the new location it allocates on line 18 can be written into $g.Head$ at most once, and has occurred exactly once when the operation completes.

Lemma 10. The following properties always hold:

1. The linked list starting from $g.Init.Node$ is never disconnected.

2. $g.Head.ptr$ always points to a node in the linked list starting from $g.Init.Node$. Furthermore, if $g.Head.ptr$ is changed from pointing to a node $A$ to pointing to a node $B$, then the next pointer of $A$ points to $B$.

3. $g.Tail$ always points to a node in the linked list starting from $g.Init.Node$ that is not before the node pointed to by $g.Head.ptr$. Furthermore, if $g.Tail$ is changed from pointing to a node $A$ to pointing to a node $B$, then the next pointer of $A$ points to $B$. 


4. Whenever a node $A$ is linked to a node $B$ (in the sense that the next pointer of $A$ is set to point to $B$), $A$ is the last node in the linked list starting from $g_{\text{Init.Node}}$ and $g_{\text{Tail}}$ points to $A$.

Proof. First observe that all these properties hold when the queue is initialized. From this, we show that each property must always hold as follows:

1. Since no step of the algorithm can change the next pointer of a node from a non-NULL value, links between nodes can never be removed. Thus the linked list starting from $g_{\text{Init.Node}}$ can never be disconnected, i.e., Property 1 always holds.

2. $g_{\text{Head.ptr}}$ can only be changed by a successful CAS on line 66. (Note that line 66 cannot change $g_{\text{Head.ptr}}$.) If the CAS in line 66 is successful, then since line 64 is executed before line 66 observe that if $A$ is the node that $g_{\text{Head.ptr}}$ originally pointed to and $B$ is the node that $g_{\text{Head.ptr}}$ points to afterward, then the next pointer of $A$ points to $B$ (because Property 1 always holds). Thus $g_{\text{Head.ptr}}$ is simply changed from one node to the linked list starting from $g_{\text{Init.Node}}$. So Property 2 always holds.

3. The value of $g_{\text{Tail}}$ can only be changed by a successful CAS on one of lines 36, 42 and 46. If the CAS is successful, then lines 33, 39 and 41 are executed before lines 35, 42 and 46, respectively, observe that if $A$ is the node that $g_{\text{Tail}}$ originally pointed to and $B$ is the node that $g_{\text{Tail}}$ points to afterward, then the next pointer of $A$ points to $B$ (because Property 1 always holds). Thus $g_{\text{Tail}}$ still points to a node in the linked list starting from $g_{\text{Init.Node}}$. Moreover, $g_{\text{Tail}}$ never lags behind $g_{\text{Head.ptr}}$, because line 66, the only line that moves $g_{\text{Head.ptr}}$ to the next node, is only executed when $l_{\text{Head.ptr}} \neq l_{\text{Tail}}$ (line 59), where \( l_{\text{Head}} = g_{\text{Head}} \) (line 53) and \( l_{\text{Tail}} = g_{\text{Tail}} \) (line 54). If $g_{\text{Tail}}$ ever lags behind $g_{\text{Head.ptr}}$, then some operation must have moved $g_{\text{Head.ptr}}$ one step forward when $l_{\text{Head.ptr}} = l_{\text{Tail}}$, which never happens. So Property 3 always holds.

4. The only step of the algorithm that can link two nodes is a successful CAS on \( l_{\text{Tail}} \to \text{next} \) on line 43. We need to show this \( l_{\text{Tail}} \) is pointing to the last node in the linked list starting from $g_{\text{Init.Node}}$. Since Property 3 always holds and \( l_{\text{Tail}} \) was set to $g_{\text{Tail}}$ on line 29, \( l_{\text{Tail}} \) points to a node in the linked list starting from $g_{\text{Init.Node}}$. Since the CAS succeeds, the next pointer of this node is NULL. So since Property 1 always holds, this node that is pointed to by \( l_{\text{Tail}} \) must be the last node of the linked list starting from $g_{\text{Init.Node}}$. Furthermore, since Property 3 always holds, this node is also still pointed to by $g_{\text{Tail}}$. Thus Property 4 always holds.

Definition 11. The state of a 2-nonblocking queue is defined to be the linked list of nodes between the nodes pointed to by $g_{\text{Head.ptr}}$ and $g_{\text{Tail}}$, excluding the node pointed to by $g_{\text{Head.ptr}}$.

