An $\Omega(n \log n)$ Lower Bound on the Cost of Mutual Exclusion

Rui Fan *
MIT Computer Science and AI Laboratory
Cambridge, MA 02139, USA
rfan@theory.csail.mit.edu

Nancy Lynch
MIT Computer Science and AI Laboratory
Cambridge, MA 02139, USA
lynch@theory.csail.mit.edu

ABSTRACT

We prove an $\Omega(n \log n)$ lower bound on the number of non-busywaiting memory accesses by any deterministic algorithm solving n process mutual exclusion that communicates via shared registers. The cost of the algorithm is measured in the *state change* cost model, a variation of the cache coherent model. Our bound is tight in this model. We introduce a novel information theoretic proof technique. We first establish a lower bound on the information needed by processes to solve mutual exclusion. Then we relate the amount of information processes can acquire through shared memory accesses to the cost they incur. We believe our proof technique is flexible and intuitive, and may be applied to a variety of other problems and system models.

Categories and Subject Descriptors

F.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms

Theory, Algorithms

Keywords

Mutual exclusion, Time complexity, Lower bound techniques, Information theory

1. INTRODUCTION

In the mutual exclusion (*mutex*) problem, a set of processes communicating via shared memory access a shared resource, with the requirement that at most one process can access the resource at any time. Mutual exclusion is a fundamental primitive in many distributed algorithms, and is also a foundational problem in the theory of distributed computing. Numerous algorithms for solving the problem in a variety of cost models and hardware architectures have been proposed over the past four decades. In addition, a number of recent works have focused on proving lower bounds for the cost of mutual exclusion. The *cost* of a mutex algorithm may be measured in terms of the number of memory accesses the algorithm performs, the number of shared variables it accesses, or other measures reflective of the performance of the algorithm in a multicomputing environment. In this paper, we study the cost of a mutex algorithm using the *state change* cost model, a simplification of the standard *cache coherent* model, in which an algorithm is charged only for performing shared memory operations causing a process to change its state. Let a *canonical execution* consist of n different processes, each of which enters the critical section exactly once. We prove that any deterministic mutex algorithm using registers must incur a cost of $\Omega(n \log n)$ in some canonical execution. This lower bound is tight, as the algorithm of Yang and Anderson [13] has $O(n \log n)$ cost in all canonical executions with our cost measure. To prove the result, we introduce a novel technique which is *information theoretic* in nature. We first argue that in each canonical execution, processes need to cumulatively acquire a certain amount of information. We then relate the amount of information processes can obtain by accessing shared memory to the cost of those accesses, to obtain a lower bound on the cost of the mutex algorithm. Our technique can be extended to show the same lower bound when processes are allowed access to comparison-based shared memory objects, in addition to registers. Furthermore, we believe that with some modifications, we can use the techniques to prove an $\Omega(n \log n)$ lower bound on the cost of some canonical execution in the cache coherent model. A report on these results is in preparation.

We now give a brief description of our proof technique. Intuitively, in order for n processes to all enter the critical section without colliding, the "visibility graph" of the processes, formed by adding a directed edge from each process that "sees" another process, must contain a directed chain on all n processes. Indeed, if there exist two processes, neither of which sees the other, then an adversary can make both processes enter the critical section at the same time. To form a directed visibility chain, the processes must all together collect enough information to compute a permutation $\pi \in S_n$. Such a permutation takes $\Omega(n \log n)$ bits to specify. We show that in some canonical executions, each time the processes perform some memory accesses with cost C, they gain only $O(C)$ bits of information. This implies that in some canonical executions, the processes must incur $\Omega(n \log n)$ cost. To formalize this intuition, we construct, for any permutation $\pi \in S_n$, an execution α_π in which a process ordered lower in π does not see any processes ordered higher

*This work was supported by USAF-AFRL #FA9550-04-1-0121, NSF Grant CCR-0121277, and a grant from Cisco Systems.

PODC'06, July 22-26, 2006, Denver, Colorado, USA.

in π[1,2]. In this execution, we can show that the processes must enter their critical sections in the order specified by π. This implies that α_π must be different for different π, so that the set $\{\alpha_\pi\}_{\pi \in S_n}$ contains $n!$ different executions. Then, we show that if the cost of execution α_π is C_π, we can *encode* α_π, that is, produce a string that uniquely identifies α_π, using $O(C_\pi)$ bits. But since it takes $\Omega(n \log n)$ bits to uniquely identify an element from a set of size $n!$, some execution must have cost $C_\pi = \Omega(n \log n)$.

The remainder of our paper is organized as follows. In Section 2, we describe related work on mutual exclusion and other lower bounds. In Section 3, we formally define the mutual exclusion problem and the state change cost model. We give a detailed overview of our proof in Section 4. In Section 5, we present an algorithm that, for every $\pi \in S_n$, produces a different execution α_π with some cost C_π. We show in Section 6 how to encode α_π as a string E_π of length $O(C_\pi)$. In Section 7, we show E_π uniquely identifies α_π, by presenting a decoding algorithm that recovers α_π from E_π. Our main lower bound result follows as a corollary of this unique decoding. Lastly, in Section 8, we summarize our results and techniques, and discuss some future work and open problems.

