# Space-Optimal Wait-Free Queues

(Preliminary Version)

Ted Herman and Valeriu Damian-Iordache
University of Iowa herman@cs.uiowa.edu

August 20, 1997

**Abstract**

The paper presents a wait-free construction for a bounded-length queue. The total space requirement is $Dk + O(kn)$ where $D$ is a bound on the queue length, $k$ is the size of a queue item, and $n$ is the number of processes that concurrently execute operations on the queue. Thus in the case $D = \Omega(n)$, the construction is optimal with respect to space. The construction uses only atomic read/write and compare-and-swap operations on single memory words.

## Contents

# 1 Introduction

Queues are fundamental building blocks for operating system internals. Because operating systems manage concurrent activities, it is natural to specify that data types such as queues have concurrent operations. Desirable attributes for such concurrent operations are properties such as deadlock-freedom, non-blocking, and ultimately wait-freedom.

The possibility of a wait-free queue is implied by the universal construction [1] in terms of atomic read/write and low-level consensus primitives. Two sources of overhead arise from a direct implementation of a queue using the universal construction, time overhead for an operation and space overhead for the data structures used in the construction. The paper [4] suggests an approach improving time overhead, and the research described in this abstract tackles the problem of space overhead for the case of a wait-free queue.

The result given here is a wait-free construction for a bounded-length queue. The total space requirement is $O(D+n)$ where $D$ is a bound on the queue length (we assume each queue item has constant space) and $n$ is the number of processes that concurrently execute operations on the queue. Thus in the case $D = \Omega(n)$, the construction is optimal with respect to space (by comparison, direct application of the universal construction results in an $O(Dn^3)$ space requirement since each of $n^3$ cells represents a queue state). The problem of constructing a space-efficient wait-free is made difficult when the type of atomic operation is limited. Using *Fetch&Add* and a two-word *Compare&Swap* simplifies the construction task (for instance, a space-efficient wait-free heap given in [6] uses a conditional atomic assignment to two words). Our work shows that a wait-free queue can be implemented using only atomic read/write and compare-and-swap operations on single memory words.

## 2 Construction Overview

The basis for the construction is a linked list of cells, resembling the work [3, 2]. Each cell can contain one queue item; a shared pool of $D + O(n)$ cells contains the linked list representing the queue and other cells that are either available or at various intermediate states between queue membership and availability for recycling into the queue.

A *dequeue* operation either returns an item from the queue or an *empty* indication in case the queue had no items at the time (modulo linearizability) of the operation. An *enqueue* operation may fail if the queue contains $D$ items at the time (modulo linearizability) of the enqueue.

The primary technical difficulties with a link list representation derive from the helping method and the unreliability of the compare-and-swap in following a pointer chain, as explained in [2]. We resolve these difficulties by three techniques: $(i)$ instead of using addresses as pointers, cells are given unique *id* fields and all "next" pointers rely only on cell ids; $(ii)$ cell recycling and id recycling is carefully controlled and synchronized with *publication* (and subsequent *retraction*) of cell ids that are currently scanned by any process; and $(iii)$ the dequeue operation is split into several parts, a `deqplace` sub-operation, a `dechain` sub-operation, and remaining steps that return an item to the operation's invoker.

The id of a cell is a memory word consisting of two fields, a process number and a sequence number. All the *Compare&Swap* operations for our construction refer to a word containing an id or a word of status bits: no *Compare&Swap* operation is used to manipulate an address or the combination of an address and a counter (as in [3, 5]). Mapping a cell id to its location is implemented by scanning the pool of all cells (for better performance, an auxiliary *hint address* field could be provided).

## 3 Data Structures

Our wait-free queue implementation uses a fixed amount of memory: $O(D + n)$ words of shared memory are required, where $D$ is the maximum size of the queue and $n$ is the number of concurrent processes invoking queue operations. Each word of memory behaves as an $n$-writer, $n$-reader, atomic register with a *Compare&Swap* primitive. The $\mathsf{C\&S}(w, a, b)$ primitive compares a memory word $w$ to a value $a$ and then overwrites $w$ with $b$ if the comparison result satisfies $w = a$; the primitive also returns to the invoking process a boolean indication of whether or not the primitive assigned $w = b$. Both $a$ and $b$ are values local to a process (they are not words of shared memory).

### 3.1 Queue Representation: Chain of Cells

The queue implementation in this paper is roughly the standard linked-list data structure: the queue is represented by a chain of *cells*, where each cell contains a pointer and the application data associated with the queue element. Cells are dynamically allocated and marked free by queue operations. A dequeue operation takes data from the head of the chain and an enqueue operation threads a free cell on the end of the chain. The fields of a cell are:

| Cell record: | `next` | `status` | `enqid` | `deqid` | `data` |
|---|---|---|---|---|---|

The `data` field holds the application data for a queue element. Each of the other fields is a word of shared memory, and these fields are documented in subsections that follow.

The cells available for queue operations consists of a fixed array of $D + 7n + 4$ cell records. The notation `cell[i]` refers to the $i$-th cell record in the array of cells. Dynamic allocation is thus implemented by "recycling" cells contained in the fixed array of cells.

### 3.1.1 Operation Identifiers

One of the difficulties in implementing a wait-free queue using the C&S primitive is the *ABA problem* [2]: between reading a word of memory and subsequently using C&S to modify that word, the contents of the word could potentially change any number of times, then subsequently be restored to its original value so that the C&S succeeds where one might hope that it fail. In particular, suppose that a cell to be modified is recycled between a read and C&S; recycling cells cannot be avoided in a queue implementation that uses bounded memory. In certain conditions, it could be possible that a cell is recycled yet contains the same value (for instance a pointer) that a scheduled C&S primitive will modify. The modification resulting from such a C&S primitive could thus erroneously modify a cell field.

The difficulty of cell recycling and pending modification has a simple solution: do not recycle cells that potentially have pending modifications. The universal construction uses a technique of marking cells whereby each process $p$ sets a bit in a cell when it is known that $p$ will not subsequently modify the cell. Cell `x` can be recycled when every process has set a bit in `x`. This technique immediately requires $\Omega(Dn)$ space for a queue, so we look to another method.

The problem of cell recycling can be attributed to identifying a cell by its *address* rather than its place in the history of queue operations. We therefore equip each cell with a field specifying the identity of the enqueue operation responsible for its placement in the queue. Such an identifier is a word representing a tuple $\langle p, i \rangle$ where $p$ is a process identifier and $i$ is a sequence number. Throughout the paper we refer to these tuples as *ids.* Ids are useful not only for enqueue operations, but also for dequeue operations and to synchronize completion of internal operations by reinterpreting a cell id as a tuple $\langle s, i \rangle$ where $s$ is a small integer representing the status of an internal operation and $i$ corresponds to the sequence number of an enqueue or dequeue operation's id field. It is important that id fields consist of a single word, since they are often manipulated with C&S primitives.

When a cell to be enqueued is allocated, it is assigned a unique id field called the `enqid`. The space of possible `enqid` ids is presumably larger than the number of cells; we use this property to insure that cell recycling can be detected — a recycled cell will have a different `enqid` than its previous incarnations. For technical reasons, we find it also useful to record the identity of a `dequeue` operation in each cell. Each cell has both an `enqid` and `deqid` field. The `next` field of a cell contains id rather than an address pointer.

#### 3.1.2 Cell Status

Each cell recycled through the queue repeatedly goes through various stages of allocation as it is made free and later reclaimed. A cell's `status` field indicates the current state of a cell, one of two possible values, `allocating`, or `active`.

#### 3.1.3 Reserved Cells and Ids

Three cells in shared memory are set aside from dynamic allocation and play special roles in queue operations. The ids of these three cells are fixed and known to all processes.

- `head` is the permanent anchor cell for the queue. The id of `head` is `headid`.
- `null` is a special cell introduced to make regular the coding of suboperations following the chain of cells in the queue. The id of the `null` cell is `nullid`. The `null` cell is an end marker for the chain of cells in the queue, and the most recently enqueued cell in the chain contains `nullid` in its `next` field.
- `void` is another special cell introduced to make regular coding of suboperations; it has id `voidid`.
- One other fixed id is known to all processes and used frequently in the code of operations: the id `noid` is a dummy id that refers to no cell, and is the default id placed in fields of a newly allocated cell.