By Lemma 10 the state of a queue is always well defined, because $g_{\text{Head.ptr}}$ and $g_{\text{Tail}}$ both point to nodes in the linked list of nodes starting from $g_{\text{Init.Node}}$ and $g_{\text{Tail}}$ always points to a node that is not before the node pointed to by $g_{\text{Head.ptr}}$.

Lemma 12. The following two properties always hold:

(a) $g_{\text{Tail}} \to \text{flag} = 1$.

(b) Whenever a node is added to the list, it has \text{flag} = 0.

Proof. For (a) we note that $g_{\text{Tail}} \to \text{flag}$ is initially 1. Furthermore, $g_{\text{Tail}}$ is changed only on lines 36, 42 and 46. In each case, the preceding line has set to 1 the \text{flag} field of the node to which $g_{\text{Tail}}$ will point. Finally, observe that after the \text{flag} field of this node is set to 1, it can never be changed by any step of the algorithm. Thus (a) always holds.

For (b) we note that the only place where a node is added to the list is on line 44. Suppose, for contradiction, that a process $p$ adds to the list a node $r$ with \text{flag} = 1 in some execution of line 44, say at time $t_1$. The \text{flag} field of a node is set to 1 only in lines 35, 41 and 45 and in all these cases the node has already been added to the list. Therefore, some process $p'$ added $r$ to the list before time $t_1$, say at time $t_2$. Let $t_0$ be the last time before $t_1$ when $p$ executed line 29. If $t_2$ is before $t_0$, then there are two cases: either $r \to \text{flag}$ is set to 1 before $t_0$, or not. If $r \to \text{flag}$ is set to 1 before $t_0$, then the process $p$ that would add $r$ to the list at time $t_1$ would, after time $t_0$, find that $r \to \text{flag} = 1$ on line 31 and so not execute line 44 at time $t_1$, contradicting that $p$ adds $r$ to the list at that time. Otherwise, since $r \to \text{flag}$ was not set to 1 before $t_0$ and we have already proven that (a) always holds, $g_{\text{Tail}}$ is also not moved to the added node $r$ before $t_0$ either. So the value of $g_{\text{Tail}} \to \text{next}$ that $p$ read after $t_0$ is not NULL, but rather a pointer to the node $r$. But then, by line 40 $p$ does not execute line 44 at time $t_1$, contradicting that $p$ adds $r$ to the list at that time.

If $t_2$ is between $t_0$ and $t_1$, then there are two cases. In case 1, $p$ finds that the value of $g_{\text{Tail}} \to \text{next}$ that $p$ read after $t_0$ is not NULL, but rather a pointer to the node $r$. So again, by line 40 $p$ does not execute
Lemma 13. Any node is linked in the queue at most once.

Proof. Suppose, for contradiction, that some node \( r \) is linked into the queue more than once. Consider the time just before \( r \) is linked into the queue for the second time.

By Property 1 of Lemma 10 and Definition 11, \( r \) is already within the the linked list starting from \( g_{Init} \) at this time. By Property 1 of Lemma 10 \( g_Tail \) points to the last node in this linked list just before \( r \) is linked into it again. Then, since Properties 1 and 3 of Lemma 10 always hold and \( g_Tail \) now points to the last node of the linked list, \( g_Tail \) has previously pointed to every node of the linked list. Consequently, since \( r \) is already within the linked list at this time, there must exist an earlier time when \( g_Tail \) pointed to \( r \). Thus by Lemma 12(a), \( r \rightarrow \text{flag} = 1 \) at this earlier time.

Finally, observe that \( r \rightarrow \text{flag} = 1 \) that has previously been set by some process. It is either a new value or the value set by a process that \( g_Tail \) previously pointed to at time \( t_1 \). At time \( t_2 \), \( g_Tail \) will execute line 55 to change the value in the location \( addr_0 \) to \( \text{flag} = 1 \). By Property 4 of Lemma 10, \( g_Tail \) can never be changed from 1. Thus \( r \rightarrow \text{flag} \) still contains 1 at the time when \( r \) is linked into the queue for the second time — contradicting Lemma 12(b).

Now observe that by Lemmas 10 and 13.