2. RELATED WORK

Mutual exclusion is a seminal problem in distributed computing. Starting with Dijkstra's work in the 1960's, research in the area has progressed in response to, and has sometimes driven, changes in computer hardware and the theory of distributed computing. For interesting accounts of the history of this problem, we refer the reader to the excellent book by Raynal [12] and survey by Anderson, Kim and Herman [4].

The performance of a mutual exclusion algorithm can be measured in a variety of ways. An especially relevant measure for modern computer architectures is *memory contention*. In [1], Alur and Taubenfeld prove that for any nontrivial mutual exclusion algorithm, some process must perform an unbounded number of memory accesses to enter its critical section. This comes from the need for some processes to busywait until the process currently in the critical section exits. Therefore, in order for a mutex algorithm to scale, it must ensure that its busywaiting steps do not congest the shared memory. *Local-spin* algorithms were proposed in [8] and [11], in which processes busywait only on *local* or *cached* variables, thereby relieving the gridlock on main memory. Local-spin mutex algorithms include [13], [10] and [3], among many others. In particular, the algorithm of Yang and Anderson [13] performs $O(n \log n)$ remote memory accesses[3] in an execution in which n processes each complete their critical section once. The state change cost model we propose discounts certain busywaiting steps by charging an algorithm only for memory accesses that change a process's state. Yang and Anderson's algorithm also has $O(n \log n)$ cost using the state change model.

A number of lower bounds exist on the memory complexity for solving mutual exclusion [6]. Recently, considerable research has focused on proving time complexity (number of memory accesses) lower bounds for the problem. Cypher [7] first proved that any mutual exclusion algorithm must perform $\Omega(n \frac{\log \log n}{\log \log \log n})$ total remote memory accesses in some canonical execution. An improved, but non-amortized lower bound by Anderson and Kim [2] showed that *some* process must perform at least $\Omega(\frac{\log n}{\log \log n})$ remote memory accesses. However, this result does not give a nontrivial lower bound for the *total* number of remote accesses performed by all the processes. The techniques in these papers involve keeping the set of processes contending for the critical section "invisible" from each other, and eliminating certain processes when they become visible. Our technique is fundamentally different in that we do not eliminate processes, nor do we try to keep all processes invisible from each other. Intuitively, we show that in order for n processes to solve mutual exclusion, they must collectively gather enough information to compute a directed "visibility chain", in which each process sees the next process in the chain. We then relate the amount of information processes can acquire with the cost they must incur to obtain that information. Information-based arguments have also been used by Jayanti [9] and Attiya and Hendler [5], among others, though in quite different forms.

3. MODEL

In this section, we define the formal model for proving our lower bound. We first describe the general computational model, then define the mutual exclusion problem, and the state change cost model for computing the cost of an algorithm.

3.1 The Shared Memory Framework

In the remainder of this paper, fix an integer $n \geq 1$. A *system* consists of a set of *processes* $p_1, \ldots, p_n$, and a collection L of *shared variables*. A shared variable consists of a *type* and an *initial value*. In this paper, we restrict the types of all shared variables to be multi-reader multi-writer registers. Each process is modeled as a deterministic automaton, consisting of a *state set*, an *initial state*, and a deterministic *transition function* that computes a *step* (*e.g.*, a memory access) for the process to execute based on its current state. We write $\delta(s, i)$ for the transition function of process p_i, where s is a state of p_i. For every $i \in [n]$, a *read step* of p_i is $\mathsf{read}_i(\ell)$, where $\ell \in L$, and represents a read by process p_i on register ℓ. We let $own(\mathsf{read}_i(\cdot)) = i$ be the process performing the read. A *write step* of p_i is $\mathsf{write}_i(\ell, v)$, where $\ell \in L, v \in V$, and represents a write of value v by process p_i on register ℓ. Here, V is some arbitrary fixed set. We let $own(\mathsf{write}_i(\cdot,\cdot)) = i$. We define $val(\mathsf{write}_\cdot(\cdot, v)) = v$ to be the value written by a write step. Let e be a step. We define $type(e)$ to be `R` if e is a read step, and `W` if e is a write step. We say that a step $\mathsf{read}_\cdot(\ell)$ or $\mathsf{write}_\cdot(\ell, \cdot)$ *accesses* register ℓ. An *algorithm* specifies a process automaton p_i, for each $i \in [n]$.

A *system state* is a tuple consisting of the states of all the processes and the values of all the registers. We assume that all systems have a default *initial state* s_0, consisting of the initial values of all the registers and the initial states of all the processes. An *execution* consists of a (possibly infinite) alternating sequence of system states and process steps. That is, an execution is of the form $s_0 e_1 s_1 e_2 s_2 \ldots$, where each s_i is a system state, and each e_i is a step by some process. For any execution α, we define

---

[1] A process is *ordered lower* in π if it appears earlier in π. For example, if $\pi = (4213)$, then 4 is ordered lower in π than 1.

[2] Actually, we construct an *equivalence class* of executions, all of which have this property.

[3] A remote memory access is the unit of cost in local spin algorithms.