### 3.2 Wills and Helping

The notion of *helping*, as described in the universal construction, appears to be essential to any non-trivial wait-free construction of a sequential object such as a queue. Accordingly, each enqueue and dequeue operation is described by a *will* record; process $p$ announces its current operation in a will so that other processes can help $p$ and thereby assure that $p$'s operation is completed in a finite number of $p$'s steps. A will record specifies the id field and type (dequeue or enqueue) for the operation. For an enqueue operation, the `id` field is the `enqid` of the cell to be threaded onto the queue; the `enqIndex` is the index of the cell to be enqueued, for instance, `cell[W.enqIndex]` will be enqueued for will `W`. For a dequeue operation, the `deqEnqid` will (upon completion) contain the `enqid` of the cell that is dequeued.

Will record:

| id | type | enqIndex | deqEnqid | status |
|---|---|---|---|---|

The `status` field is used for synchronization purposes, and contains a tuple of the form $\langle s, i\rangle$ where $s$ represents the state of the will, either *start* or *done*, and $i$ is the sequence number of the will's `id` field.

The helping of wills is implemented by having an array of will records reside in shared memory: `will` is an $n$-element array of will records. Each process `i` enters its operation in `will[i]`.

### 3.3 Id Publication

The use of cell ids in `next` and other fields enables our solution to the ABA problem. To ensure that cell recycling does not duplicate cell ids, and also to prevent recycling of cells currently processed by certain suboperations, we introduce a mechanism by which a process can *publish* a number of ids. Publication of an id is implemented by reserving five words of shared memory per process — these words contain the ids currently published by a process. Publication operations, described in Section 4, allow a process to publish or retract an id, and to query the set of all published ids. We consider the reserved ids `headid`, `nullid`, and `voidid` to be permanently published.

## 4 Low-level Routines and Exceptions

This section describes some low-level routines for which we do not present programs; these routines have straightforward implementations in terms of atomic read and write primitives to shared memory. For each of these low-level routines, we present the signature (result type and list of parameters) and a brief description of the function's result. Several of the routines may signal *exceptions* under certain conditions. An exception either aborts the program that invokes the routine or is handled by an exception mechanism provided by the program calling the routine. We use Java-like syntax for the signature of a routine. The signature

```
int foo ( int x, cell y, id k ) throws Error;
```

specifies a routine `foo` that either returns a result of type `int` or throws an `Error` exception. The three parameters to `foo` are of type `int`, `cell` and `id`.

Rather than relying on pointers to represent a chain of cells in the queue, the implementation uses an id tuple in a cell's `next` field. The following routines are used to locate cells by an id, create ids, and publish ids.

`cell find( id m ) throws Error;`

The `find` routine locates the cell corresponding to a given id `m`. If no cell has an `enqid` field containing `m`, or if `m` is a reserved id (one of `headid`, `voidid nullid`, or `void`), then `find` throws an `Error` exception. The implementation of `find` is not atomic, so it is possible that `find(m)` throws an `Error` even though, at some point during the execution of `find(m)` a cell `x` with `x.enqid` of `m` exists; if such a cell `x` exists throughout the execution of `find(m)`, then the routine will return `x`.

`cell snapshot(cell x)`

The `snapshot` routine creates a new, local copy of cell `x` — the local copy is not in shared memory. This local copy is not atomic, although each field is atomically copied from some value of cell `x`.

`void verify(cell x, cell y) throws Error`

The `verify` routine compares the fields of the two cells `x` and `y`, either returning to the caller if all fields were found equal, or throwing and `Error` exception if any fields were found to differ. The comparison of `x` and `y` is not atomic.

`id composeId(int a, int b)`

This function returns as an id the tuple $\langle a, b \rangle$, where $a$ is either a process number or a status value.

`int processOf(int m)`

The argument `m` is an id, hence a tuple of the form $\langle p, i \rangle$. The function call `processOf(`$\langle p, i \rangle$`)` returns the value $p$.

`int seqnumOf(int m)`

The argument `m` is an id, hence a tuple of the form $\langle p, i \rangle$. The function call `seqnumOf(`$\langle p, i \rangle$`)` returns the value $i$.

`id getNewId()`

A process invokes `getNewId` to obtain an id that is not contained in the `next`, `enqid`, or `deqid` of any cell in shared memory. (Note that the invoking process number is an implicit parameter to `getNewId()`.) Although the execution of `getNewId` is not atomic, each process $p$ controls the creation of all cell ids of the form $\langle p, i \rangle$, so a non-atomic scan of the cell array suffices to verify that a new id differs from any currently existing id or any id concurrently created by another process.

`void publish(id m)`

A process invokes `publish(m)` to publish id `m` in a shared "publication" set. One possible implementation of such a publication object would be a record in shared memory containing five words for the storage of five ids (we show in the sequel that at most five ids are needed at any time in a publication object.) A `publish(m)` has no effect if `m` is already stored in the publication set. Each process has its own publication set and the `publish(m)` invocation implicitly refers to the publication object belonging to the invoking process.

`void retract(id m)`

A process removes an id `m` from its publication set; if `m` is not in the publication set, then `retract(m)` has no effect.

`void retractALL()`

The `retractAll` routine resets the state of a publication set to an empty set.

`void retractALLbut(id m)`

An execution of `retractAllbut(m)` assigns the publication set to the value {`m`}.

`boolean isPublished(id m)`

A process invokes `isPublished(m)` to determine whether or not `m` is contained in the publication set of any process. Execution of `isPublished` is not atomic, so mileage may vary in the presence of concurrent publication routines.

`void markDone(will w, id m)`

Enqueue or dequeue operations invoke `markDone` to set the status of a specified will to `done`. The id parameter `m` confirms the will's identity: `markDone(w,m)` will change `w.status` only if (*i*) `processOf(w.status)` is `start`, and (*ii*) `seqnumOf(w.status)` and `seqnumOf(m)` are equal. Conditions (*i*) and (*ii*) are atomically checked along with the write to `w.status` by a C&S primitive.

`boolean isDone(will w, id m)`

A process invokes `isDone(w,m)` to determine if the operation associated with will `w` has completed with respect to id `m`. The result of `isDone(w,m)` is *true* if `processOf(w.status)` is not `start`, or if `seqnumOf(w.status)` differs from `seqnumOf(m)`. These two conditions are not atomically checked.

## 5 Cell and Will Invariants

At any state of shared memory, a number of properties hold invariantly for cell and will data structures. Some of these invariants are assertions about individual cells and wills or are assertions concerning the history of modifications to structures. Other invariants are relations between cells and between wills and cells. This section defines a number of invariants used freely to reason about procedures defined in subsequent sections. For the most part, the invariants of this section are consequences of the queue construction that can be straightforwardly derived from the programs presented, and we do not prove these invariants.

At any state of shared memory, we call a cell *free* if its `enqid` has a `noid` value. The following invariants are assertions over the state of shared memory at all times during operation execution.

**(G1)** No two non-free cells have the same `enqid`.

**(G2)** For any cell `x`, if `x.next` contains a non-reserved id (i.e., the id is not one of `headid`, `voidid`, `noid`, or `nullid`), then there exists a cell `y` such that `x.next` = `y.enqid`.

**(G3)** The set of cell `next` fields containing non-`noid` ids define an acyclic directed graph of cells (there is no cycle of cells).

**(G4)** If distinct cells have the same value for the `deqid` field, then this value is `noid`.

To reason about queue operations it is useful to distinguish between the states and possible modifications that a cell experiences as it joins the queue, is removed from the queue, and is recycled to the pool of cells available for dynamic allocation. For any cell `x` let $x_i$ denote the *i-th incarnation* of `x`, where an incarnation of a cell consists of the history of modifications to cell `x` following the assignment of a non-`noid` id to `x.enqid` up to the next change to `x.enqid`. During incarnation $x_i$, there are two possibilities for cell `x` at any state: either `x` is *chained*, meaning that some cell `y` exists such that `y.next=x.enqid`, or `x` is *unchained*, meaning that no cell's `next` field contains the id of cell `x`.

