Chapter 3

The Shared Memory Protocol

In this chapter, we describe the design of the shared memory protocol, addressing the challenges of data layout and concurrent queue management. We begin by defining the data structures, which illustrate the optimization techniques discussed in the previous chapter. An example of ping-pong-style communication follows, describing the operations performed by the layer to construct, to use, and to destroy channels for communication. We next introduce our solution for managing concurrent message queues—an algorithm that provides performance superior to traditional approaches—and discuss the necessary hardware synchronization primitives and expected performance. After proving a few properties of our algorithm, we conclude with a few comments on the queue semantics that it provides and a brief comparison with aspects of other approaches.

3.1 Data Structures

Local messages pass through an endpoint's shared memory region and move between processors via cache-coherence transactions. The data structures used in these operations—the control and shared memory queue blocks in an endpoint—must hence pay careful attention to how the data are laid out in relation to cache boundaries. At the same time, the structures must reflect the philosophy of finding the minimal critical path for sending and receiving messages through preprocessing of information and function splitting. This section describes the data structures used in our implementation, briefly relating their purpose and illustrating the techniques used to eliminate false sharing and to re-

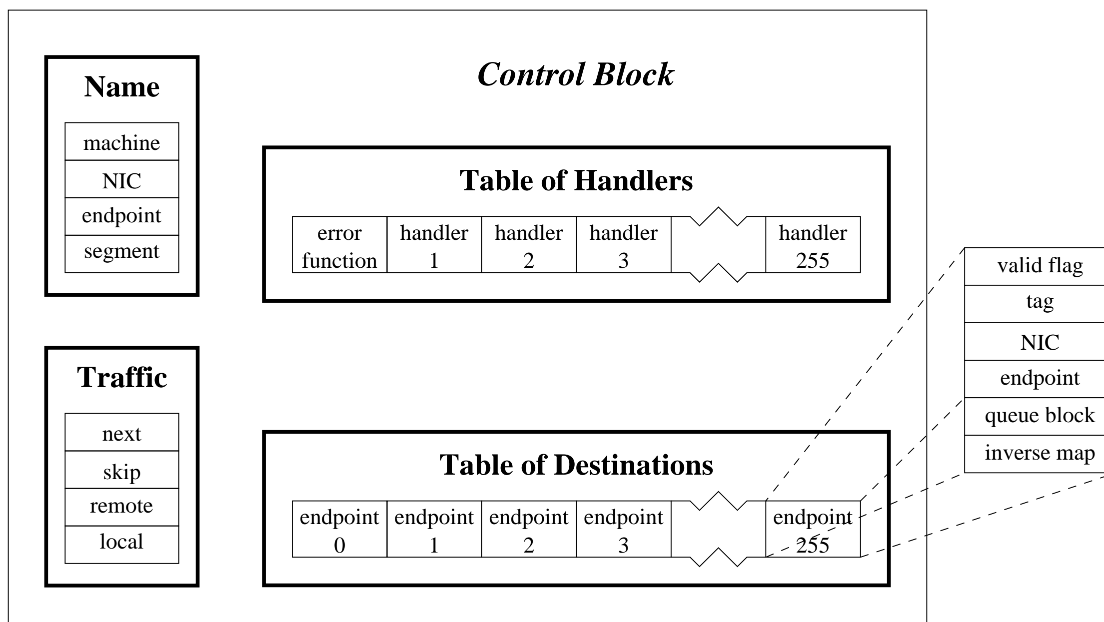

Figure 3.1: Block diagram of a control block. The control block defines an endpoint's physical name, handler functionality, and message destinations. The block also maintains traffic statistics for adaptive polling.

duce the number of coherence transactions. The exact use of the individual fields becomes apparent in the next two sections on operations.

3.1.1 The control block

An endpoint's control block holds information used by the AM-II library functions, as depicted in Figure 3.1. The control block also maintains information about network flow control and caches for network data, but local message operations use only the parts shown.

The name structure represents the physical endpoint name assigned at creation time by the Active Message layer. These names, which are opaque to higher levels of software, provide enough information to find endpoints within the network. The first field holds a machine identifier that uniquely identifies the computer on which the endpoint was created and allows other endpoints to select the appropriate protocol for communication. Our implementation uses the 32-bit value returned by `gethostid()` as the identifier for a given machine. The next two fields locate an endpoint within the network. The NIC field identifies the NIC associated with the endpoint, and any NIC can translate this information into a route. The endpoint number distinguishes between endpoints serviced by the

destination NIC. The last field holds an identifier for the segment in which an endpoint's shared memory queue block resides and allows a process within the same computer to map the queue block into its address space.

The traffic structure maintains estimates of the frequency of local and remote traffic for adaptive polling purposes. Adaptive polling reduces the impact of remote message poll operations on local message performance, as demonstrated in Chapter 7. Two of the fields, remote and local, maintain exponential moving averages of arrival frequency. The skip field holds the total number of calls to the poll operation between remote polls, and the last field, next, holds the actual number of calls remaining.

The table of handlers is simply an array of function pointers indexed by default from 0 to 255. The value 0 is a special case and is used by the Active Message layer to return messages to the sender in the case of a network failure or a security exception.

The table of destinations translates the short integer names used in communication operations into an optimized form of the physical endpoint name. The NIC identifier and endpoint number are unmodified, but the machine and segment identifiers are replaced with a pair of shared memory queue block pointers. For a remote endpoint, these pointers are set to NULL. For a local endpoint, each pointer refers to the queue block of one endpoint in the address space of the process that owns the other. Each destination structure also contains a flag to indicate the validity of the other data in the structure and a copy of the destination endpoint tag for checking access rights.

3.1.2 The shared memory queue block

The shared memory queue block, our extension to the endpoint abstraction, holds local message queues in a System V shared memory segment to allow access by multiple processes within an SMP.[1] A diagram of the shared memory queue block appears in Figure 3.2. A copy of the endpoint tag is used for access control, while two queue structures

[1]The default parameters for System V IPC can be fairly restrictive. In the case of Solaris, for example, a process can map only five segments of at most 1 MB simultaneously. Fortunately, these parameters are readily changed (in Solaris) by appending the following magical lines to the `/etc/system` file on each machine and rebooting:

```
set shmsys:shminfo_shmmax=268435456
set shmsys:shminfo_shmmin=8192
set shmsys:shminfo_shmmni=65536
set shmsys:shminfo_shmseg=8192
```