$\alpha(t) = s_0 e_1 s_1 \ldots e_t s_t$ to be the length t prefix of α[4]. We let the *projection* of α on a process p_i be the sequence $\alpha|i$ consisting only of the states and steps of p_i. If α is a finite execution, we define $st(\alpha)$ to be the final system state of α, and, for $i \in [n]$, $st(\alpha, i)$ to be the state of process p_i in $st(\alpha)$. We say an execution β is an *extension* of α if β contains α as a prefix. For convenience, we sometimes represent an execution simply as a sequence of process steps, $e_1 e_2 \ldots$. Since we assume that the system has a unique initial state, and that all the processes and variables are deterministic, we can uniquely identify the system state after any sequence of process steps, and therefore both representations of executions are equivalent. Given an algorithm $\mathcal{A}$, we let $execs(\mathcal{A})$ denote the set of all executions of $\mathcal{A}$.

Given a permutation $\pi \in S_n$, we think of π as a bijection from $[n]$ to itself, and we write $\pi^{-1}(i)$ for the element that maps to i under π, for $i \in [n]$. We write $i \leq_\pi j$ if $\pi^{-1}(i) \leq \pi^{-1}(j)$; that is, i equals j, or i comes before j in π. Lastly, if $S \subseteq [n]$, we write $\min_\pi S$ for the minimum element in S, where elements are ordered by $\leq_\pi$.

3.2 The Mutual Exclusion Problem

Let $\mathcal{A}$ be an algorithm. For each process p_i, the steps of p_i contains the following *critical steps*: $\mathsf{try}_i, \mathsf{enter}_i, \mathsf{exit}_i, \mathsf{rem}_i$. For any critical step e, we define $type(e) = \mathsf{C}$. For simplicity, we assume that these steps, and the read and write steps of p_i, are the only steps performed by p_i. We say a process p_i is in its *trying section* if its last critical step in an execution is try_i. We say it is in its *critical section* if the last critical step is enter_i. We say it is in its *exit section* if the last critical step is exit_i. Finally, we say it is in its *remainder* section if the last critical step is rem_i, or there are no critical steps.

We say that $\mathcal{A}$ solves the *livelock-free mutual exclusion problem* if any finite execution $\alpha \in execs(\mathcal{A})$ satisfies the following properties.

- **Well Formedness**: Let p_i be any process, and consider the subsequence s of α consisting only of p_i's critical steps. Then s forms a prefix of the sequence $\mathsf{try}_i \circ \mathsf{enter}_i \circ \mathsf{exit}_i \circ \mathsf{rem}_i \circ \mathsf{try}_i \circ \mathsf{enter}_i \circ \mathsf{exit}_i \circ \mathsf{rem}_i \ldots$.
- **Mutual Exclusion**: For any two processes $p_i \neq p_j$, if the last occurrence of a critical step by p_i in α is enter_i, then the last critical step by p_j in α is *not* enter_j.

In addition, every *fair* execution[5] α of $\mathcal{A}$ satisfies:

- **Livelock Freedom**: For any process p_i, and any try_i step in α, there exists a later step enter_j, for some process p_j. In addition, for any exit_i step, there exists a later step rem_k, for some process p_k.

The well formedness condition says that every process progresses cyclically through its trying, critical, exit and remainder sections. The mutual exclusion property says that no two processes can be in their critical sections at the same time. The livelock freedom property says that if a process is in its trying section, then eventually, some process, perhaps not the same one, enters its critical section. Additionally, if a process is in its exit section, then eventually some process enters its remainder section. This means that the overall system always makes progress.

In addition to satisfying the three properties above, we want the mutex algorithm to be *nontrivial*, so that each process may request to enter the critical section anytime it is in the remainder section. For simplicity, we assume that the initial step of each process p_i is try_i.

3.3 The State Change Cost Model

In this section, we define the state change cost model for measuring the cost of a shared memory algorithm. In [1], it was proven that the cost of any shared memory mutual exclusion algorithm is infinite if we count every shared memory access. To obtain a more meaningful measure for cost, researchers have focused on models in which some memory accesses are free. Two important models that have been studied are the *distributed shared memory (DSM)* model and the *cache coherent (CC)* model. We define a new cost model, called the *state change (SC)* cost model, which is a simplification of the cache coherent model. Informally, the state change cost model charges an algorithm for a memory access only when the process performing the access changes its state. In particular, we charge the algorithm for each write performed by a process[6]. Additionally, the state change cost model allows unit cost busywaiting reads, but only on one variable at a time. However, the model is sufficiently generous that it permits algorithms to incur $O(n \log n)$ in all canonical executions. Formally, the cost model is defined as follows.

DEFINITION 3.1. ***The State Change Cost Model*** *Let $\mathcal{A}$ be an algorithm, and let $\alpha = s_0 e_1 s_1 \ldots e_t s_t \in execs(\mathcal{A})$ be a finite execution.*

1. *Let p_i be a process, and $j \in [t]$. We define $sc(\alpha, i, j)$ to be 1 if e_j is a shared memory access step by p_i, and $st(\alpha(j-1), i) \neq st(\alpha(j), i)$; it is 0 otherwise.*
2. *We define the* cost *of execution α to be $C(\alpha) = \sum_{i \in [n]} \sum_{j \in [t]} sc(\alpha, i, j)$.*

Notice that this model charges only for steps of p_i accessing the shared memory, and not for the critical steps of p_i (even though p_i may change its state after a critical step). The cost of α is simply the number of times a process changes state following shared memory steps, summed over all the processes. The SC cost model allows a limited form of busywaiting reads with bounded cost. For example, suppose the value of a register ℓ is currently 0, and process p_i repeatedly reads ℓ, until its value becomes 1. As long as ℓ's value is not 1, the process does not change its state, and thus, continues to read ℓ. Then, the algorithm is charged one unit for all reads up to when p_i reads ℓ as 1.