To facilitate discussion of dequeue operation routines presented in Section 10, we say that a cell is *logically dequeued* if its `deqid` is any value except `noid`. After a cell has been logically dequeued, but before it becomes unchained, it is removed from the chain of cells representing the queue. If a cell `x` is not in the chain of cells anchored at the `head` cell, then we say `x` is *dechained*. The following temporal invariants hold for incarnations of non-reserved cells (the reserved cells do not have incarnations, since they are not recycled).

**(G5)** Incarnation $x_i$ of cell `x` ends by the assignment `x.enqid=noid`.

**(G6)** The field `x.deqid` of incarnation $x_i$ changes precisely once, from value `noid` to a non-`noid` value.

**(G7)** The sequence of values of `x.next` for incarnation $x_i$ is: ($i$) `nullid`; ($ii$) some id other than a reserved cell id, i.e., some value outside the set {`headid`,`nullid`,`voidid`,`noid`}; ($iii$) `voidid`. Thus `x.next` changes twice during incarnation $x_i$.

**(G8)** The incarnation $x_i$ consists of three segments: first, `x` is unchained and `x.next=nullid`; second, `x` is chained; and third, `x` is unchained. This invariant permits us to refer to the state `x` in an incarnation as either *prechained*, *chained*, or *postchained*.

**(G9)** At the state where cell `x` transits from unchained to chained during $x_i$, cell `x` is in the chain of cells originating at `head` and `x.next` is `nullid` (cell `x` is the item at the end of the queue when it becomes chained).

**(G10)** While `x` is prechained during $x_i$, the value `x.enqid` is published.

Similar to invariant (**G7**), we also have an invariant for the `head` cell's `next` field:

**(G11)** The value of `head.next` is never a reserved id (i.e., it is never one of `headid`, `voidid`, `noid`, `nullid`). The `next` field of `head` is the only `next` field of any cell that atomically changes from a non-reserved id to another non-reserved id.

**(G12)** There exists exactly one chained cell `x` with `x.next` $=$ `nullid` at every state.

Invariant (**G12**) implies that an empty queue cannot be represented by having `head.next` be `nullid`. In Section 10, where the dequeue sub-operations are described, we explain that a dequeue operation consists of phases that first mark a cell as logically dequeued, and later remove the logically dequeued cell from the chain anchored at `head`. This technique allows an empty queue to be represented by any chain of cells such that each cell in the chain has been marked as logically dequeued.

Similar to the way changes of cell `enqid` id fields define cell incarnations, we associate *will incarnations* with changes to the `id` field of a will record. For any will `w` let $w_i$ denote the `i-th incarnation` of `w`, where an incarnation of a will consists of the history of modifications to will `w` following the assignment of a non-`noid` id to `w.id` up to the next change to `w.id`. Each will incarnation represents a queue operation: there is a bijection between the history of successful queue operations (enqueue and dequeue) and the history of will incarnations. For any will incarnation representing a successful operation (i.e., neither an enqueue that fails because the queue is full, nor a dequeue that fails because the queue is empty), we define an associated cell incarnation: the *cell incarnation of* $w_i$ is the incarnation $x_j$ that overlaps at some state with $w_i$, and corresponds to `w.enqIndex` if $w_i$ represents an enqueue operation or corresponds to `w.deqEnqid` in case $w_i$ represents a dequeue operation. (Technically, this definition should take into account the possibility of incomplete cell incarnations, which occur when an item is enqueued but never dequeued; for simplicity, we assume that every enqueued item is evenutally dequeued in the above definition.) Some invariants concerning will incarnations follow.

**(G13)** The status of a will during an incarnation is initially `start`, and changes only once during the incarnation to `done`.

**(G14)** The type of a will during an incarnation is constant; the type is either `ENQ` or `DEQ`. The `enqIndex` field is also constant throughout a will's incarnation.

**(G15)** If incarnation $w_i$ has type `ENQ` and status `start`, then `w.enqIndex` is the index of the cell to be enqueued, i.e., `cell[w.enqIndex]` is the cell to be enqueued for $w_i$. While `w.status` is `start` during $w_i$, then the fields `cell[w.enqIndex].enqid` and `w.id` have the same value.

**(G16)** If, during incarnation $w_i$, the field `cell[w.enqIndex].next` has a value different from `nullid`, then `w.status` is `done`.

Every cell incarnation corresponds to some successful enqueue operation: incarnation $x_j$ is the cell incarnation of some will incarnation $w_i$. Similarly, every cell incarnation corresponds to a dequeue operation. Subsequent sections use terminology such as "the will of cell `x`" or the "cell of will `w`" as abbreviations for the correspondences between will and cell incarnations described here. Note that since the cell `enqid` and will `id` fields are ids of the form $\langle p, i\rangle$, we can associate a process with each will and cell. The *creator* of $w_i$ is the process $p$ such that `w.id` $= \langle p, i\rangle$ during incarnation $w_i$; the *creator* of $x_j$ is the process $p$ such that `x.enqid` $= \langle p, j\rangle$ during incarnation $x_j$. The cell incarnation of $w_i$ and $w_i$ thus have the same creating process. For an enqueue operation represented by $w_i$, the assertion "the cell of $w_i$ is chained" means that the cell incarnation $x_j$ of $w_i$ has reached a state where `x` is chained.

## 6 Cell Allocation and Recycling

Cell recycling is crucial to the implementation of a space-efficient queue. Concurrent enqueue and dequeue operations complicate the determination of when a cell can be recycled and whether or not the supply of available cells has been exhausted. Since our implementation uses read, write and C&S as the only atomic memory primitives, a simple counter of the number of free cells cannot directly be maintained. A property of our cell recycling routine is that, if no cell is available for an allocation request, the queue can be considered "full" — and this results in a failing enqueue operation which returns a queue-full indication to the enqueue's invocation. A precise definition of a full queue is given later in this section.

The formal definition of cell recycling is: cell `x` is *recycled* by process $p$ when the following steps by $p$ occur: (**1**) $p$ ends one incarnation of `x` and either starts a new incarnation of `x` or converts `x` to a free cell; or (**2**) $p$ starts a new incarnation for `x` that was previously a free cell. Thus, a cell is recycled by ending an incarnation, beginning an incarnation, or a combination of these two. This definition of a recycling operation permits two processes to participate in the transition from one incarnation to the next for a given cell: it may be that process $p$ ends incarnation $x_i$ and process $q$ starts incarnation $x_{i+1}$.

The first question for cell recycling is: when is it safe to convert a cell's status from `active` to `allocating`? We say that a cell `x` is *freeable* if it is `active` and either free or is postchained in its incarnation. The following routine is a partial determination of whether or not a given cell is freeable.

```
boolean freeable( cell x ) {
   if (x.enqid == noid) return true;
   for (i=0; i<numCells; i++)
      if (cell[i].next == x.enqid) return false;
   if (published(x.enqid) || published(x.deqid)) return false;
   return true; }
```

Because `freeable` does not execute atomically, we require certain properties of queue operations to establish that a *true* result of `freeable(x)` correctly identifies `x` as a freeable cell. Since `freeable` modifies no cell or will fields, it satisfies all the cell and will invariants of Section 5. The order of checking conditions of a freeable cell (first that it is unchained, then that it is unpublished) turns out to be crucial for the correct processing of concurrent enqueue and dequeue operations; this order of checking does not provide atomicity, but provides a sequencing property used in Section 7 for the proof of Lemma 4.

**Lemma 1** If process $p$ obtains the result *true* from invoking `freeable(x)` and no other process $q$ recycles `x` during execution of `freeable(x)`, then cell `x` is freeable at the time `freeable(x)` returns.

**Proof:** In the case of free `x` the result is immediate since no concurrent recycling occurs. If `x` belongs to an incarnation during `freeable(x)`, then `freeable(x)` $=$ *true* implies no cell was found to point to `x` during computation of `freeable(x)`. Invariant (**G10**) asserts that `x` remains published while it is prechained. Once `x` is postchained, it remains unchained. Therefore `freeable(x)` returns *true* only if `x` is freeable. □

Note that `x` can be freeable, but `freeable(x)` returns *false* because `x` was published during execution of `freeable(x)`. Additional properties shown below ensure that the number of instances where `freeable(x)` returns *false* for freeable `x` is limited.