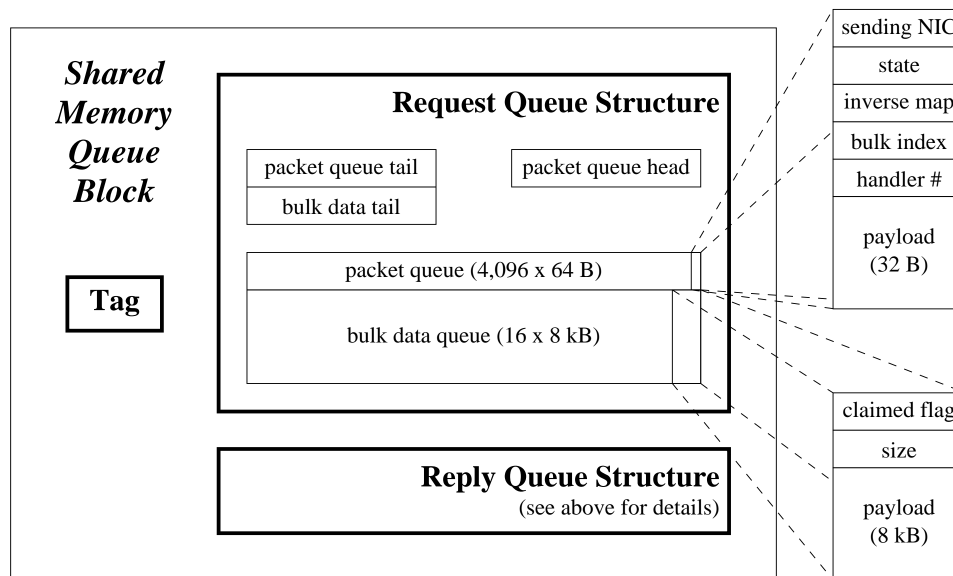

Figure 3.2: Block diagram of a shared memory queue block. Short messages use only the packet queue. Bulk data transfers use the bulk data queue as well.

hold request and reply messages received by the endpoint. Each queue structure further divides into three sections: queue tail information, accessed only by senders; queue head information, accessed only by recipients; and two cyclic data queues, accessed by both senders and recipients. The queues are the packet queue, which contains the handler index and arguments, and the bulk data queue, which holds data for bulk data transfers. Short messages use only the packet queue, while bulk data transfers use both queues.

To eliminate false sharing and thereby reduce the number of memory transactions, the following data occupy distinct L2 cache lines:

- the endpoint tag
- the two queue tails (on one cache line)
- the packet queue head
- each packet
- each bulk data block claimed flag and size field
- each bulk data block payload

In addition to the handler index and arguments, entries in the packet queue contain four other fields: a NIC number, a packet state, an inverse queue block mapping, a bulk data index. The first of these, the NIC number, is used by the Active Message layer to select the appropriate protocol for a reply message (see Section 3.2). The packet state differentiates between short messages and bulk data transfers and serves as the handshake state in transferring data from a sender to a recipient. A claimed flag serves the latter purpose for the bulk data queue. The inverse queue block mapping points to the shared memory queue block of the sending endpoint in the address space of the process that owns the receiving endpoint, enabling reply messages to avoid a potentially expensive lookup operation. The last field, the bulk index, records the association between a bulk data transfer packet and the data itself.

Two factors govern the length of the queues: multiprogramming performance and memory footprint. The long packet queue allows an application to receive a relatively large number of short messages while descheduled. With medium messages, greater insertion overhead makes queue length less important, and we select a shorter queue to keep the memory requirement reasonable: 0.75 MB for the full block.

3.1.3 Queue structure rationale

The shared memory queue block differs significantly from the network queue block in its lack of send queues. The absence arises from a fundamental difference between the methods used to transmit data over the network and within an SMP. In the network case, a sender cannot directly deposit data into memory located across the network, and must instead rely on a third party, such as a Myrinet NIC, to move the data. Within an SMP, the situation is just the opposite: direct access is possible through shared memory, and no third party exists to perform the transfer.

The separation of the request and reply queues avoids a well-known deadlock scenario [CSwAG98]. Consider two endpoints with single queues, each of which is filled with requests. If the endpoints simultaneously issue request messages to one another, both must wait. Neither endpoint can make space in its own queue without handling a request and issuing a reply, but the target queue for the reply may also be full, and a third endpoint may fill the newly created space, in which case the replying endpoint returns to its original

state with a deeper call stack. The stack is finite, but failing to make space results in deadlock.

The problem can also be solved by limiting the total number of requests (not the number per processor) in the network to few enough that a process' stack can absorb them all. This approach does not scale to large systems, however.

As shown in the figure, only two queue structures are allocated for each endpoint. Multiple endpoints can insert messages into these queues concurrently, and the enqueue operation must be carefully designed to ensure that concurrent operations occur atomically with respect to one another. Concurrent access also increases the importance of the queue block layout, as several processors' caches may compete for the data at any point in time.

3.2 Ping-Pong Example

Communication protocols like active messages obtain high performance by optimizing frequent operations, such as sending a message, at the expense of infrequent ones, such as creating an endpoint. In this section, we illustrate the techniques used for this optimization through an example of a two-process ping-pong communication. Given processes A and B within a single SMP, we step through endpoint creation and naming, discuss resource location and access control, cover the actual communication, and conclude with comments on endpoint destruction. We focus in slightly more detail on the communication operations, but defer the most important operation—the insertion of a message into a queue—until Section 3.3.

3.2.1 Endpoint creation

Process A begins by creating endpoint 1. In the creation process, the active message layer allocates space for the three major components of the endpoint and initializes all fields to their default values. The control block is allocated from private host memory. The endpoint is assigned a unique physical name by the NIC, and the traffic statistics are initialized to indicate no traffic and no network skipping. Handler 0 is set to the default error function. All other handlers are set to `abort()`, and all destination endpoint entries are marked as invalid. A shared memory segment is allocated for the shared memory queue block, and the segment identifier is recorded as part of the endpoint's physical name in the control block. The segment is mapped into process A's address space. The tag is set to

a distinguished value that causes rejection of all messages. The queue structure head and tail values are initialized, and the packets and the bulk data blocks are marked as FREE. Backing storage for the network queue block, which contains information similar to that found in the shared memory queue block, is allocated from kernel memory and mapped into process A's address space. The block is dynamically paged onto the NIC when process A inserts its first remote message or performs certain control operations.

After creating the endpoint, process A performs application-level endpoint initialization, installing a reply handler for the ping-pong communication and setting the tag to a value known by process B. AM-II does not specify the methods through which tags are found. We typically preselect one value for each application for simplicity,[2] but one can construct many more complex and secure schemes with or without the use of active messages. As an alternative, process A can set the tag to accept all incoming messages using a second distinguished tag value.

3.2.2 Resource location

Process A has completed the creation of its communication endpoint and is ready to initiate contact with process B. As with tags, AM-II allows an application to define its own methods for the discovery of endpoint names. A simple hash table server suits the purpose for our example. First, process A generates a virtual, application-level name for endpoint 1, "A's endpoint," and declares the virtual-to-physical name translation publicly through the hash table server. Process A then waits for the appearance of "B's endpoint," at which point it learns the physical name of endpoint 2.

3.2.3 Destination preparation

Process A is now prepared to map endpoint 2 into endpoint 1's table of destination endpoints, bringing endpoint 2's shared memory queue block into its address space to permit the direct insertion of messages. As the queue block might be accessible to a number of endpoints, this mapping assumes a high level of trust between processes A and B. Mapping an endpoint thus also implies making an access control decision. If this level of trust is not acceptable, a process can sacrifice performance in favor of security by allocating a separate endpoint for each process with which it communicates.

[2] For reasons beyond the author's ken as a vegetarian, tags are commonly set to the value 0xDEADBEEF.