The cost of most practical algorithms is higher in the state change cost model than in the standard cache coherent model. For example, the CC model allows bounded cost busywaits on multiple registers at the same time, while the SC model does not. However, we believe the SC cost model is a mathematically clean and interesting model which allows a clear demonstration of our proof techniques, with few nonessential technical complications.

4. OVERVIEW OF THE LOWER BOUND

In this section, we give a detailed overview of our lower bound proof. Let $\mathcal{A}$ be any livelock-free mutual exclusion algorithm. The proof consists of three steps, which we call the *construction step*, the *encoding step*, and the *decoding step*. The construction step builds a finite execution

[4] Or simply α, if α has length $< t$.

[5] An execution is fair if every process that performs at least one critical step, and whose last critical step is not rem_i, takes another step.

[6] Note that if the process does not change its state after a write, that process will stay in the same state forever, and livelock freedom will be violated.

$\alpha_\pi \in execs(\mathcal{A})$ for each permutation $\pi \in S_n$, such that different permutations lead to different executions. The encode step produces a string E_π of length $O(C(\alpha_\pi))$ for each α_π,. The decode step reproduces α_π using only input E_π. Since each E_π uniquely identifies one of $n!$ different executions, some E_π must have length $\Omega(n \log n)$. Therefore, the corresponding execution α_π must have cost $\Omega(n \log n)$.

Fix a permutation $\pi = (\pi_1, \ldots, \pi_n) \in S_n$. We say that a process p_i has *lower (resp., higher) index (in π)* than process p_j if i comes before (resp., after) j in π. Roughly speaking, the construction step works as follows. We construct, in n stages, n different finite executions, $\alpha_1, \ldots, \alpha_n \in execs(\mathcal{A})$, where $\alpha_n = \alpha_\pi$. In each α_i, only the first i processes in the permutation, $p_{\pi_1}, \ldots, p_{\pi_i}$, take steps. Thus, α_1 is a solo execution by process p_{π_1}. Each process runs until it has completed its critical and exit sections once. We will show that the processes complete their critical sections in the order given by π, that is, p_{π_1} first, then p_{π_2}, etc., and finally, p_{π_i}. Next, we construct execution α_{i+1} in which process $p_{\pi_{i+1}}$ also takes steps, until it completes its critical and exit sections. α_{i+1} is constructed by starting with α_i, and then inserting steps by $p_{\pi_{i+1}}$, in such a way that $p_{\pi_{i+1}}$ is *not seen* by any of the the lower indexed processes $p_{\pi_1}, \ldots, p_{\pi_i}$. Informally, this is done by placing some of $p_{\pi_{i+1}}$'s writes immediately before writes by lower indexed processes, so that the latter writes overwrite any trace of $p_{\pi_{i+1}}$'s presence. Of course, there are many possible ways to make $p_{\pi_{i+1}}$ unseen to the lower indexed processes. For example, we can place all of $p_{\pi_{i+1}}$'s steps after all steps by lower indexed processes. But doing that, we may not be able to encode the execution using only $O(C_\pi)$ bits. The key to the construction is to produce an execution that both ensures higher indexed processes are unseen by lower indexed ones, and that can also be encoded efficiently.

While the above describes the intuition for the construction step, it is not exactly how we actually perform the construction. Instead of directly generating an execution α_i in stage i, we actually generate a set of *metasteps* M_i and a partial order $\preceq_i$ on M_i in stage i. A metastep consists of three sets of steps, the *read* steps, the *write* steps, and the *winning* step, which is a write step. All steps access the same register. A process performs at most one step in a metastep. We say a process is *contained* in the metastep if it takes a step in the metastep, and we say the *winner* of the metastep is the process performing the winning step. The purpose of a metastep is to hide the presence of all processes contained in the metastep, except possibly the winner. Given a set of metasteps M_i and a partial order $\preceq_i$ on M_i, we can generate an execution by first ordering M_i using *any* total order consistent with $\preceq_i$, to produce a sequence of metasteps. Then, for each metastep in the sequence, we expand the metastep into a sequence of steps, consisting of the write steps of the metastep, ordered arbitrarily, followed by the winning step, followed by the read steps, ordered arbitrarily. Notice that this sequence hides the presence of all processes except possibly the winner. The sequence of steps resulting from totally ordering M_i and then expanding each metastep is an execution which we call a *linearization* of $(M_i, \preceq_i)$. Of course, there may be many total orders consistent with $\preceq_i$, and many ways to expand each metastep, leading to many different linearizations. However, we will show that for the particular M_i and $\preceq_i$ we construct, all linearizations are "essentially the same". For example, in any linearization, all processes $p_{\pi_1}, \ldots, p_{\pi_i}$ complete their critical sections once, and they do so in that order. It is the set M_n and partial order $\preceq_n$, generated at the end of stage n in the construction step, that we eventually encode in the encoding step. The reason we construct a partial order of metasteps instead of directly constructing executions is that the partial order $\preceq_i$ affords us more flexibility in stage $i+1$ when we add steps by process $p_{\pi_{i+1}}$, which in turn leads to a more efficient encoding. We can show that all linearizations of $(M_n, \preceq_n)$ have the same cost, and we call this cost C_π.