Corollary 14. The \( g_Tail \) can be set to point to each node at most once.

Lemma 15. Any new location allocated by a DEQUEUE operation can not be pointed to by \( g_{Head} \) more than once.

Proof. Suppose, for contradiction, that \( g_{Head} \) points to a location \( addr \) allocated by a single DEQUEUE operation twice. Let \( t_3 \) be the earliest time after a process \( p \) executes line 60 or 66 and succeeds in setting \( g_{Head} \rightarrow \text{addr} \). Let \( t_3 \) be the earliest time after time \( t_1 \) when a process \( q \) (not necessarily different from \( p \)) executes line 60 or 66 and also succeeds in setting \( g_{Head} \rightarrow \text{addr} \). Let \( t_2 \) be the last time \( q \) reads \( g_{Head} \) on line 55 before \( q \) sets \( g_{Head} \rightarrow \text{addr} \). If \( t_2 \) is before \( t_1 \), then since process \( p \) changes the value of \( g_{Head} \) at time \( t_1 \), at time \( t_3 \), \( g_{Head} \neq g_{Head} \) for process \( q \) on line 60 or 66 so \( q \) fails its CAS, a contradiction.

If \( t_2 \) is after \( t_1 \), process \( q \) will read a \( g_{Head} \) value that has previously been set by \( p \). It is either a newer value or the value set by \( p \). In the latter case, process \( q \) will execute line 55 to change the value in the location \( addr \), and then it finds the value in the location \( addr \) is not \( \text{NULL} \) in line 56. So \( q \) will return on line 57 without executing line 60 or 66 a contradiction.

In the former case, there must be some other process(es) performing successful CAS(es) on line 60 or 66 to change \( g_{Head} \) after \( p \) changes \( g_{Head} \). At least one of them must see \( g_{Head} \rightarrow \text{addr} \), and executes line 55 (and these happens between \( t_1 \) and \( t_2 \)). Then after time \( t_2 \), process \( q \) will find that the value in the location \( addr \) is not \( \text{NULL} \) in line 56. So \( q \) will return on line 57 without executing line 60 or 66 a contradiction.

Lemma 16. If a node’s flag field contains 1, then the node is in the linked list of nodes starting from \( g_{Init} \).

Proof. The node \( g_{Init} \) begins with \( \text{flag} = 1 \), but all other nodes have \( \text{flag} \) set to 0 when allocated (lines 4 to 7). The \( \text{flag} \) field of a node can then be set to 1 only on lines 35, 41, and 45.

If a node’s \( \text{flag} \) field is set to 1 on line 35 or 41 then this node was previously the node after the node pointed to by \( g_{Tail} \) (lines 29 to 30 or lines 32 to 33). So by Lemma 10 this node is in the linked list of nodes starting from \( g_{Init} \).

If a node’s \( \text{flag} \) field is set to 1 on line 45 then this node was previously linked on line 44 to a node previously pointed to by \( g_{Tail} \) (line 29). So by Lemma 10 this node is in the linked list of nodes starting from \( g_{Init} \).

Lemma 17. For each ENQUEUE operation, \( g_{Tail} \) has been set to point to the new node it allocates on line 4 exactly once.

Proof. An ENQUEUE operation returns only after, it has performed a TRYTOENQUEUE procedure on line 8 with the node that it allocated on line 9 and the TRYTOENQUEUE procedure has returned done instead of failed. A TRYTOENQUEUE procedure returns done either on line 35 or line 47.

In the former case, the TRYTOENQUEUE procedure previously found that \( l_{Node} \rightarrow \text{flag} = 1 \) (line 31), and then attempted to move \( g_{Tail} \) forward in the linked list of nodes starting from \( g_{Init} \) if \( g_{Tail} \) was not already at the end of the list (lines 32 to 36). By Lemma 16 \( l_{Node} \) was already in the linked list of nodes starting from \( g_{Init} \) when \( l_{Node} \rightarrow \text{flag} \) was set to 1. By Property 4 of Lemma 10 \( g_{Tail} \) was pointing
to the node before \( l_{\text{Node}} \) when \( l_{\text{Node}} \) was linked into the list. By Lemma 10, \( g_{\text{Tail}} \) can only be moved forward by node through the list starting from \( g_{\text{Init}}_{\text{Node}} \) that can never be disconnected. Thus after the \textsc{Trytoenqueue} procedure attempted to move \( g_{\text{Tail}} \) forward in the linked list of nodes starting from \( g_{\text{Init}}_{\text{Node}} \) if \( g_{\text{Tail}} \) was not already at the end of the list (lines 52 to 56), observe that \( g_{\text{Tail}} \) must have previously been set to point to \( l_{\text{Node}} \).