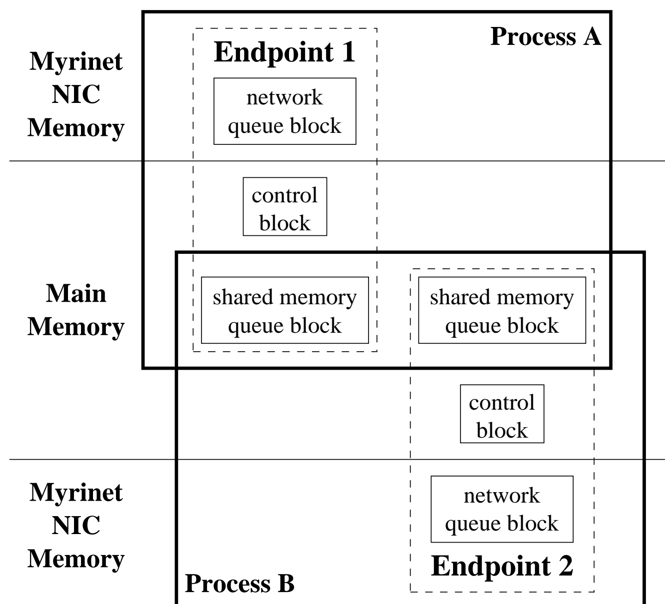

Figure 3.3: Two processes ready to communicate. Each process owns one endpoint and has mapped the shared memory queue block of the other process' endpoint into its address space.

The use of System V shared memory segments as storage implicitly ties access control to the model supported by System V interprocess communication (IPC). The IPC model is identical to the one used by traditional Unix file systems. Each segment has distinct read and write access bits for the owner of the segment, for a Unix group associated with the segment, and for all other users. In future operating systems, we expect shared memory segments to benefit from advances in file system access methods. Access control lists are on the horizon with commercial operating systems, and a more general capability system is perhaps not much more distant. Extensions to the Myrinet NIC driver code offers another alternative: process A might be required to present endpoint 2's tag before being allowed to map endpoint 2's queue block into its address space.

The mapping operation takes as arguments a destination index, an endpoint name, and a tag. The NIC and endpoint numbers from the endpoint name are simply copied into the corresponding fields in the destination entry, as is the tag. Using the name, the active message layer decides whether the endpoint is local or remote. Process A finds that endpoint 2 is local and checks the shared segment identified as a part of endpoint 2's name. After checking a hash table for existing mappings of the segment, process A maps

the segment, adds an entry to the hash table, and records the segment address in the destination entry. The entry is then marked as valid. The last field of the destination entry, the inverse mapping, must be assigned by process B. Process A sends a local message to endpoint 2 requesting that process B map endpoint 1's shared memory queue block and return the segment address to endpoint 1. When the reply arrives, the address is copied into the destination entry, completing the mapping operation. Figure 3.3 shows the situation at this point. Although unimportant to the example in this chapter, control operations such as setting the tag have paged the network control blocks in the figure onto the NIC's.

3.2.4 Ping-pong communication

The communication layer is fully initialized, and process A is ready to communicate with process B. Sending a short request requires only three arguments: the source endpoint, the destination endpoint index, and the handler index. Additional arguments are passed with the message and delivered to the handler as formal parameters. As the first step in sending a message, the active message layer validates the state of the system, checking that all initializations are complete and that the source and destination endpoint are valid.

Once the arguments have been checked, the communication layer decides which protocol to use. Through precomputation in the mapping operation described earlier, this decision is reduced to a single comparison. The shared memory queue block pointer in the destination entry is NULL for a remote endpoint, and non-NULL for a local endpoint.

A poll for incoming messages constitutes the next step. Checking for message arrival on every send operation helps the layer to remain responsive to incoming traffic. The local poll operation need only check the state of the packet at the head of each packet queue. When a message is available, the recipient advances the packet queue head and passes the arguments and, for bulk data transfers, the associated data block, to the appropriate handler routine. After this call returns, the packet and the data block are marked as FREE.

The two steps following the poll are specific to the shared memory protocol. The inverse map field of the destination entry must be copied into a request message to optimize the process of sending a reply. If this field is NULL, the send operation polls until the arrival of the reply carrying the datum. Once a valid inverse mapping has been found, the active message layer checks the tag in the destination entry against the tag in the destination endpoint's shared memory queue block. Messages with invalid tags are returned

immediately via the sending endpoint's handler 0. For the network protocol, the tag check is performed by the NIC associated with the destination endpoint.

At this point, only message insertion remains. We defer the details of this operation to Section 3.3, but note that, as an effect of the insertion, process A changes the state of one of the packets in endpoint 2's request queue structure to indicate the presence of the message. Successful insertion of the message into endpoint 2 marks the end of the send operation, and process A enters a polling loop to await process B's reply.

Process B, which has also completed its initializations, has been waiting in a similar loop and notices the request message when it appears in endpoint 2's queue. The communication layer passes information about the message to the application's handler via an opaque message token, and the application hands this token back to the layer as one of the arguments to a reply operation. The second argument to the reply is a handler index, and the remaining arguments are passed as formal parameters to the reply handler.

A reply requires only protocol selection, polling, and message insertion. Argument checks are skipped—the token and its contents are assumed to be valid. Selecting the appropriate protocol for a reply message again requires only a single comparison. The message packet for remote messages contains the NIC number of the source endpoint; local messages instead fill this field with the NIC number of the destination endpoint. The reply operation compares this value with the replying endpoint's NIC number: equality indicates a local message, and inequality indicates a remote message. As messages from an endpoint to itself are never remote, overloading the meaning of the NIC number field does not cause any problems. After a poll, the reply operation fills a packet in endpoint 1's reply queue structure and returns.

3.2.5 Endpoint destruction

The ping-pong communication is complete, and the two processes are almost ready to terminate. Process A first removes the translation of "A's endpoint" from the hash table server, then destroys endpoint 1, unmapping endpoint 2's shared memory queue block and freeing the memory associated with endpoint 1. The semantics of shared memory segments delays the actual destruction of endpoint 1's segment until all other processes (in the case of our example, process B) have unmapped the segment. Having freed these resources, process A terminates.

3.3 Lock-Free Algorithm

The shared memory message queues permit multiple processes to insert messages concurrently, requiring atomic enqueue operations to prevent interference. Our many-to-one approach distinguishes this thesis from much of the existing work on shared memory message-passing. Managing concurrent access efficiently is the aspect of the shared memory protocol most critical to performance and presents a complex and challenging problem. In this section, we focus on the algorithm used in our active message implementation and provide a few basic notions about alternatives. A more detailed discussion of the latter appears in Chapter 6.

Traditional concurrent access algorithms use critical sections to prevent interference between processes: a process obtains a mutually exclusive lock to enter a critical section, thereby preventing other processes from entering concurrently. A process that stalls or completes its scheduling quantum while holding a lock can delay other processes indefinitely, and a process that waits busily to obtain a *spin lock* can waste processor cycles. Numerous techniques for avoiding these problems appear in the literature, and together are known as preemption-safe locking [MS97]. Such techniques generally require interaction with the operating system and can thus incur significant overhead. We avoid mutual exclusion through the use of a lock-free algorithm. Our algorithm couples synchronization tightly to the data structure and results in superior performance.

The section begins with a description of the algorithm, discussing logical synchronization primitives and their specific use in message insertion. We then consider the actual requirements for hardware support of the algorithm, highlighting alternative primitive implementations. We conclude with performance predictions relative to locking algorithms based on the structure of our algorithm. In Chapter 6, we confirm our predictions on an Enterprise 5000, demonstrating results superior to those obtained with alternative algorithms on both dedicated and multiprogrammed machines using a suite of microbenchmarks and applications. We also compare performance on an SMP with performance on a comparable NOW, illustrating the impact of the faster protocol at the application level.