If a freeable cell has been identified, the next step in recycling is to convert that cell to `allocating` status. The `makeFree` routine does this. The `makeFree` routine is also an important serialization point in the queue construction, as stated in the following properties.

**(G17)** No cell `x` is recycled unless `makeFree(x)` first executes on that cell.

**(G18)** No cell `x` transits from unchained to chained state unless `makeFree(x)` is first invoked to recycle `x`.

These properties, together with the atomicity of the C&S primitive, are used to guarantee that no cell is simultaneously allocated by two processes.

```
boolean makeFree( cell x ) {
   if (!C&S(x.status,active,allocating)) return false;
   if (!freeable(x)) { x.status = active; return false; }
   x.next = voidid; x.enqid = noid; x.deqid = noid; return true; }
```

**Lemma 2** If process $p$ obtains the result *true* from invoking `makeFree(x)`, then cell `x` was freeable during execution of `makeFree(x)` and no other process recycles `x` until process $p$ either completes recycling `x` or returns `x` to the pool of free cells.

**Proof:** Notice that `makeFree` speculatively converts the status of *any* given cell `x` from `active` to `allocating`, which appears to be a mistake if `x` is not freeable. The third line of the routine corrects the status if `x` is not freeable. The reason for this speculative technique is serialization: the same cell `x` cannot simultaneously be captured by two processes executing `makeFree` because the C&S primitive succeeds for only one process, and all recycling occurs by `makeFree` processing (asserted in (**G17**)). Because `x.status` is `allocating` during execution of `freeable(x)`, the antecedent condition of Lemma 1 is satisfied and `freeable(x)` returns *true* only if `x` is freeable. □

The remainder of this section presents the procedures for allocating cells in preparation for enqueue operations. The main procedure, `allocate(p)`, uses a helping strategy to locate a freeable cell and convert it into a cell suitable for enqueuing a new item onto the queue. The primary concerns for the correctness of `allocate(p)` are:

1. If a cell `x` is returned by `allocate(p)`, then `x` is not allocated again until `x` is enqueued and dequeued from the queue.

2. The execution of `allocate(p)` should be wait-free.
3. If `allocate(p)` fails to allocate a cell, it should be that the queue contained at least $D$ items at some point during the execution of `allocate(p)`.

The first of these concerns is addressed by the construction of `makeFree` and invariants (**G5**)–(**G10**). The third concern is the main difficulty because `freeable(x)` does not guarantee to return *true* for a freeable cell `x`, and this can result in underestimation of the actual number of freeable cells. Consider the following routine `getFreeCell()`, designed to locate a freeable cell, convert it to `allocating` status, and return its index in the cell array.

```
int getFreeCell() {
  for (t=0; t<2n^2; t++)
    for (i=0; i<numCells; i++) if (makeFree(cell[i])) return i;
  return -1; }
```

An execution of `getFreeCell()` may fail to obtain a cell, returning `-1` instead of an index, for a number of reasons. For instance, it could be that no freeable cells exist because all cells are in the current queue. Or it could be that a slow execution of `getFreeCell()` happens to find that each cell it tests is not freeable at the time of the test, but at each point during the execution, many other cells were freeable because of concurrently completing enqueue and dequeue operations. The following invariants help us in proving some properties about `getFreeCell()` executions.

**(G19)** For each process $p$, at most 5 cell ids are published by $p$ at any state.

**(G20)** A pending enqueue or dequeue operation can block from recycling at most two cells that are neither published nor contained in the current queue. Cases for such pending operations are:

- A pending enqueue blocks from recycling a cell that it has obtained by the `allocate(p)` routine, but has not yet been assigned a new `enqid` to start a new incarnation; such a cell has `allocating` status.
- A pending dequeue may block from recycling two cells that are neither contained in the queue nor are published.

**(G21)** At most four cells are blocked from recycling that are neither published, nor blocked by a pending enqueue or dequeue operation. These are: `head`, `null`, `void`, and one additional cell in the case that the queue is empty.

From invariants (**G19**) and (**G20**), it follows that for a given state there can be $7(n-1)$ cells that are freeable, but not obtained a `getFreeCell` invocation (assuming that the process invoking `getFreeCell` is neither a dequeue operation nor has published any ids). In fact, the number of such cells can exceed $7(n-1)$ during an execution of `getFreeCell` because, although the number of published cells per process is at most 5 at any state, the set of published cells may change during the (nonatomic) execution of `getFreeCell`. We therefore introduce an invariant limiting the scope of ids that a process publishes during an operation.

**(G22)** When a process completes a dequeue or an enqueue operation, all ids it has previously published are retracted.

**(G23)** During process $p$'s execution of a dequeue or enqueue operation, any id published is either the id of some pending operation's will or `k=x.enqid` of a cell `x` contained in the queue at the instant that `publish(k)` was invoked.

These invariants assert that a process does not publish ids at random, but limits the ids it publishes to refer to cells it encounters while scanning the chain of queue cells.

**Lemma 3** If the queue contains fewer than $D$ items when process $p$ invokes `getFreeCell()`, and no other process concurrently and successfully executes `getFreeCell()`, then process $p$ obtains an `allocating` cell.

**Proof:** Because no concurrent operation obtains an allocating cell, the cells contained in the queue during $p$'s execution of `getFreeCell()` is limited to the $k < D$ cells initially present in the queue plus at most $(n-1)$ other cells added by concurrent enqueue operations; however, any such concurrent enqueue operations have published the ids of the cells added up to the time the cells are added to the queue. We claim that the total number of cells that are either published, in the queue, or blocked by a pending operation is at most $k + 7n$ during $p$'s execution of `getFreeCell()`. To see this, consider the (at most) 5 cells published by some process at the moment when $p$'s execution of `getFreeCell()` begins. None of these 5 cells may be contained in the current queue (all may refer to old cell incarnations). However, any cell publication subsequent to $p$'s beginning of its `getFreeCell()` invocation, must refer to some cell contained in the $k$ cells initially present or to one of the (at most) $n-1$ cells added to the queue during $p$'s invocation. Thus at most $5n$ published ids outside of the (at most) $k+n-1$ queue cells will return *false* for an execution of `freeable(x)` and be blocked from allocation. Therefore $p$ finds at least one freeable cell during execution of `getFreeCell()` and successfully obtains such a cell by the `makeFree` routine. □

The proof of Lemma 3 does not refer to the $2n^2$ bound in the outer loop of `getFreeCell`'s code: the $2n^2$ bound is significant only for wait-freedom, whereas Lemma 3 only shows that the application of `makeFree` to all cells in the cell array is sufficient to ensure that *some* process obtains a free cell. For the proof of Lemma 3, the outer loop of `getFreeCell` could be discarded. The outer loop is useful in the context of `allocate`, the final component of dynamic cell allocation.

The routine `allocate(p)` is executed by a process `p` to obtain a cell for an enqueue operation. The parameter `p` is explicitly given (unlike other functions where we let the process be implicit) to make the helping role of processes clear in the allocation strategy. An $n$-word array of shared memory is used to announce the request for a cell to other processes and announce the completion (successful or failing) of a an allocation request. To announce a cell request, process `p` assigns `F[p]=-1`. The value `-1` will be replaced either by the index of a cell allocated to satisfy the request, or by the value `-2` if the request was not satisfied.

```
int allocate(p) {
  F[p] = -1; /* Announce need for cell */
  h[p] = (h[p]+1) mod n ; /* Choose whom to help */
  c = getFreeCell();
  if (c == -1) {
    if (C&S(F[p],-1,-2)) return -1;
    else return F[p]; }
  if (C&S(F[h[p]],-1,c)) {
    c = getFreeCell();
    if (c == -1) {
      if (C&S(F[p],-1,-2)) return -1;
      else return F[p]; } }
  if (C&S(F[p],-1,c)) return c;
  else {
    cell[F[p]].status = active;
    return c; } }
```

We have also used an array for *help indicies.* Each process `p` maintains the index of a process it chooses to help for an allocation request in `h[p]`; this was done for notational convenience and can be instead implemented by a private variable for process `p`. The strategy of `allocate(p)` is, roughly speaking, that `p` first tries to obtain a cell using `getFreeCell`, and if that succeeds, `p` tries to give the cell to another process requesting cell allocation; process `p` then attends to its own request, possibly calling `getFreeCell` a second time to finish its cell allocation.