In the latter case, the \textsc{Trytoenqueue} procedure previously set the \texttt{next} field of some node to point to \( l_{\text{Node}} \) on line 44 then attempted to change \( g_{\text{Tail}} \) from \( l_{\text{Tail}} \) to \( l_{\text{Node}} \) on line 46. By Property 3 of Lemma 10 line 44 added \( l_{\text{Node}} \) to the end of the linked list starting from \( g_{\text{Init}}_{\text{Node}} \), with \( g_{\text{Tail}} = l_{\text{Tail}} \) pointing to the node that was previously at the end.

Then observe that after line 46 regardless of whether the CAS succeeds, \( g_{\text{Tail}} \neq l_{\text{Tail}} \). So \( g_{\text{Tail}} \) has been changed after line 44. Thus by Property 3 of Lemma 10, \( g_{\text{Tail}} \) must have been set to point to \( l_{\text{Node}} \), the next node in the linked list, at some time between lines 44 and 47.

So in both cases, a \textsc{Trytoenqueue} procedure only returns \texttt{done} after \( g_{\text{Tail}} \) has previously been set to point to \( l_{\text{Node}} \). Thus by Corollary 14 when an \textsc{enqueue} operation completes, \( g_{\text{Tail}} \) has previously been set to point to the node it allocated on line 18 exactly once.

\textbf{Lemma 18.} For each \textsc{dequeue} operation, \( g_{\text{Head}}.\text{addr} \) has been set to point to the new location it allocates on line 18 exactly once.

\textit{Proof.} An \textsc{dequeue} operation returns only after, it has performed a \textsc{trytodequeue} procedure on line 21 with the location that it allocated on line 18 and the \textsc{trytodequeue} procedure did not return \texttt{failed} (line 25). A \textsc{trytodequeue} procedure returns a non-\texttt{failed} value only on lines 57, 61 and 67.

If a \textsc{trytodequeue} procedure returns on line 57, then it has found that \( l_{\text{addr}} \neq \text{NULL} \) on line 56. A non-\texttt{NULL} value can only be written to \( l_{\text{addr}} \) on line 55 and only if \( l_{\text{addr}} \) has been previously pointed to by \( g_{\text{Head}}.\text{addr} \) (line 53). Thus \( g_{\text{Head}}.\text{addr} \) has previously pointed to \( l_{\text{addr}} \) when a \textsc{trytodequeue} procedure returns on line 57.

If a \textsc{trytodequeue} procedure returns on line 61 or line 67 then it previously set \( g_{\text{Head}}.\text{addr} \) to \( l_{\text{addr}} \) on line 60 or 65.

So in all cases, a \textsc{trytodequeue} procedure only returns a non-\texttt{failed} value after \( g_{\text{Head}}.\text{addr} \) has previously been set to point to \( l_{\text{addr}} \). Thus by Lemma 15 when an \textsc{dequeue} operation completes, \( g_{\text{Head}}.\text{addr} \) has previously been set to point to the location it allocated on line 18 exactly once.

By Lemmas 14, 15, 17 and 18 we have the following:

- For each \textsc{enqueue} operation, a pointer to the new node it allocates on line 4 can be written into \( g_{\text{Tail}} \) at most once, and has occurred exactly once when the operation completes.

- For each \textsc{dequeue} operation, a pointer to the new location it allocates on line 18 can be written into \( g_{\text{Head}} \) at most once, and has occurred exactly once when the operation completes.

Thus by Observation 9 the linearization points of the linearization \( L \) of \( H' \) are well defined and within their execution intervals.

\textbf{Theorem 19.} The linearization respects the sequential specification of a queue.

\textit{Proof.} By Definition 11 we regard the nodes between \( g_{\text{Head}}.\text{ptr} \) and \( g_{\text{Tail}} \) as the actual nodes of the queue, excluding the node pointed to by \( g_{\text{Head}}.\text{ptr} \). By Lemma 10 the queue is always connected and there is no transient state.