3.3.1 Message insertion

The lock-free algorithm relies on two synchronization primitives, FETCH&INCREMENT (F&I) and COMPARE&SWAP (CAS), to ensure atomicity with respect to other mes-

```
FETCH&INCREMENT(^address)
    value ← address^
    address^ ← value + 1
    return value

COMPARE&SWAP(^address, old, new)
    if address^ = old
        address^ ← new
        return TRUE
    return FALSE
```

Figure 3.4: Pseudo-code for synchronization primitives. Each operation is atomic with respect to other memory accesses.

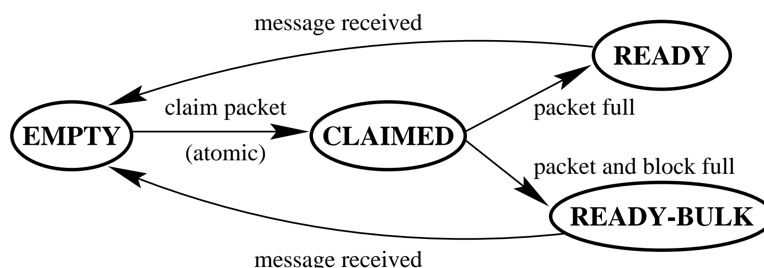

Figure 3.5: State diagram for message packets. The FREE to CLAIMED transition requires instruction-level atomicity for correctness. Unique threads of control perform all other transitions, allowing for non-atomic operations.

sage insertions. The hardware must in turn guarantee that these primitive operations are atomic with respect to all other memory accesses. FETCH&INCREMENT(*address*) adds one to the value at an address and returns the previous value. COMPARE&SWAP(*address, old, new*) compares the value at an address with an expected value, old. If the two values are equal, the operation writes a third value, new, into the address and returns TRUE. Otherwise, CAS returns FALSE. Pseudo-code for these primitives appears in Figure 3.4.

As a message moves through a message packet, the packet cycles through three states, as depicted by Figure 3.5. To enqueue a short message, a sender claims a packet in the destination queue structure with CLAIMPACKET, changing the state of the returned packet from FREE to CLAIMED, then fills in the packet. Claiming a packet involves concurrent access to the queue and must be performed atomically with respect to other claims, but only one sender accesses a packet while filling it. Once a packet is full, the sender changes its state to READY. For bulk data transfers, a sender uses CLAIMBULK to

```
CLAIMPACKET(^q)
    index ← FETCH&INCREMENT(q^.tail) mod Q_LENGTH
    while TRUE
        if COMPARE&SWAP(q^.packet[index].state, FREE, CLAIMED)
            return index
        (back off exponentially and poll)

CLAIMBULK(^q)
    block ← FETCH&INCREMENT(q^.bulk_tail) mod BULK_LENGTH
    while TRUE
        if COMPARE&SWAP(q^.bulk[block].claimed, FREE, CLAIMED)
            index ← CLAIMPACKET(q)
            q^.packet[index].bulk_index ← block
            return index
        (back off exponentially and poll)
```

Figure 3.6: Pseudo-code for our lock-free approach to claiming a packet from a queue q. ClaimBulk claims both a packet and a bulk data block.

claim both a packet and a bulk data block, fills in both, and changes the packet state to READY-BULK. As described earlier, the receiver's poll operation detects when a message at the head of a packet queue is ready for delivery and invokes the appropriate handler. After this call returns, both the packet and the data block are marked as FREE, completing the cycle of states.

Pseudo-code for the claim operations appears in Figure 3.6. The analogue of the critical section in ClaimPacket consists of two steps. First, a sender obtains a packet assignment by atomically incrementing the queue tail using F&I. Next, the sender claims the assigned packet by changing its state from FREE to CLAIMED with CAS. The number of assigned packets may exceed the queue size, in which case multiple senders compete for a single packet in the second step. ClaimBulk takes a similar approach, obtaining a bulk data block and claiming that block before competing for a packet.

If a sender's claim fails, the queue is full, and the sender backs off exponentially to minimize memory transactions. The backoff strategy starts with a 1 microsecond delay and doubles the delay with each failure to a limit of 255 microseconds. If the backoff strategy reaches the delay limit and the claim continues to fail, the sending process relinquishes its processor to allow another process—perhaps the queue's receiver—to make progress.

The backoff strategy can also benefit from interaction with the operating system

```
TEST&SET(^address)
    value ← address^
    address^ ← LOCKED
    return value

FETCH&INCREMENT(^address)
    repeat
        value ← address^
        next ← (value + 1)
    until COMPARE&SWAP(address^, index, next)
    return value
```

Figure 3.7: Pseudo-code for TEST&SET and for a version of FETCH&INCREMENT based on COMPARE&SWAP. This form of FETCH&INCREMENT admits starvation.

scheduler. Rather than simply yielding its processor, a process can sleep until an event occurs. This approach informs the scheduler that the process currently has no useful work and does not wish to compete with the processes responsible for providing it with work. When signaling an event, a process must notify the sleeping process through the kernel. In keeping with active message semantics, event notification is synchronous with respect to control flow within the program. As Arpaci-Dusseau *et al.* found for uniprocessor clusters [ADCM98], an application must utilize this method at all levels in order for the strategy to be fully effective. Due to subtle issues with AM-II kernel support, we did not integrate event notification between the protocols, and use it only with the shared memory protocol in isolation. Events such as waiting for a concurrent queue to drain permitted no obvious signaling criteria, hence we did not use events in the case of full queue backoff. As apparent from the performance data in Chapters 6 and 7, events do improve multiprogramming performance, but the overhead of the additional interaction with the kernel is also non-trivial and can degrade performance on dedicated machines.

The code shown in the figure requires that both Q_LENGTH and BULK_LENGTH be powers of two, but removing this restriction requires only slight modifications.

3.3.2 Hardware support

The lock-free algorithm depends logically on the availability of the F&I and CAS synchronization primitives, but can be implemented with others. Consider, for example, the CAS-based version of F&I shown in Figure 3.7. As the Sparc instruction set lacks the

state	lock	substate
FREE	UNLOCKED	FREE
CLAIMED	LOCKED	FREE
READY	LOCKED	READY
READY-BULK	LOCKED	READY-BULK

Table 3.1: Extended state requirements for a TEST&SET-based variant of the lock-free algorithm. Recall that only the transition from FREE to CLAIMED need be atomic.

F&I primitive, that architecture requires the use of this alternative solution. The approach shown decouples the read and modify components of the F&I from the write component, yet guarantees atomicity by performing the write only when the value remains unchanged. Such constructions have an interesting theoretical history, culminating in Herlihy's definition of universality for synchronization primitives in [Her88]. A *universal* primitive (*e.g.*, CAS) can be used to implement any other primitive in such a way that no process impedes the progress of any other. Although based on CAS, the F&I shown in the figure does not prevent such interference. A successful F&I by one process can cause an F&I attempt by a second process to fail, even to the point of starving the second process. In return for a weaker guarantee, this version of F&I provides much better performance in the common case. We present a version that prevents starvation in Chapter 9.