We now describe the encoding step. In this step, an encoding algorithm takes as input $(M_n, \preceq_n)$, produced after stage n of the construction step. For any process p_i, we can show that all the metasteps containing p_i in M_n are totally ordered in $\preceq_n$. Thus, for any metastep containing p_i, we can say the metastep is p_i's *j'th metastep*, for some j. The encoding algorithm uses a table with n columns and an infinite number of rows. In the j'th row and i'th column of the table, which we call cell $T(i, j)$, the encoder records what process p_i does in its j'th metastep. However, to make the encoding short, we only record, roughly speaking, the *type*, either read or write, of the step that p_i performs in its j'th metastep. In addition, if p_i is the winner of the metastep, we also record a *signature* of the entire metastep. The signature basically contains a *count* of how many processes in the metastep perform read steps, and how many perform write steps (including the winning step). Note that the signature does not specify *which* processes read or write in the metastep, nor the register or value associated with any step. Now, if there are k processes involved in a metastep, the total number of bits we use to encode the metastep is $O(k) + O(\log k) = O(k)$. Indeed, for each non-winner process in the metastep, we use $O(1)$ bits to record its step type. For the winner process, we record its step type, and use $O(\log k)$ bits to record how many readers and writers are in the metastep. Notice that the cost to the algorithm for performing this metastep is also $O(k)$. Informally, this shows that the size of the encoding is proportional to the cost incurred by the algorithm. The final encoding of $(M_n, \preceq_n)$ is formed by iterating over all the metasteps in M_n, each time filling the table as described above. Then, we concatenate together all the nonempty cells in the table into a string E_π.

Lastly, we describe how, using E_π as input, the decoding step constructs an execution α_π that is a linearization of $(M_n, \preceq_n)$[7]. Roughly speaking, at any time during the decoding process, there exists an $m \in M_n$ such that the decoder algorithm has produced a linearization of all the metasteps that $\preceq_n m$ (as well as some metasteps that are incomparable to m). We say all such metasteps have been *executed.* This linearization is a prefix of α_π. Using this prefix, the decoder tries to find a minimal (with respect to $\preceq_n$) unexecuted metastep, which it then executes, by appending the steps in the metastep to the prefix; then the decoder restarts the decoding loop. To find a minimal unexecuted metastep, the decoder applies the transition function δ of $\mathcal{A}$ to the prefix to compute, for each process p_i, the step that p_i takes in the smallest unexecuted metastep containing p_i. Then, by reading E_π, the decoder finds the signature, that is, the number of readers and writers, of some minimal unexecuted metasteps. Suppose the decoder finds the signature of a metastep m' accessing some register ℓ (recall that all steps in a metastep access the same register), and suppose

[7] Note that even though our discussion involves π, the decoder does *not* know π. The only input to the decoder is the string E_π.

the signature indicates that m' contains r read steps and w write steps. Then, since the decoder knows each process's next step, it can check whether there are r processes whose next step reads ℓ, and w processes whose next step writes to ℓ. If so, the decoder *executes* all these steps on ℓ. That is, it appends them to the current execution, placing all the write steps before all the read steps, and placing the winning write — the write performed by the process whose cell in E_π contains the signature — last among the writes. We can show that these steps are exactly the steps contained in m', and that m' is a minimal unexecuted metastep. After executing m', the decoder tries to find a minimal metastep $\not\preceq_n m'$. By repeating this process until it has read all of E_π, the decoder produces an execution that is a linearization of $(M_n, \preceq_n)$.

5. THE CONSTRUCTION STEP

5.1 Description of the Construction

In this section, we present the algorithm for the construction step. For the remainder of this paper, fix $\mathcal{A}$ to be any livelock-free mutual exclusion algorithm with transition function δ. We begin with a formal definition of a metastep.

DEFINITION 5.1. *A* metastep *is identified by a label $m \in \mathcal{M}$, where $\mathcal{M}$ is an infinite set of labels. For any metastep m, we define the following attributes.*