**Theorem 1** Any execution of `allocate(p)` by process `p` returns `-1` only if the queue contained at least $D$ cells at some state during the execution.

**Proof:** Heading for a contradiction, suppose that at each state during some execution of `allocate(p)` the queue contains fewer than $D$ items, but `allocate(p)` returns `-1`. According to Lemma 3, any scan of the cell array by the inner loop of `getFreeCell` only fails to obtain a cell if some other process successfully allocates a cell. Therefore, if `getFreeCell()` returns `-1`, some $2n^2$ allocations by other processes succeed during the execution of `getFreeCell()`. At least half of these allocations correspond to the helping portion of `allocate`, where a process attempts to give a cell to another requesting process. In a sequence of $n$ successful `allocate` calls, a process attempts to give a cell to each other process. Consequently, if $n^2$ successful `allocate` calls occur during `p`'s execution of `getFreeCell()`, at least one process gives a cell to process `p`, which is a contradiction. □

# 7 Queue Scans

An enqueue operation consists of allocating a cell, copying item data to the cell, and then threading the cell onto the end of the chain representing the queue. Our implementation does not provide a pointer to the end of the queue, so an enqueue operation follows the chain of cells from `head` until the end of the chain is found. A dequeue operation has two phases, one of which follows the chain from `head` until a cell is found that has not yet been dequeued. The difficulty for either enqueue or dequeue operations is that the chain of cells is not static, but concurrently changes as other processes add cells to the chain, remove cells from the chain, and recycle freeable cells. If an operation executes slowly, it may be that the current cell it is reading in the chain is recycled and again added to the queue, which causes the slow process to incorrectly follow the chain.

A *scan of queue cells* is a procedure execution that sequentially examines a sequence of cells in the order they were added to the queue chain. To express this formally, we define $x_i \prec y_j$ for two cell incarnations $x_i$ and $y_i$ to be the predicate: there exists a state $\sigma$ such that $\sigma$ is a state of both $x_i$ and $y_i$ and the following holds at state $\sigma$:

$$\texttt{x.next} = \texttt{y.enqid} \ \wedge \ \texttt{y.enqid} \notin \{\texttt{noid}, \texttt{voidid}, \texttt{nullid}, \texttt{headid}\}$$

A scan of queue cells is correct if $(i)$ the scan examines a sequence of cells $\texttt{x}^0$, $\texttt{x}^1$, $\ldots$, $\texttt{x}^m$ such that $\texttt{x}^0 = \texttt{head}$ and for $0 \leq k < m$ there are indices $i$ and $j$ satisfying $x_i^k \prec x_j^{k+1}$; and $(ii)$ cell fields of $\texttt{x}^{k+1}$ examined during the scan, for $0 \leq k < m$, correspond to some state at which $\texttt{x}^k$ and $\texttt{x}^{k+1}$ are linked as specified in the definition of the $\prec$ relation.

Our implementation of queue scanning does not guarantee that a process starting a scan will correctly complete its scan; instead, we have an implementation that is partially correct. While a scan continues it is correct, and if it does not continue or complete, then an exception is thrown — which aborts the scan. The following procedure (represented as a code fragment for different instantiations of operation-dependent statements) is used in enqueue and dequeue operations to scan queue cells.

```
q = head;
Sq = snapshot(q); /* Initialize chain scan */
lag = noid;
while (true) { /* Advance to first/next chain cell */
   p = find(Sq.next);
   Sp = snapshot(p);
   publish(Sq.next); /* Speculative publication */
   verify(Sq,q);
   verify(Sp,p);
   retract(lag);
   lag = Sq.enqid;

      Operation-dependent code that examines fields of
      cell p and may terminate the while loop iteration;

   Sq = Sp;
   q = p; }
```

Each iteration of the `while` loop either advances the scan by one cell, or aborts due to an exception thrown by the `find(Sq.next)` or one of the `verify` calls. The scanning procedure enjoys properties given by the following lemmas.

**Lemma 4** During execution of the operation-dependent statements embedded in the scan procedure, cell `p` is not recycled; and the scan of the queue by the procedure is a correct scan.

**Proof:** We show first that the assertion holds for the initial iteration of the `while` loop, and then use induction to show the assertion holds for remaining iterations.

If the scan procedure reached the operation-dependent statements during the first iteration, neither `verify` nor `find` threw an exception. Therefore `publish(Sq.next)` completed before the two `verify` executions. Let $x_0$ be the id published by this first `publish(Sq.next)` execution. The first `verify` ensures that the `head` cell's `next` field is contained in a publication set at some state during the first `verify` execution. The second verify ensures that $x_0 =$ `p.next` at some state during the second `verify` execution.

We now claim that cell `p` is not recycled until a `retract(`$x_0$`)` executes (this retraction executes in the third iteration, provided no exception is thrown). The claim is shown by contradiction: suppose cell `p` is recycled at some point following the second `verify` execution, but before $x_0$ is retracted. Because `freeable` does not return *true* if it finds a cell's id is published — which is prerequisite to cell recycling — it follows that the recycling of `p` includes a `freeable` execution that begins prior to the publication of $x_0$. For any cell `z`, a `freeable(z)` execution does not return *true* if it finds any `next` reference to `z.enqid`, and since `head.next` $= x_0$ and $x_0 =$ `p.enqid` hold after the publication of $x_0$ (implied by successful `verify` executions), we conclude that the `freeable` execution for recycling of `p` did not observe `head.next` referring to `p`. The code for `freeable` specifies that a cell is checked first to see that it is unchained, and then to see that it is unpublished. Therefore the `freeable` execution for `p`'s recycling determined that `p` was unchained and unpublished *before* the publication of $x_0$; recycling occured by `makeFree after` the two `verify` executions. Throughout execution of `freeable` and `makeFree`, cell `p` has `allocating` status and is therefore not manipulated by any other process allocating cells. Yet between `freeable`'s determination that `p` was unchained and unpublished, some other process must have threaded `p` onto the queue so that the two `verify` statements succeed. It follows from (**G18**) that no cell is enqueued unless it is first allocated, and this contradicts the continous `allocating` status of `p` during the recycling that begins before $x_0$'s publication. Therefore, `p` is not recycled until `retract(`$x_0$`)` executes.

The lemma's assertion is thus established for the first iteration of the scan procedure (the correct scan order in the first iteration is a consequence of the two successful `verify` executions). When the second iteration begins, $x_0 =$ `q.next` which is protected from recycling during the second iteration because $x_0$ remains published (`lag` is `headid` during the second iteration, and the `retract(headid)` executes harmlessly). By reasoning similar to that given for the first iteration, if both `verify` executions succeed in the second iteration, then at some state during the second `verify`, `q.next` $= x_1$ and $x_1 =$ `p.enqid`, and $x_1$ was published prior to this state. Since `q` cannot be recycled so long as $x_0$ remains published, and $x_1$ was published before `q.next` possibly changes from $x_1$ to some other value, cell `p` cannot be recycled until `retract(`$x_1$`)` executes. When the third iteration begins, `lag` $= x_0$ and both $x_0$ and $x_1$ are published.

For the general induction over iterations, suppose $x_i$ and $x_{i+1}$ are published, `lag` $= x_i$, and `q.enqid` $= x_{i+1}$ at the start of iteration $(i + 2)$, numbering iterations starting from zero. The arguments concerning publication and successful `verify` executions can be repeated to show that the scan of cells is correct and that `p` is not recycled during execution of the application-dependent code. □

**Lemma 5** At each state while a process $t$ executes the queue scan procedure, at most two ids are published by $t$ as a result of procedure statements.