The specific requirements of the lock-free algorithm also admit replacement of the CAS in the claim step with a simple TEST&SET (T&S) primitive. TEST&SET(*address*) replaces the value at an address with a LOCKED value and returns the previous value, and was perhaps the first synchronization primitive ever suggested. Pseudo-code for T&S appears in Figure 3.7. Use of T&S with the lock-free algorithm requires that the packet state be split into lock and substate values, as shown in Table 3.1. The table lists equivalent values for each possible state. Using T&S, the EMPTY to CLAIMED transition remains atomic: only one process can succeed. The sender implicitly hands the "lock" to the receiver by setting the state to READY or READY-BULK, and the sender must clear the lock after changing the substate to FREE to complete the transition back to the EMPTY state.

3.3.3 Performance notions

Careful scrutiny of the lock-free algorithm allows us to predict its performance relative to locking algorithms for both high and low levels of contention. The key to both predictions is the separation between assigning a packet and claiming it. This separation

increases the number of synchronization primitives performed and results in slightly worse performance in the absence of contention. The same separation, however, results in a much shorter window of vulnerability to contention. The bottleneck step is packet assignment, for which the Enterprise 5000 uses the CAS construction of F&I. This approach is vulnerable to failure only between the completion of the queue tail load and the execution of the CAS, a period covering roughly a tenth of a microsecond on that machine. In contrast, the critical section in a locking algorithm spans at least two cache misses (the queue tail and the packet state) and totals roughly a microsecond. Hence we expect the lock-free algorithm to outperform locking algorithms under contention.

In practice, our lock-free algorithm demonstrates performance superior to both spin locks and preemption-safe locks. The degree of robustness afforded by eliminating locks reduces the impact of interactions with the scheduler yet avoids the high overhead inherent to operating system support. As shown in the detailed performance results in Chapter 6, the lock-free algorithm outperforms other algorithms on both dedicated and multi-programmed machines.

3.4 Theoretical Analysis

In this section, we address several theoretical aspects of our lock-free algorithm. We begin with two proofs of correctness. After making a few basic assumptions about the nature of the system and application program, we prove that the algorithm neither loses messages nor admits deadlock. A discussion of the queue semantics provided by the algorithm follows the proofs, and we conclude the section with commentary on the relative disadvantages of pointer-based algorithms.

3.4.1 Message loss

We first prove that messages do not get lost in a queue—that the receiver cannot wait indefinitely for a message to arrive at the head of the queue while ready messages are present elsewhere. Equivalently, a message inserted atomically into an empty queue must always appear at the head of the queue. In order for this property to hold, the process scheduler must not allow any process to starve. A process that claims the packet at the head of a queue and is subsequently starved by the scheduler blocks all further message reception from that queue.

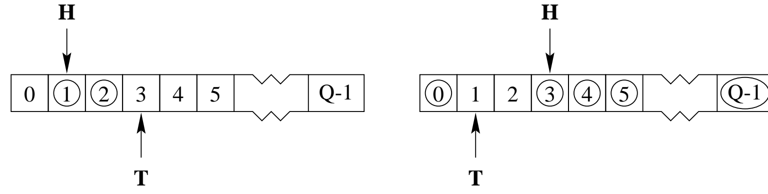

Figure 3.8: Cyclic message queue illustration. Numbers of active packets are circled.

Before formally stating this theorem, we make two definitions. First, recall that when a sender calls ClaimPacket, the algorithm assigns the sender one packet from the queue. Such an assignment is *outstanding* from its time of inception until the receiver frees the assigned packet after handling the message associated with the assignment. Note that we use the term assignment as an abbreviation for packet assignment and do not mean to include the assignment of bulk data blocks. We address bulk data transfers as a corollary to the theorem for short messages. Second, a process scheduler is *starvation-free* if there exists a time quantum $\tau \in \mathbb{R}$, $\tau > 0$ such that a process competing with $P-1$ other processes is guaranteed to execute at least one instruction in any time period of length $P\tau$. In these terms, we state our theorem:

Theorem 3.1 *If governed by a starvation-free scheduler, the packet at the head of a queue with outstanding assignments becomes ready for delivery in bounded time.*

Intuitively, we prove this theorem by establishing that the distribution of outstanding assignments over the queue is fairly even and that the packet at the head of a non-empty queue always has an outstanding assignment. The starvation-free property then guarantees that the head packet becomes ready for delivery in bounded time. We prepare for the formal proof by establishing three useful lemmas.

Consider a cyclic queue consisting of Q message packets numbered 0 through $Q-1$. Let H be the number of the packet at the head of the queue, and define T to be the number of the packet at the tail. A packet i is termed *active* if it falls between the head and the tail of the queue, as illustrated by Figure 3.8. In particular, packet i is active when it satisfies the inequality

$$(i - H) \bmod Q \quad < \quad (T - H) \bmod Q \tag{3.1}$$

Let A be the set of active packets:

$$A \equiv \{i \in \mathbb{Z},\ 0 \leq i < Q \quad | \quad \text{packet } i \text{ is active}\}$$

and note that $|A| = (T - H) \bmod Q$. Our first lemma concerns properties of the set A.

Lemma 3.2 *The head and tail of the queue have certain properties with respect to A. In particular,*

(i) $T \notin A$

(ii) $H = T \ \rightarrow\ A = \emptyset$

(iii) $H \neq T \ \rightarrow\ H \in A$

(iv) $A \neq \emptyset \ \rightarrow\ H \in A$

Proof of Lemma 3.2: Proof of the first three clauses requires only the replacement of variables in (3.1). For (i), replace i with T. (ii) follows from replacement of H with T. To see (iii), replace i with H and recognize that T can differ from H by at most $Q - 1$. (iv) follows directly from (ii) and (iii): $A \neq \emptyset \ \rightarrow\ H \neq T \ \rightarrow\ H \in A$.

Having established these properties, we are ready to state the lemma central to the theorem. Let C_i be the number of outstanding assignments to packet i.

Lemma 3.3 *Outstanding assignments are distributed evenly over a queue, with active packets assigned one time more than packets that are not active. In particular,*

$$\exists n \in \mathbb{Z},\ n \geq 0 \text{ such that}$$
$$\forall i \in \mathbb{Z},\ 0 \leq i < Q,\quad (i \in A \ \wedge\ C_i = n + 1)\ \vee\ (i \notin A \ \wedge\ C_i = n) \tag{3.2}$$

Proof of Lemma 3.3: In a queue's initial state, the C_i are uniformly 0, and $T = H = 0$. By clause (ii) of Lemma 3.2, $A = \emptyset$, thus $n = 0$ satisfies (3.2). We prove the lemma by showing that (3.2) remains true under the operations of packet assignment and message reception. Let unprimed variables represent the state of the queue before an operation and primed variables represent the state after the operation. Furthermore, let n satisfy (3.2) before an operation. The proof then requires that we find a value of n' that satisfies (3.2) after the operation.

First consider packet assignment. By definition, assignments are made to the tail of the queue, advancing the tail cyclicly:

$$\begin{aligned} H' &= H \\ T' &= (T+1) \bmod Q \\ C'_T &= C_T + 1 \\ \forall i \in \mathbb{Z},\ 0 \le i < Q,\ i \ne T, \quad C'_i &= C_i \end{aligned}$$

Notice that by clause (i) of Lemma 3.2, $T \notin A$, thus $C_T = n$ and $C'_T = n+1$.