1. *We let $read(m)$ and $write(m)$ be a set of read and write steps, resp. We let $win(m)$ be a write step. All steps access the same register. Any process performs at most one step in $read(m) \cup write(m) \cup win(m)$. We say a step in $read(m) \cup write(m) \cup win(m)$ is* contained *in m. We say $read(m)$ and $write(m)$ are the* reads *and* writes *of m, and we say $win(m)$ is the* winning step *of m.*
2. *We let $own(m)$ be the set of processes that perform a step contained in m. We say any process in $own(m)$ is* contained *in m. We say the process that performs $win(m)$ is the* winner *of m.*
3. *For any $i \in own(m)$, we let $step(m, i)$ be the step that process p_i takes in m.*
4. *We let $type(m) \in \{\mathtt{R}, \mathtt{W}, \mathtt{C}\}$. If $type(m) = \mathtt{R}$ (resp., $\mathtt{W}$, $\mathtt{C}$), we say m is a* read *(resp.,* write*,* critical*) metastep. If $type(m) = \mathtt{W}$, then $win(m) \neq \emptyset$; if $type(m) = \mathtt{R}$, then $win(m) = \emptyset$.*
5. *If $type(m) \in \{\mathtt{R}, \mathtt{W}\}$, we let $reg(m)$ be the register that all steps in m access. We say m* accesses *$reg(m)$.*
6. *If $type(m) = \mathtt{W}$, we let $val(m)$ be the value written by $win(m)$. We call $val(m)$ the* value *of m.*
7. *If $type(m) = \mathtt{C}$, then we let $crit(m)$ contain a critical step.*
8. *We let $pread(m)$ contain a set of read metasteps, and we call this the* preread set *of m. The meaning of $pread(m)$ will be described later.*

Let M be a set of metasteps, and let $\preceq$ be a partial order on M. Then a *linearization* of $(M, \preceq)$ is any execution produced by the procedure $\text{LIN}(M, \preceq)$, shown in Figure 1. The procedure $\text{SEQ}(m)$ returns a sequence of steps consisting of the write steps of m, then the winning step of m, then the read steps. It uses the helper function *concat*, which concatenates a set of steps in an arbitrary order.

In this section, we show how to create a set of metasteps M_i and a partial order $\preceq_i$ on M_i, for $i = 1, \ldots, n$. For every i, the only processes that may take steps in any metastep of M_i are processes $p_{\pi_1}, \ldots, p_{\pi_i}$. Furthermore, $(M_i, \preceq_i)$ satisfies the property that in any linearization of $(M_i, \preceq_i)$ processes $p_{\pi_1}, \ldots, p_{\pi_i}$ all complete their critical sections once, and they do so in that order. The processes also complete their exit sections. The construction algorithm is shown in Figure 1.

The procedure CONSTRUCT performs the construction in n stages. It takes as input an arbitrary permutation $\pi \in S_n$. For the remainder of this paper, fix an arbitrary π. In stage i, CONSTRUCT builds M_i and $\preceq_i$ by calling the procedure GENERATE with inputs M_{i-1} and $\preceq_{i-1}$ (constructed in stage $i-1$) and π_i. We define $M_0 = \preceq_0 = \emptyset$. We now describe $\text{GENERATE}(M_i, \preceq_i, \pi_i)$. For simplicity, we write M for M_i, $\preceq$ for $\preceq_i$, and j for π_i in the remainder of this section. GENERATE proceeds in a loop, and terminates when process p_j performs its rem_j action, that is, completes its critical and exit sections. In each iteration of the loop, we compute the next step e that p_j takes. Then, we either *insert* e into an existing metastep of M, or create a new metastep containing only e[8]. Let m' be the metastep that was modified or created in the *previous* iteration of the loop. Then, roughly speaking, we will insert e into a metastep if we can find a *write* metastep that $\not\preceq m'$, and which accesses the same register as e. If we cannot find such a metastep, we create a new metastep for e. If e is a write step, we make e the winning step of the new (write) metastep. Then we check whether there are any *read* metasteps in M on e's register that $\not\preceq m'$. If so, we order all such metasteps before e's metastep. We also add these metasteps to the preread set of e's metastep. If e is a read step, we simply create a new read metastep containing e. Now, we set m' to be the metastep we just modified or created, add m' to M if m' is a newly created metastep, and change $\preceq$ to order m'. Then, we let execution α be a linearization of *all metasteps* $\preceq m'$. Using α, we can now compute p_j's next step, and the loop repeats.

We now describe the construction in more detail. We will refer to specific line numbers in Figure 1 in angle brackets. For example, $\langle 8 \rangle$ refers to line 8. In $\langle 8 \rangle$, GENERATE begins by creating a new metastep m containing only the critical step try_j, indicating that p_j starts in its trying section. We add m to M, and set m' to m. m' keeps track of the metastep we created or modified during the previous iteration of the main loop. We then begin the main repeat loop, which ends when p_j performs its rem_j step. The loop begins at $\langle 10 \rangle$ by setting α to be a linearization of all metasteps $\preceq m'$. This is computed by the function $\text{PLIN}(M, \preceq, m')$. Using α, we can compute p_j's next step e as $\delta(\alpha, j)$. Let ℓ be the register that e accesses, if e is a read or write step.

We split into two cases, depending on e's type. If e is a write step $\langle 13 \rangle$, then we set m_w to be the minimum write metastep in M that accesses ℓ, and that $\not\preceq m'$. For any register, we can show that the set of all write metasteps accessing that register are totally ordered. Thus, if m_w exists, it is unique. When m_w exists, we insert e into m_w, by adding e to $write(m_w)$ $\langle 16 \rangle$. The idea is that this hides p_i's presence, because e will immediately be overwritten by another write step in m_w when we linearize any set of metasteps including m_w. Next, we add the relation (m', m_w) to $\preceq$, indicating

[8]If we insert e into a metastep m, then we will still refer to the metastep as m; that is, even though $own(m)$ changes, m retains the same "name". We do this for notational convenience.