**Proof:** Except for the first two iterations, each iteration retracts an id published in a previous iteration, and each iteration publishes only one id. □

# 8 Cell Chaining

The first application of the queue scanning procedure is the `threadCell` routine, which scans the queue to find the end of the chain of cells, rooted at `head`, representing the queue; then `threadCell` chains a specified cell to the end of the queue. The parameter to `threadCell` is a will of type `ENQ`, which implicitly specifies the cell to add to the queue. The presentation in this section only shows partial correctness: invocations of `threadCell` that complete only guarantee safety properties of the queue and the input will, but do not guarantee that an invocation of `threadCell` will successfully complete. Liveness of `threadCell` calls results from a helping strategy, which is presented in Section 12

The presentation of `threadCell` follows. Notice that `threadCell`'s signature declares that an `Error` exception, but none of the statements in the routine appear to throw an exception: recall that the scan procedure use `find` and `verify` routines, which can throw exceptions; such a exception causes immediate termination of a `threadCell` execution and is either handled by the caller of `threadCell` or propagates as an `Error` exception.

```
void threadCell( will w, id e ) throws Error {
   x = cell[t];
   if (w.id != e) return;

   (Scan procedure statements from q = head; up to lag = Sq.enqid;)

      if (w.id == noid || w.type != ENQ) return;
      if (isDone(w,e) || p.enqid == e) return;
      t = w.enqIndex;
      if (e != cell[t].enqid) return;
      if (p.next == nullid) { /* found end of queue */
         v = processOf(p.enqid);
         markDone(will[v],p.enqid); /* ensure previous end of chain's
                                 will has done status prior to chaining */
         if (C&S(p.next,nullid,e)) {
            markDone(w,e); return; } }

   (Scan procedure statements Sq = Sp; and q = p;)

   } }
```

A `threadCell(w,e)` invocation occurs not only by the creator of will `w`, but also by other processes that help operations for the purpose of wait-freedom. Since processors communication only through shared memory and cannot send each other direct signals, it is possible that the creator of `w` finishes the will's operation and the incarnation $w_i$ ends, but some helping process continues to execute `threadCell(w,e)`. To avoid synchronization problems of will recycling during `threadCell` helping, we suppose that the id `e` of will incarnation $w_i$ is published before an invocation of `threadCell(w,e)` and remains published throughout that `threadCell(w,e)` execution. We say that an invocation (and execution) of `threadCell(w,e)` is *pertinent* to $w_i$ if the id of $w_i$ is `e`. Note that, because ids are eventually reused, it is possible for numerous will incarnations to be pertinent to a `threadCell` invocation with parameters `(w,e)`. In fact, it is even possible that the routine that initiates the `threadCell(w,e)` execution is attending to a will different from the one that initially causes `e` to be published, but that will `w` goes through many incarnations and becomes again an enqueue operation with id `e`. This scenario is permissible for the following lemma.

**Lemma 6** At most one `threadCell` execution pertinent to $w_i$ threads the cell of $w_i$.

**Proof:** By contradiction. Suppose one execution `threadCell(w,e)` first succeeds in the execution of the C&S primitive to add `x`, cell of $w_i$, to the queue; and suppose a different `threadCell(w,e)` execution subsequently succeeds in executing C&S primitive adding `x` to the end of the queue. The second `threadCell` did not encounter `x` in its queue scan before it reached the end of the chain. Because the first `threadCell` added `x` and the queue scan follows cells in order, it follows that the second `threadCell` started its queue scan with some cell `after` cell `x` in the queue (since $w_i$ and $x_i$ have the same id, no confusion with different incarnations of `x` is possible while the id remains published). This implies the second `threadCell` began its queue scan after `x` was dequeued and removed from the chain of cells rooted at `head`. Invariants (**G12**) and (**G16**) imply that no cell is removed from the queue unless its will has been marked `done`; but this contradicts the success of the second `threadCell`, which checks will `w` at each iteration of the scan procedure. □

**Lemma 7** If the cell of $w_i$ is chained, then all concurrent and subsequent `threadCell` executions pertinent to $w_i$ either return or throw exceptions.

**Proof:** The proof of Lemma 6 shows that `threadCell` executions beginning after the cell of $w_i$ is chained either find that $w_i$ is `done` or find the cell of $w_i$ in scanning the queue — and at most $D + O(n)$ cells can be scanned before either the cell of $w_i$ is encountered or an exception is thrown. □

## 9 Enqueue Operation

Every enqueue operation applied to the queue is the result of the following `enqueue(z)` procedure, which is charged with allocating a cell, preparing a will, and invoking `threadCell` to enqueue item `z` on the queue.

The `enqueue` routine is the first to use the *try-catch* mechanism for handling exceptions. A statement

`try` { $S$ } `catch Error` { $T$ }

executes $S$, and if $S$ returns without throwing an exception, then $T$ is skipped; however if $S$ throws an exception, $T$ is executed.

One statement in the code of `enqueue` refers to a routine presented in the sequel, `helpout`; for the immediate purpose of explaining the enqueue operation, this statement can be ignored.

```
boolean enqueue( datum z ) {
  helpout();

  /* Allocate a cell & new id or declare queue full */
  c = cell.allocate(id);
  if (c == -1) return false;
  n = cell[c];
  e = genCellid(); publish(e);

  /* Prepare new cell n for enqueue */
  n.enqid = e; n.next = nullid; n.data = z; n.status = active;

  /* Fill in will to allow helping */
  W.id = e; W.type = ENQ; W.enqIndex = c; W.deqEnqid = e;
  W.state = composeID(start,seqnumOf(e));

  s = false;
  while (!s) { /* Repeatedly try to thread cell */
    try { threadCell(W, e); s = true; }
    catch (Error) { retractAllbut(e); } }

  W.id = noid;
  retractALL(); return true; }
```

**Lemma 8** If an invocation `enqueue(z)` returns value *true*, then `z` was added to the end of the queue at some point during the execution of `enqueue(z)`; if `enqueue(z)` returns value *false*, then `z` was not added to the queue and at some state during execution of `enqueue(z)`, the queue contained at least $D$ items.

**Proof:** Lemma 3 implies that if `allocate` does not return a cell, then at some state during its execution, the queue contained at least $D$ items. Once a cell is allocated and a will prepared, an invocation of `threadCell` returns without throwing an error only if it finds the will's cell is chained, the will is `done`, or the will incarnation has ended (the latter is not possible for `enqueue`'s invocation of `threadCell`). □

**Lemma 9** For a will incarnation $w_i$ of type `ENQ`, at most one invocation of `threadCell(w)` successfully chains the cell of $w_i$.

**Proof:** The `enqueue` operation repeatedly invokes `threadCell` only so long as an exception is thrown, therefore only one `threadCell` invocation returns successfully. □

## 10 Dequeue Placement and Dechaining

The enqueue operation relies on the identification of each cell in the queue using the `enqid` field. The scan procedure guarantees correct order of scanning queue elements, so that concurrent `threadCell` executions for the same will incarnation either chain the will's cell, encounter the will's cell in the queue scan, or detect the completion of the enqueue operation via the will's status field. Our intent for a dequeue operation is to repeat this technique using a cell's `deqid` field. To dequeue a cell, the `deqid` of a cell is converted from `noid` to a unique id — the id of the will corresponding to the dequeue operation — and concurrent helping of a dequeue operation can terminate by encountering the will's id in the `deqid` field of a cell during a queue scan. The `deqPlace` routine logically dequeues a cell by atomically assigning the dequeue operation id to the appropriate `deqid` field.

Once a cell's `deqid` field changes from `noid` to the id of the dequeue operation, that cell is *logically dequeued*. The assignment to a cell's `deqid` occurs by the C&S primitive, so each dequeue operation logically dequeues a distinct cell. We define a separate routine to remove a logically dequeued cell from the chain: the `deChain` routine removes the first cell from the chain anchored at `head`, provided that this cell is logically dequeued. Each dequeue operation concludes with a call to `deChain`.