Two cases are possible. If the assignment does not bring the tail to the head, $T' \ne H$, and we can write

$$(T' - H) \bmod Q = [(T - H) \bmod Q] + 1$$

$$\text{hence} \qquad A' = A \cup \{T\}$$

In this case, packet T becomes active due to the assignment. $C'_T = n+1$ thus implies that $n' = n$ satisfies (3.2).

Next consider the case in which the assignment brings the tail to the head, $T' = H$. By clause (ii) of Lemma 3.2, $A' = \emptyset$, and all of the C'_i must equal n' to satisfy (3.2). A choice of $n' = n+1$ results in $C'_T = n'$, but the same must also hold for the other C_i, *i.e.*, all other packets must be active prior to the assignment. Replacing T with $T' - 1$ in (3.1), we find:

$$\begin{aligned} & (i-H) \bmod Q \quad < \quad (T'-1-H) \bmod Q \quad = \quad (-1) \bmod Q \quad = \quad Q-1 \\ \text{hence} \quad & A = \{i \in \mathbb{Z},\ 0 \le i < Q \quad | \quad i \ne T\} \\ \text{and} \quad & \forall i \in \mathbb{Z},\ 0 \le i < Q,\ i \ne T, \quad C'_i = C_i = n+1 = n' \end{aligned}$$

as desired, proving that n' satisfies (3.2) and thus that (3.2) remains true under packet assignment.

The proof of invariance under message reception is analogous. By definition, messages are received from the head of the queue, advancing the head cyclicly:

$$\begin{aligned} H' &= (H+1) \bmod Q \\ T' &= T \\ C'_H &= C_H - 1 \\ \forall i \in \mathbb{Z},\ 0 \le i < Q,\ i \ne H, \quad C'_i &= C_i \end{aligned}$$

Notice that $(H - H') \bmod Q = Q - 1$, which implies $H \notin A'$; a packet is never active immediately after a message has been received from it.

Two cases are again possible. If the head does not match the tail prior to reception, $H \neq T$, and clause (iii) of Lemma 3.2 implies that $H \in A$. Thus $C_H = n + 1$ and $C'_H = n$. $H \neq T$ also implies

$$\begin{aligned} & (T - H') \bmod Q = [(T - H) \bmod Q] - 1 \\ \text{hence} \quad & A' = A \setminus \{H\} \end{aligned}$$

In this case, the head packet ceases to be active due to the reception. $C'_H = n$ thus implies that $n' = n$ satisfies (3.2).

Otherwise, the head and the tail match prior to reception, $H = T$, which by clause (ii) of Lemma 3.2 implies that $A = \emptyset$. Hence $C'_H = n - 1$ and all other C'_i are equal to n. We require $n > 0$ for message reception to occur, as $n = 0$ in this case implies an empty queue. As $H \notin A'$, a choice of $n' = n - 1 \geq 0$ satisfies (3.2) provided that all packets except H are active after the reception. Writing (3.1) in primed form and replacing H' with $H + 1$, we find:

$$\begin{aligned} & (i - H - 1) \bmod Q \quad < \quad (T - H - 1) \bmod Q \quad = \quad (-1) \bmod Q \quad = \quad Q - 1 \\ \text{hence} \quad & A' = \{i \in \mathbb{Z},\ 0 \leq i < Q \quad | \quad i \neq H\} \end{aligned}$$

as desired, proving that n' satisfies (3.2) and completing the proof of the lemma.

Given an even distribution, we need now merely show that messages appear at the head of the queue first.

Lemma 3.4 *If a queue has outstanding assignments, the packet at the head of the queue has outstanding assignments. In particular,*

$$\forall i \in \mathbb{Z},\ 0 \leq i < Q, \quad C_i > 0 \ \rightarrow \ C_H > 0$$

Proof of Lemma 3.4: Pick $i \in \mathbb{Z}$, $0 \leq i < Q$ such that $C_i > 0$. If $i \in A$, $H \in A$ by clause (iv) of Lemma 3.2 and $C_H = C_i > 0$ by Lemma 3.3. Otherwise, $i \notin A$, and Lemma 3.3 implies that $C_H \geq C_i > 0$, completing the proof.

We are now ready to prove the theorem. Two facts are essential to the final proof: one process always wins the competition to claim a particular packet, and the code for claiming and filling a packet requires a finite number of instructions when successful.

Proof of Theorem 3.1: Assume that a queue has outstanding assignments at some time t and that the packet H at the head of the queue is not ready for delivery at that time. By Lemma 3.4, we know that H has at least one outstanding assignment, $C_H > 0$. Consider the code path for inserting a message into H after the assignment step. At that point, bulk data transfers have already obtained a bulk data block and are inside of ClaimPacket, as are short messages. A sender must back off from any previous claim failure, claim H, fill H and possibly a bulk data block, and change H's state. If the packet claim succeeds, as it must for exactly one sender when H is free, the code does not branch indefinitely (recall that the backoff period is bounded). Let N denote the maximum number of instructions executed by the successful process. Let P be the maximum number of processes that the system supports, and let $P\tau$ be the length of the time period in which a process is guaranteed to execute at least one instruction. As the scheduler is starvation-free, $P\tau$ is finite. The sender that succeeds in claiming H thus marks H as ready for delivery no later than time $t + NP\tau$, proving the theorem.

Theorem 3.1 holds for all messages, including bulk data transfers, once they have been assigned packets. Proving that a bulk data transfer waiting to claim a bulk data block also implies the arrival of a message takes a small additional effort. A bulk data block assignment is outstanding from its time of inception until the receiver frees the assigned block after handling the message associated with the bulk data block assignment.

Corollary 3.5 *If governed by a starvation-free scheduler, the packet at the head of a queue with outstanding bulk data block assignments becomes ready for delivery in bounded time.*

Proof of Corollary 3.5: Let p be a process with an outstanding bulk data block assignment. If p has yet to claim its assigned bulk data block, it attempts to do so in bounded time under a starvation-free scheduler. If the claim fails, let p' be the process that last successfully claimed the block assigned to p. Otherwise, let $p' = p$. If p' has already been assigned a packet, the claim must be outstanding since the bulk data block associated with p' has not been received (and hence neither has the assigned packet), and the proof is done.

Otherwise, a packet is assigned to p' in bounded time under a starvation-free scheduler, and the packet at the head of the queue becomes ready in bounded time by Theorem 3.1, completing the proof.

3.4.2 Communication deadlock

We next prove that the lock-free algorithm cannot result in deadlock within the communication layer. Intuitively, deadlock implies that a program reaches a state in which none of its processes is capable of making forward progress. A synchronous communication layer, however, can guarantee only that the processes do not deadlock within the layer. Avoiding deadlock in general also requires that each process remain responsive to communication. For the lock-free algorithm, we must demonstrate that processes can not all block indefinitely within the algorithm without sending or receiving messages. As processes only block within the algorithm when waiting to insert into a full queue, we use the full queue condition as the basis for a more formal construction.