1: **procedure** Construct(π)
2: $M_0 \leftarrow \emptyset$; $\preceq_0 \leftarrow \emptyset$
3: **for** $i \leftarrow 1, n$ **do**
4: $(M_i, \preceq_i) \leftarrow$ Generate$(M_{i-1}, \preceq_{i-1}, \pi_i)$
5: **end for**
6: **return** M_n, and the reflexive, transitive closure of $\preceq_n$

7: **procedure** Generate$(M, \preceq, j)$
8: $m \leftarrow$ new metastep; $crit(m) \leftarrow \{\mathsf{try}_j\}$
9: $M \leftarrow M \cup \{m\}$; $m' \leftarrow m$
10: **repeat**
11: $\alpha \leftarrow$ Plin$(M, \preceq, m')$; $e \leftarrow \delta(\alpha, j)$; $\ell \leftarrow reg(e)$
12: **switch**
13: **case** $type(e) = \mathtt{W}$:
14: $m_w \leftarrow \min_{\preceq}\{\mu \mid (\mu \in M) \wedge (reg(\mu) = \ell) \wedge (type(\mu) = \mathtt{W}) \wedge (\mu \not\preceq m')\}$
15: **if** m_w exists **then**
16: $write(m_w) \leftarrow write(m_w) \cup \{e\}$
17: $\preceq \leftarrow \preceq \cup \{(m', m_w)\}$; $m' \leftarrow m_w$
18: **else**
19: $m \leftarrow$ new metastep; $win(m) \leftarrow \{e\}$
20: $reg(m) \leftarrow \ell$; $type(m) \leftarrow \mathtt{W}$; $M \leftarrow M \cup \{m\}$
21: $M^r \leftarrow \max_{\preceq}\{\mu \mid (\mu \in M) \wedge (reg(\mu) = \ell) \wedge (type(\mu) = \mathtt{R}) \wedge (\mu \not\preceq m')\}$
22: **if** M^r exists **then**
23: $pread(m) \leftarrow M^r$
24: **for all** $\mu \in M^r$ **do** $\preceq \leftarrow \preceq \cup \{(\mu, m)\}$ **end for**
25: **end if**
26: $\preceq \leftarrow \preceq \cup \{(m', m)\}$; $m' \leftarrow m$
27: **case** $type(e) = \mathtt{R}$:
28: $m_{sw} \leftarrow \min_{\preceq}\{\mu \mid (\mu \in M) \wedge (reg(\mu) = \ell) \wedge (type(\mu) = \mathtt{W}) \wedge (\mu \not\preceq m') \wedge \mathrm{SC}(\alpha, \mu, j)\}$
29: **if** m_{sw} exists **then**
30: $read(m_{sw}) \leftarrow read(m_{sw}) \cup \{e\}$
31: $\preceq \leftarrow \preceq \cup \{(m', m_{sw})\}$; $m' \leftarrow m_{sw}$
32: **else**
33: $m \leftarrow$ new metastep; $read(m) \leftarrow \{e\}$
34: $reg(m) \leftarrow \ell$; $type(m) \leftarrow \mathtt{R}$; $M \leftarrow M \cup \{m\}$
35: $\preceq \leftarrow \preceq \cup \{(m', m)\}$; $m' \leftarrow m$
36: **end if**
37: **case** $type(e) = \mathtt{C}$:
38: $m \leftarrow$ new metastep; $crit(m) \leftarrow \{e\}$
39: $M \leftarrow M \cup \{m\}$; $m' \leftarrow m$
40: **end switch**
41: **until** $e = \mathsf{rem}_j$
42: **return** M and $\preceq$
43: **end procedure**

44: **procedure** Seq(m)
45: **if** $type(m) \in \{\mathtt{W}, \mathtt{R}\}$ **then**
46: **return** $concat(write(m)) \circ win(m) \circ concat(read(m))$
47: **else return** $crit(m)$
48: **end procedure**

49: **procedure** Lin$(M, \preceq)$
50: let $\leq^M$ be a total order on M consistent with $\preceq$
51: order M using $\leq^M$ as $m_1, m_2, \ldots, m_u$
52: **return** Seq$(m_1) \circ \ldots \circ$ Seq(m_u)
53: **end procedure**

54: **procedure** Plin$(M, \preceq, m)$
55: $N \leftarrow \{\mu \mid (\mu \in M) \wedge (\mu \preceq m)\}$
56: **return** Lin$(N, \preceq|_N)$
57: **end procedure**

50: **procedure** SC(α, m, i)
51: $\ell \leftarrow reg(m)$; $v \leftarrow val(m)$; choose $j \in [n], j \neq i$
52: **return** $st(\alpha \circ \mathsf{write}_j(\ell, v) \circ \mathsf{read}_i(\ell), i) \neq st(\alpha, i)$
53: **end procedure**

Figure 1: Stage i of the construction step.

that $m' \preceq m_w$. Finally, we set m' to be m_w.