The strategy of using the `deqid` and queue scan to logically dequeue a cell is complicated by the possibility of an empty queue. Suppose, due to helping, that two processes attempt to dequeue a cell on behalf of one will incarnation. If the first process detects the queue empty, but the second finds the queue is non-empty and logically dequeues a cell, then the queue is in an illegal state. To prevent this possibility, the `deqPlace` routine does not directly manipulate a cell's `deqid` field; instead, the cell to be logically dequeued is selected by saving its `enqid` in the will (the `deqEnqid` field), and then routine `deqMark` is called to either finish the logical dequeue or back out from the attempt in the case of a conflict due to a concurrent empty queue detection or competing dequeue operation.

```
void deqPlace( will w ) throws Error {
   d = w.id;

   (Scan procedure statements from q = head; up to lag = Sq.enqid;)

      if (isDone(w,d)) return;
      /* Exit if deqid of will encountered */
      if (p.deqid == d) { deqMark(w); return; }

      /* Attempt to set deqid of cell */
      if (p.deqid == noid) {
         t = p.enqid;
         C&S(w.deqEnqid,d,t);
         deqMark(w);
         return; }

      /* End of chain ⇒ queue empty */
      if (p.next == nullid) {
         if (C&S(w.deqEnqid,d,voidid)) { markDone(w,d); return; }
         else { deqMark(w); throw Error; } }

   (Scan procedure statements Sq = Sp; and q = p;)

   } }
```

As with `threadCell`, we assume that a will `w`'s is published prior to the `deqPlace(w)` invocation and remains published throughout. The `deqMark(w)` invocation either throws an exception or completes the logical dequeue of a cell by assigning a `deqid` field of the cell — with one exception, which is the case of an empty queue. We make the convention that if `w.deqEnqid` is `voidid`, then no cell should be logically dequeued because an invocation of `deqPlace(w)` found the queue to be empty.

**Lemma 10** If a `deqPlace(w)` pertinent to $w_i$ succeeds in assigning `w.deqEnqid=voidid`, then no other `deqPlace` invocation pertinent to $w_i$ logically dequeues a cell.

**Proof:** If some `deqPlace(w)` invocation successfully executes C&S to assign `w.deqEnqid` the value `voidid`, then `w` had previously the id of the will in the `deqEnqid` field, which is published throughout the `deqPlace` execution: this guarantees the C&S applies to the pertinent will for the logical dequeue attempt. Once `w.deqEnqid` has the `voidid` value, no subsequent statements executed change this value throughout the incarnation $w_i$ (this is an invariant, proved from the code of `deqMark`). Changing `w.Enqid` is prerequisite to any logical dequeue pertinent to $w_i$, hence the lemma. □

**Lemma 11** At most one `deqPlace(w)` pertinent to will $w_i$ logically dequeues a cell.

**Proof:** Before any `deqPlace(w)` invocation can logically dequeue the cell for $w_i$, it changes `w.deqEnqid`, and calls `deqMark(w)` to set the `deqid` of the cell to be dequeued. If `deqMark(w)` succeeds in changing `deqid` to be the id of $w_i$, using the C&S primitive, then no other `deqMark(w)` changes this `deqid` field. □

A `deqPlace(w)` execution locates the first cell `x` in the queue that has not yet been captured by another `deqPlace` invocation. Then `deqPlace` "bids" on cell `x` by attempting to set `w.deqEnqid` to the id of `x`. Whether this bidding succeeds or not depends on concurrent `deqPlace` executions, which either detect the queue to be empty or attempt to bid on `x` for a competing dequeue's will. The routine `deqMark(w)` is called to go beyond the bidding stage to capture the desired cell. An invocation of `deqMark(w)` returns if `w` has `done` status (or has progressed to a new incarnation); in case will `w` is not yet done, `deqMark` attempts to complete the bidding process by assigning `x.deqid` the value in `w.id`, and then mark the will as being done.

```
void deqMark( will W ) throws Error {
   m = W.id; t = W.deqEnqid;
   if (t == voidid) { markDone(W,m); return; }

   (Scan procedure statements from q = head; up to lag = Sq.enqid;)

      if (W.type != DEQ || isDone(W,m)) return;
      if (W.deqEnqid != t) throw Error;
      if (p.enqid == t) {
         C&S(p.deqid,noid,m);
         if (p.deqid == m) { markDone(W,m); return; }
         while (W.id == m && W.deqEnqid == t && !isDone(W,m))
            C&S(W.deqEnqid,t,m);
         throw Error; }
      if (p.next == nullid) {
         while (W.id == m && W.deqEnqid == t && !isDone(W,m))
            C&S(W.deqEnqid,t,m);
         throw Error; }

   (Scan procedure statements Sq = Sp; and q = p;)

   } }
```

Unlike statements in other routines that modify will fields, the two C&S statements that access `W.deqEnqid` do not replace a value that is synchronized to the will's id, which is the usual technique for ensuring modification to the proper will incarnation. Similarly, the C&S modification to `p.deqid` does not specify the will id, which presents the danger of logically dequeuing multiple cells on behalf of a given will. The proof of the following lemma addresses these concerns.

**Lemma 12** If an invocation of `deqMark(w)` is pertinent to $w_i$, then $(i)$ any change to a cell's `deqid` is a cell dequeued for $w_i$, and $(ii)$ any change to a will's `deqEnqid` is either $w_i$ or an incarnation of `w` that is enqueue operation, or between incarnations of `w`.

**Proof:** The queue scan procedure ensures that cell `p` will not be recycled during `deqMark`'s processing of `p`. Since the `deqid` field changes only once during a cell incarnation, the change from `noid` to the id of `w` is permanent for `p`'s current incarnation. The two C&S statements that assign to `w.deqEnqid` specify as precondition that `p.enqid` be the value contained in `w.deqEnqid`. If the C&S succeeds, then we claim `w` at that instant cannot be of type `DEQ` for an incarnation $w_j$ other than $w_i$: this situation implies $j > i$, requiring that $w_i$ ended and a new incarnation began. But in order for $w_i$ to end, it must be that cell `p` was either dequeued by $w_i$ or by some other process, so that $w_j$ does not bid on `p`. We conclude that $w_j$ does not assign `p.enqid` to `w.deqEnqid` in this case, which contradicts the assumption of `deqMark` changing the `deqEnqid` field of a later dequeuing incarnation of $w_i$. □

**Lemma 13** At most one execution of `deqMark(w)` logically dequeues a cell for a dequeuing will incarnation $w_i$.

**Proof:** The proof is essentially the same proof as that given for Lemma 6, based on the correct queue scan order, but using the stronger condition that only one value of `w.deqEnqid` exists at any state. □

**Lemma 14** If an invocation of `deqMark` returns without throwing an error, then $w_i$ has logically dequeued one queue cell and $w_i$ has `done` status.

**Proof:** The claim follows from straightforward inspection of the code and the observation that if the will undergoes a new incarnation during `deqMark` execution, then the original incarnation already finished. □

After a cell has been logically dequeued, the data associated with the queue item should be extracted for the dequeue operation and the cell should then be made freeable. We separate these steps so that the implementation is nonblocking. Once a cell has been logically dequeued, another operation (enqueue or dequeue) is able to remove it from the chain so that later cells can be made available for dequeueing and possible recycling. The `deChain` routine is invoked at the end of each dequeue operation and the beginning of every enqueue operation to potentially make logically dequeued cells freeable. Each `deChain` invocation uses a limited form of the queue scan procedure (limited to `head` and its successor in the chain) to secure access to the first cell in the queue. If this cell has been logically dequeued, it can be removed from the chain. We make one exception: if a cell is the last in the chain, it is *not* removed from the chain. Since a subsequent enqueue operation can change the status of a logically dequeued cell from being the last in the chain, each enqueue operation first invokes `deChain` and possibly free up a cell it will need for recycling. The `deChain` routine also attempts to mark the wills (`enqid` and `deqid`) associated with the cell it removes from the chain: this satisfies the requirement in scan procedures that a will be `done` before its cell is removed from the queue.

```
void deChain() {
  try { q = head; Sq = snapshot(q);
    e = Sq.next; p = find(e); Sp = snapshot(p);
    publish(Sq.next); verify(q,Sq); verify(p,Sp); }
  catch (Error) { return; }

  if (p.next == nullcellid || p.next == voidcellid) return;
  if (p.deqid == noid) return;
  a = p.enqid; b = p.next;
  markDone(will[processOf(p.enqid)],p.enqid);
  markDone(will[processOf(p.deqid)],p.deqid);
  if (C&S(q.next,a,b)) C&S(p.next,b,voidid);
  return; }
```

**Lemma 15** If an execution of `deChain` changes `head.next`, then one logically dequeued cell is removed from the queue and the removed cell has `voidid` in its `next` field when `deChain` finishes. If an execution of `deChain` does not remove a cell from the queue, then the either the chain anchored at `head` has no logically dequeued cells, or the chain consists of but one cell, or some other concurrent `deChain` instead removes a logicially dequeued cell.