Let $F(t)$ be the set of processes waiting to insert messages into full queues at time t, and let $T_f(t)$ be the target process for $f \in F$—the process that receives messages from the full queue on which f is blocked trying to insert a message. Define the insertion graph $G(V(t), E(t))$ to be the directed graph with vertices consisting of $F(t)$ and the set of target processes $T(t)$ and with edges from each member of $F(t)$ to its target process:

$$
\begin{aligned}
T(t) &\equiv \{f \in F(t) \quad | \quad T_f(t)\} \\
V(t) &\equiv F(t) \quad \cup \quad T(t) \\
E(t) &\equiv \{(f, g) \in F(t) \times T(t) \quad | \quad g = T_f(t)\}
\end{aligned}
$$

In light of this graph construction, we can provide a meaningful definition of deadlock within the message layer: a *deadlock* occurs when no process in $V(t) \neq \emptyset$ can send or receive a message in bounded time. We define deadlock in terms of $G(V(t), E(t))$ to avoid speculation about higher levels of software; processes outside of $V(t)$ may be capable of progress, but we cannot assume that such progress is possible.

Processes must remain responsive to the communication layer. Formally, a process is *responsive* if there exists a $\tau \in \mathbb{R}, \tau > 0$ such that the process sends a request or checks for message arrival on each endpoint at least once during any time period in which execution time outside of the communication layer totals at least τ. The actual time required depends

on the time allocated to the process by the scheduler, but is bounded under a starvation-free scheduler. Note that sending a request causes the communication layer to check for incoming messages. We are now ready to state our theorem.

Theorem 3.6 *If all processes are responsive, the lock-free algorithm cannot result in deadlock.*

Note that the theorem does not require that the scheduler be starvation-free. If a scheduler starves a process and thereby prevents all processes from making progress, the result is starvation, not deadlock. However, in proving this theorem, we first show that some process does make progress under a starvation-free scheduler, which in turn implies that the process can make progress, independent of whether or not it actually does so.

The lock-free algorithm avoids deadlock by polling while backing off from a full queue, polling for all messages when sending a request, but polling only for replies when sending a reply. Although this additional splitting of the network substantially complicates the proof, it is also necessary to avoid deadlock, as mentioned in Section 3.1.3. The added difficulty revolves upon the case in which a process sending a request targets the full queue of a process that blocked while sending a reply. The latter process polls only for replies, but we cannot guarantee that a reply becomes available in bounded time without considering longer chains of dependencies. We begin the proof by establishing an evolutionary property of the target process relationship: a blocked process and its target process either make progress or reach the send-request, poll-reply state in bounded time.

Lemma 3.7 *Let $f \in F(t)$ be a process blocked on a full queue at time t, and let the responsive process $g = T_f(t)$ be f's target process. If governed by a starvation-free scheduler, one of the following holds:*

(i) Some process in $V(t)$ makes progress in bounded time by either sending or receiving a message.

(ii) f is blocked on g's request queue at time t, and g blocks on a full reply queue in bounded time.

Proof of Lemma 3.7: First consider the case in which $g \notin F(t)$, and assume for now that g does not block on a full queue before receiving a message. As f has an outstanding

bulk data block or packet assignment to a queue owned by g and the scheduler is starvation-free, Theorem 3.1 and Corollary 3.5 imply that a message becomes ready for g to receive in bounded time. As g is responsive and the scheduler is starvation-free, g receives the message within bounded time after it becomes ready, resulting in case (i). However, g can block on a full queue before receiving a message, *i.e.*, also in bounded time. If any other process in $V(t)$ sends or receives a message before g blocks, the result is case (i). We can thus assume without loss of generality that G does not change until g blocks at a time t' with $V(t') = V(t) \cup \{T_g(t')\}$ and $E(t') = E(t) \cup \{(g, T_g(t'))\}$.

Next consider the case in which g is blocked (at either time t or time t') on a full request queue. Assume for now that g neither succeeds in inserting its request nor blocks on a full reply queue (while handling a request) before receiving a message. Given the assumptions, a message becomes ready for g to receive in bounded time. While blocked on a full request queue, g polls for messages and hence receives the message within bounded time after it becomes ready, resulting in case (i). If g succeeds in inserting its request before receiving a message, the result is also case (i). Otherwise, g blocks on a full reply queue in bounded time, and we make the same assumptions as before about the progress of other processes.

Thus all cases evolve in bounded time to the one in which g is blocked on a full reply queue. If f is blocked on g's request queue, case (ii) results immediately, hence assume instead that f is blocked on g's reply queue. Also assume for now that g does not succeed in inserting its reply before receiving a reply. Given the assumptions, a reply becomes ready for g to receive in bounded time. While blocked on a full reply queue, g polls for replies and hence receives the reply within bounded time after it becomes ready, resulting in case (i). If g succeeds in inserting its reply before receiving a reply, the result is the same, thus completing the proof.

The lemma is the framework for the rest of the proof, the essence of which lies in applying the lemma twice to locate a pair of processes that cannot enter the send-request, poll-reply relationship. We now prove the theorem.

Proof of Theorem 3.6: Assume without loss of generality that the scheduler is starvation-free; a scheduler that starves processes does not change the fact that a process can make progress (when scheduled). Let $G(V(t), E(t))$ be the insertion graph at some time t. If

$F(t) = \emptyset$, $V(t) = \emptyset$, and we are done. Otherwise, pick $f \in F(t)$ and let g denote its target process, $g = T_f(t)$. If f is blocked on g's reply queue, Lemma 3.7 guarantees that some process in $V(t)$ makes progress in bounded time, and we are done. Thus assume that f is blocked on g's request queue. Lemma 3.7 then guarantees either that some process in $V(t)$ makes progress or that g blocks on a full reply queue in bounded time. Assuming that the latter occurs and that it occurs at a time t', let $h = T_g(t')$. As g is blocked on h's reply queue at time t', Lemma 3.7 guarantees that some process in $V(t')$ makes progress in bounded time from t', which in turn occurs in bounded time from t, completing the proof.

As a final comment on deadlock issues, we note that the order defined by CLAIM-BULK—reserving space in the bulk data queue before competing for space in the packet queue—circumvents a deadlock scenario that arises with the reverse order. Consider a process performing a bulk data transfer and holding the only outstanding assignment to the packet at the head of a packet queue. If the associated bulk data queue is full, the process cannot complete its message insertion, but neither can the queues' owner receive any messages holding bulk data blocks until it receives the message at the head of the packet queue. In terms of the proofs, this reversed order of operations violates the assertion in the proof of Theorem 3.1 that a process with an outstanding assignment cannot delay indefinitely before marking its packet as ready; in this case, the process must wait for a bulk data block to become available, but that event never happens. Once messages can be lost in the queue, deadlock is straightforward.

3.4.3 Queue semantics

The lock-free algorithm gives rise to several subtle issues of semantics. The astute reader may have noticed that we avoided a detailed discussion of starvation due to the algorithm itself. In fact, the algorithm does not prevent such scenarios; rather it guarantees only that some process makes progress. A process that consistently loses the competition for its assigned packet, for example, may never insert its message into a queue. In practice, an individual packet is rarely assigned to more than one process simultaneously, and starvation does not occur.

Starvation is nevertheless an inherent aspect of all algorithms that compete for central control information using synchronization primitives that provide no guarantee of

fairness. The prevalence of universal primitives on modern machines thus has a subtle drawback in that it has kept primitives that guarantee fairness from appearing in some instruction set architectures (*e.g.*, the Sparc). As with simpler non-universal primitives such as T&S, the number of times that a process can fail a universal CAS operation is not generally bounded by hardware. Whereas a primitive such as F&I typically guarantees fairness, a version of F&I constructed from CAS admits starvation. Although not a significant problem for today's machines, the level of contention and the potential for starvation grow with the number of processors in an SMP, and the problem might become significant in the future.