We linearize an \textsc{enqueue} operation at the point when \( g_{\text{Tail}} \) is moved forward through the linked list to point to the node that the \textsc{enqueue} operation allocated on line 4. Thus by Definition 11 the allocated node is enqueued exactly at the linearization point of the \textsc{enqueue} operation.

We linearize a \textsc{dequeue} operation that returns a non-\texttt{⊥} value at the point when the location that it allocated on line 18 is pointed to by \( g_{\text{Head}}.\text{addr} \) (line 65). Note that at this linearization point, \( g_{\text{Head}}.\text{ptr} \) would simultaneously be moved forward through the linked list, and that the node which is dequeued by Definition 11 has value equal to the return value of the \textsc{dequeue} operation.

We linearize a \textsc{dequeue} operation that returns \texttt{⊥} at the point when the process that would on line 60 set \( g_{\text{Head}}.\text{addr} \) to point to the location it allocated on line 18 last reads \( g_{\text{Tail}} \) on line 54. Note that at this linearization point, since \( l_{\text{Head}}.\text{ptr} = l_{\text{Tail}} \) (line 59), the queue is empty by Definition 11.

Thus our linearization points are simultaneous with the queue structure changes, so they follow the semantics of a queue structure.

\hfill \Box
**B Universal 2-nonblocking algorithm**

Below we give a universal construction for 2-nonblocking objects of any type $T$. The input to this algorithm is the sequential specification of the type $T$ given in the form of a function $\text{apply}_T$. This function maps tuples of the form $(op, s)$ where $op$ is an operation to be applied to the object and $s$ is the object current state, to a tuple $(s', r)$, where $s'$ is the new state of the object and $r$ is the return value. As in our 2-DNB queue algorithm, a process first tries to help another process only once (the “altruistic” phase in lines 2-3) and then it repeatedly tries to perform its own operation until it is done (the “selfish” phase in lines 4-11). All the processes that need help compete by writing their call for help on the same shared register $g_{Ann}$; as we explained earlier, this promotes fairness. The proof of correctness is a simplification of the one that we give for the 2-DNB queue algorithm.

**Algorithm 2** Universal 2-nonblocking algorithm (code for process $p$)

1: procedure $2NB-U(op)$  
2: $(op', L_{addr}) := g_{Ann}.read()$  
3: TryToDo($op', L_{addr}$)  
4: $L_{addr} :=$ a pointer to a new location  
5: \*$(L_{addr}).write(NULL)$  
6: repeat  
7: \hspace{1em} $t :=$ TryToDo($op, L_{addr}$)  
8: \hspace{1em} if $t =$ failed then  
9: \hspace{2em} $g_{Ann}.write(op, L_{addr})$  
10: \hspace{1em} end if  
11: until $t \neq$ failed  
12: return $t$  
13: end procedure  

14: procedure TryToDo($op, L_{addr}$)  
15: \hspace{1em} $L_{R} := g_{R}.read()$  
16: \hspace{1em} \*$(L_{R}.addr).write(L_{R}.response)$  
17: \hspace{1em} if \*$(L_{addr}).read() \neq NULL$ then  
18: \hspace{2em} return \*$(L_{addr}).read()$  
19: \hspace{1em} end if  
20: \hspace{1em} $(s', response') :=$ apply$_T(op, L_{R}.state)$  
21: \hspace{1em} if CAS($g_{R}, L_{R}, (s', response', L_{addr})$) then  
22: \hspace{2em} return $response'$  
23: \hspace{2em} else  
24: \hspace{3em} return failed  
25: \hspace{2em} end if  
26: end procedure

---

**Fig. 9** Shared objects for the universal 2-nonblocking algorithm

- $g_{Ann}$: A register with two fields:
  - $op$: the operation to execute.
  - $addr$: a pointer to a location that stores a returned value or NULL, initially pointing to any non-NUL value.

- $g_{R}$: A compare&swap with three fields:
  - $state$: a register containing an integer, initially with the initial state.
  - $response$: the response value to be returned by the function, initially NULL.
  - $addr$: a pointer to a location where $R.response$ will be stored, initially pointing to any non-NUL value.