**Proof:** The `try` construct uses the queue scan logic to protect the first two cells of the queue during modifications, so that cell `p` is not recycled during `deChain` execution. At most one process succeeds in the C&S execution for any particular cell removal, so the second C&S is executed by only one process and therefore replaces `p.next` with `voidid`, unless `p.next` already contains `voidid` (see the code of `dequeue` for the alternative method of assigning `voidid` to this field). □

## 11 Dequeue Operation

The `dequeue` routine resembles the `enqueue` routine: an id for the operation is generated, the will is prepared, and `deqPlace` is invoked repeatedly until it succeeds. Our presentation of `dequeue` returns *false* if the queue was found empty at some state during the operation, and *true* if some item was dequeued (in practice, the data of the dequeued item would also be returned).

```
boolean dequeue( ) {
  helpout();
  e = genCellid();
  publish(e);

  w.id = e; w.type = DEQ; w.deqEnqid = e;
  w.status = composeID(start,seqnumOf(e));

  s = false; a = w.status; b = composeID(done,seqnumOf(e);
  while (!s) {
    try { deqPlace(w);
      C&S(w.status,a,b); s = true; }
    catch (Error) { }; retractALLbut(e); }

  s = false;
  for (i=0; i<numCells; i++)
    if (cell[i].deqid == e) s = true;
  if (!s) { w.id = noid; retractALL(); return false; }
  t = w.deqEnqid; publish(t); p = find(t);
  for (i=0; i<numCells; i++) {
    deChain();
    retractAllbut(t); }
  d = p.next;
  if (d != nullid) C&S(p.next,d,voidid);
  w.id = noid; retractALL(); return true; }
```

**Lemma 16** If `dequeue` returns *false*, then no item is removed from the queue and the queue was empty at some state during the `dequeue` execution; if `true` is returned, then a cell was removed from the queue and that cell's `next` field is `voidid` unless it is the only cell in the queue.

**Proof:** The correctness of the empty queue result follows from the `deqPlace` routine, which terminates either by logically dequeueing exactly one cell, or by logically dequeuing no cells if the queue was found empty. The subsequent scan of cells in `dequeue` either finds the cell logically dequeued or finds no such cell to determine the result (this scan determines the correct results because `e` remains published throughout). Finally, if a cell was logically dequeued, the program repeatedly invokes `deChain` sufficiently many times to imply that the logically dequeued cell has been removed from the chain, perhaps by a competing `deChain` invocation. The final assignment of `voidid` to `p.next` field is necessary because, if a competing `deChain` removed `p` from the chain, its C&S to assign `p.next` may not yet have executed. □

**Lemma 17** The number of logically dequeued cells that are not freeable is at most $1 + 2n$.

**Proof:** One cell can be logically dequeued and not freeable becuase the `deChain` routine does not remove a cell from the queue if it is the only cell in the queue. At any state, each cell that has been logically dequeued either belongs to an execution of a `dequeue` operation that has not finished (and the `deqid` is published, so the cell isn't freeable), or has been removed from the queue chain as well as logically dequeued. At most $n$ cells are in the former category. In the latter category, a cell `x` removed from the queue but having neither of its ids published (`x.enqid` or `x.deqid`) may still not be freeable because some other cell `y` satisfies `y.next` = `x.enqid`. In such a case, we deduce that `x` is not in the queue chain, has been logically dequeued, but its `dequeue` operation has not completed since `x.next` is not `voidid`. There are at most $n$ such cases, so the overall total for logically dequeued but unfreeable cells is $1 + 2n$. □

## 12 Helping and Wait-Freedom

Both `enqueue` and `dequeue` routines begin with calls to `helpout`. The `helpout` routine selects the will of a process and attempts to complete the operation of that will. The selection of the will is round-robin, and the variable `h` used in `helpout` is implicitly a private variable of the calling process that retains its value between invocations of `helpout`.

```
void helpout( ) {
  h = (h + 1) mod n ;
  w = will[h];

  if (w.type == ENQ) {
    s = false; e = w.id; publish(e);
    while (!s) {
      if (w.id != e) return;
      try { threadCell(w); s = true; }
      catch (Error) { }
      retractALLbut(e); } }

  if (w.type == DEQ) {
    s = false; d = w.id; publish(d);
    while (!s) {
      if (w.id != e) return;
      try { deqPlace(w); s = true; }
      catch (Error) { }
      retractALLbut(e); } }

  retractALL(); }
```

The following lemma asserts that our construction guarantees liveness, that is, some operation makes progress in any execution.

**Lemma 18** Any suffix of an execution in which some operation has not finished contains the completion of some operation.

**Proof:** If, in an execution, no exception is thrown in a queue scan routine, then operations complete: this is simple to verify for `threadCell`, and the only exceptions thrown explicitly in `deqPlace` and `deqMark` imply progress by another process on behalf of some operation. And if an execution is thrown during a queue scan, the cause of the execution is a change to the chain anchored at `head`, which only occurs if some operation completed to alter the chain. □

**Lemma 19** Each operation completes in a finite number of steps by the process creating the will for that operation.

**Proof:** By contradiction, suppose some set of operations $W$ do not complete. Then exceptions are thrown during each attempted `theadCell` or `deqPlace` invocation. This implies $(i)$ infinitely many operations complete, and $(ii)$ no helping process completes $w_i$. Therefore each process that helps some will of $W$ forever executes `helpout`. Since infinitely many operations complete, each process not executing `helpout` for some $w_i \in W$ tries to help an operation other than $w_i$, or tries to help a will between incarnations. Observe that a process invokes `helpout` *before* it creates a new will incarnation. Therefore no process gets stuck on behalf of a will belonging to a process helping $w_i \in W$. It follows then, that every process eventually helps some operation in $W$. But if no process makes progress for any operation in $W$, then no further operations complete, which contradicts lemma 18. □

**Lemma 20** The history of queue operations and results is linearizable to a sequential queue.

**Proof:** This lemma is a consequence of the queue scan, which guarantees queue-order in processing of cells. □

## 13 Conclusion

This paper answers the question: can a wait-free queue be implemented using single-word *Compare&Swap* using a linked list representation? This question is of interest where wait-free concurrency of enqueue and dequeue operations is desirable, but queue items and potential queue size cause high overhead for a representation that replicates the entire queue state with each operation. Our construction shows that a linked-list implementation is possible, so that queue items remain stationary in shared memory once they are enqueued.

We have not addressed a number of efficiency concerns in this construction; for instance, the translation from a cell id to the cell location is left unspecified. Results from a Java simulation of the construction suggest that a number of improvements are possible.

## References


[1] M Herlihy. Wait-free synchronization. *ACM TOPLAS 11*: 124–149, 1991.

[2] JD Valois. Lock-free linked lists using compare-and-swap. In *PODC'95*, pp. 214–222, 1995.

[3] JD Valois. Implementing lock-free queues. In *7th Intl Conf on Parallel and Distributed Computing Systems*, pp. 64–69, 1994.

[4] Y Afek, D Dauber, D Touitou. Wait-free made fast. In *Proceedings of the 26th Annual ACM Symposium on the Theory of Computing*, 1995.

[5] MM Michael, ML Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC'96*, pp. 267–275, 1996.

[6] A Israeli, L Rappoport. Efficient wait-free implementation of a concurrent priority queue. WDAG93, pp. 1–17.