A second interesting aspect of the lock-free algorithm is its lack of a first-in, first-out (FIFO) property as defined in the literature. According to Gottlieb *et al.* [GLR83], a FIFO concurrent queue must ensure the following: "If insertion of a data item p is completed before insertion of another data item q is started, then it must not be possible for a deletion yielding q to complete before a deletion yielding p has started." Herlihy and Wing's notion of linearizability [HW90] leads to a similar definition. Our lock-free algorithm does not obey this rule. Consider processes A and B simultaneously inserting messages into a two-element queue. Assign the first packet to process A and the second to process B, then assume that process A is descheduled before claiming the packet. Process B fills the second packet and begins another insertion. The queue wraps and assigns the first packet again, and B fills this packet as well. At this point, the first packet contains B's second message and the second packet contains B's first message. When the receiver looks for incoming messages, the second message is delivered first, violating the FIFO property. As active messages do not guarantee in-order delivery, the lack of a FIFO property does not make the lock-free algorithm incorrect. Nevertheless, a simple extension considered in Chapter 6 suffices to achieve FIFO semantics.

Finally, one might question whether the lock-free algorithm is truly free of locks. The three-state handshake between sender and receiver can be viewed as a form of fine-grained locking on individual message packets. Despite this fact, we chose to use the term "lock-free" to highlight the avoidance of explicit critical sections and the significantly reduced likelihood of contention between processes.

3.4.4 Pointer-based comparison

The previous section discussed the subtleties of the lock-free algorithm. We now discuss a few hazards that the lock-free algorithm avoids by design, in particular by manipulating values rather than names, *i.e.*, pointers. Pointers can lead to problems with improper success of operations, increased complexity for potentially interleaved operations, and difficulties with reusing memory. The lock-free algorithm operates on a tail index and on packet states, using no pointers or other indirect references to data. We discuss problems with pointers and their solutions to highlight the advantages of the lock-free algorithm in terms of programming complexity.

The ABA Problem

Pointer-based algorithms that make use of the CAS primitive must be aware of the ABA problem, which arises because of a separation between the memory value manipulated by CAS, often the name of a datum, and the object semantically intended for manipulation, the datum itself or an entire data structure. In the ABA problem, a process reads the name A and proceeds to operate on that value. Before the process can perform its CAS operation, a second process replaces A with B, recycles the datum referenced by A, and replaces B with A. The meaning of A at this point is clearly different from its original meaning, but the CAS operation recognizes only the name A and the first process' CAS operation "succeeds."

Figure 3.9 demonstrates the result for a linked list. In section (1), the list contains the elements A and B. A process reads A and it's *next* pointer, B. In section (2), a second process has removed A, leaving only B in the list. In section (3), further operations have moved B from the list to a free list of cells, while A has been recycled and returned to the list. At this point, the first process performs its CAS. The CAS compares the head of the list with A and "succeeds," placing a long string of garbage beginning with B onto the list, as shown in section (4).

Apart from avoiding the use of indirection, which is not always practical, two solutions exist for the ABA problem. The more common of the two is an algorithmic approach: each pointer is extended with an epoch number, and the CAS operation is applied to both the pointer and the epoch number simultaneously, incrementing the epoch with each successful CAS. If the hardware supports CAS on data significantly wider than

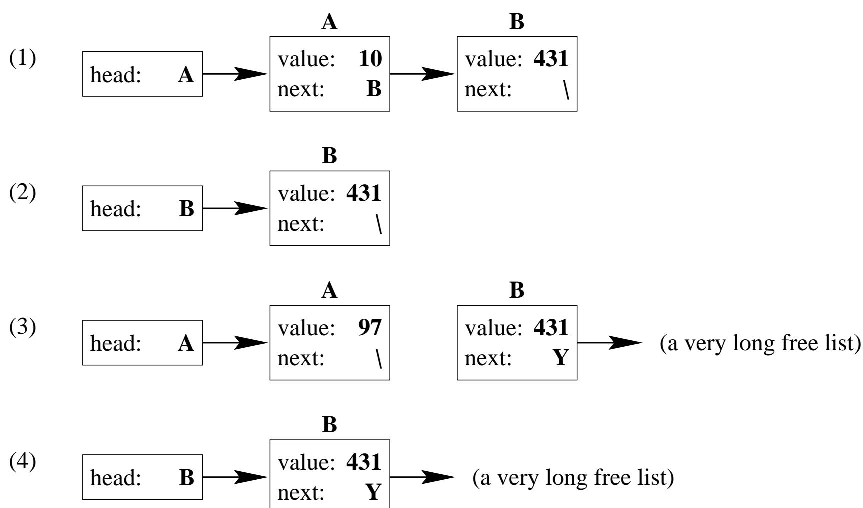

Figure 3.9: Illustration of the ABA problem on a list. See the text for details.

a pointer, then this approach is very unlikely to fail in practice. On machines without such support, however, an algorithm must make use of reduced pointers (either indices or truncated pointers) while retaining enough bits in the epoch number to make the ABA problem effectively impossible.

The second approach to the ABA problem makes use of a slightly different set of hardware primitives known as LoadLinked (LL) and StoreConditional (SC). The initial read operation in an algorithm is replaced with an LL primitive, and the CAS operation is replaced with an SC primitive. SC is guaranteed to fail if any other processor has changed the value at the linked address, so that success of the SC operation is in many cases equivalent to an absence of intervening changes to the data structure (and certainly to the location being changed).

Inconsistent Versions

A second problem arises when a process holds a stale reference to an object that is no longer an active part of the data structure. For correctness, pointer-based lock-free algorithms generally avoid non-atomic changes to the data in a concurrent structure, but such restrictions do not apply to objects once they have been retired from use. In fact, keeping old objects around for reuse is a fairly common method of avoiding dynamic memory overhead (and a significant contributor to the likelihood of the ABA problem).

Herlihy provides one solution to this problem in [Her93]. In his solution, algorithms maintain consistency information with each object. Before using an object, an operation makes a copy of the object, then checks the copy for consistency. Operation must also mark objects as inconsistent before manipulating them. The object becomes consistent again after the operation completes.

Typed Memory

The vagaries of schedulers and virtual memory can impose much more significant inconsistencies on a pointer-based algorithm. Once used in a concurrent data structure, a pointer to an object X of type Y may persist indefinitely in a register of some descheduled process. If the memory used by X is later reclaimed and recast as an object of type Z, the process can return to find the object Y completely incomprehensible. Dynamically-typed objects provide one possible solution to this problem, although concurrent operations must check the type very frequently in such a system, possibly after every memory read, and must also verify the type atomically with each write. Alternatively, Greenwald and Cheriton suggest in [GC96] that memory used for concurrent objects be "type-stable," meaning that the memory used for object X can only be reclaimed if the program can guarantee that no outstanding references to X exist.

Data Manipulation

None of the problems just discussed applies to our lock-free algorithm. The packet queue remains fixed in a single memory location, allowing the lock-free algorithm to treat packet assignments as data rather than pointers. The packet states also have a small number of fixed values, and the lock-free algorithm merely performs transitions within the state diagram. Many lock-free and non-blocking algorithms manipulate linked data structures; the data structures move from point to point in memory, leading to problems with detecting changes (the ABA problem), recognizing stale data (inconsistent versions), and avoiding operations on random values (typed memory).