In the case where m_w does not exist $\langle 18\rangle$, we create a new write metastep m containing only e, with e as the winning step. Then, we compute the set M^r of the *maximal read* metasteps in M accessing ℓ that $\not\preceq m'$. The read metasteps accessing ℓ are not necessarily totally ordered, so M^r may contain several elements. If M^r is nonempty, then we must be sure to order m after every metastep in M^r $\langle 24\rangle$. Otherwise, the processes performing the read metasteps may be able to see p_j. In addition, we set $pread(m)$ to M^r $\langle 23\rangle$. We call $pread(m)$ the *preread set* of m, and we call each metastep in $pread(m)$ a *preread* of m. A preread of m is always ordered before m in $\preceq$. Lastly, in $\langle 26\rangle$, we order m after m', then set m' to m.

The case when e is a read step is similar. Here, we begin by computing m_{sw}, which is the minimum write metasteps in M accessing ℓ that $\not\preceq m'$, and that would cause p_i to *change its state* if p_i read the value of the metastep $\langle 28\rangle$. We use the helper function SC(α, m, i), which returns a Boolean value indicating whether process p_i would change its state if it read the value of metastep m after execution α. If m_{sw} exists, then we add e to $read(m_{sw})$. Otherwise, we create a new read metastep m containing only e, and set $read(m) = \{e\}$. Note that in this latter case, performing e itself will cause p_i to change state. Indeed, if performing e does not cause p_i to change its state, and there also does not exist a write metastep causing p_i to change its state, then p_i will be stuck in its current state forever, violating the livelock freedom property of the mutex algorithm.

Lastly, if e is a critical step $\langle 37\rangle$, then we simply make a new metastep for e and order it after m'.

After i stages of the Construct procedure, we produce M_n and $\preceq_n$. For notational simplicity, in the remainder of this paper, we set $M \equiv M_n$, and $\preceq \equiv \preceq_n$.

5.2 Properties of the Construction

We now present some results about the construction step. We give mostly proof sketches, and defer full proofs to a full version of the paper.

Lemma 5.2. *For any $i \in [n]$, $\preceq_i$ defines a partial ordering on M_i.*

Lemma 5.3. *Let $i \in [n]$, and let $\ell \in L$ be any register. Then, all the write metasteps in M_i that access ℓ are totally ordered in $\preceq_i$.*

Both lemmas can be verified by induction on the main loops in procedures Construct and Generate.

Lemma 5.4. *Let $1 \leq i \leq j \leq k \leq n$, let α_j be a linearization of $(M_j, \preceq_j)$, and let α_k be a linearization of $(M_k, \preceq_k)$. Then $\alpha_j|\pi_i = \alpha_k|\pi_i$.*

This lemma says that a process p_{π_i} cannot distinguish between a linearization from the j'th or k'th stage of Construct, for $i \leq j \leq k$. This in turn implies that for $i < j \leq k$, p_{π_i} cannot tell if process p_{π_j} is present or not in any linearization of $(M_k, \preceq_k)$. This can be shown by considering how metasteps are created, ordered and linearized. Indeed, when a higher indexed process (p_{π_j}) performs a write step, this write step is placed either in a metastep so that it is immediately overwritten by the write of a lower indexed process, or placed after all reads by lower indexed processes on the same register (these reads become prereads of p_{π_j}'s write). Therefore, lower indexed processes never see any values written by higher indexed ones, and cannot tell if they are present. This lemma can be used to show the following.

Theorem 5.5. *Let $1 \leq i \leq n$. Then in any linearization of $(M_i, \preceq_i)$, each process $p_{\pi_1}, \ldots, p_{\pi_i}$ completes its critical section, and they do so in that order.*

Proof. *(Sketch)* The fact that processes complete their critical section follows from the livelock freedom property of the mutex algorithm, and Lemma 5.4. To show that they complete them in the order π, consider the minimum j such that there exists a process p_{π_k} that enters its critical section before p_{π_j}, and $j < k$. Now, consider the moment when p_{π_k} performs its enter_{π_k} step. At this point, p_{π_j} has not performed its enter_{π_j} step yet. Now, since processes $p_{\pi_1}, \ldots, p_{\pi_j}$ all do not see p_{π_k} by Lemma 5.4, then we can pause p_{π_k}, and run $p_{\pi_1}, \ldots, p_{\pi_j}$, and be guaranteed that p_{π_j} will eventually perform its enter_{π_j} step. However, then p_{π_j} and p_{π_k} have both performed their enter steps, but not their exit steps, which violates the mutual exclusion property, a contradiction. Thus, processes must enter their critical sections in the order π. □

5.3 Additional Properties of the Construction

Finally, we present several lemmas which we use in Section 7.2 to prove the correctness of the decoding step. We begin with the following definitions.

DEFINITION 5.6. *Let $N \subseteq M$. We say N is a* prefix *of M if $\forall m \in N$, we have $\{\mu \,|\, (\mu \in M) \wedge (\mu \preceq m)\} \subseteq N$.*

Thus, a prefix of M is a union of chains of $(M, \preceq)$.

DEFINITION 5.7. *Let N be a prefix of M, and let $\ell \in L$ and $i \in [n]$. Define:*

1. $\Gamma^{\mathtt{W}}_i(N) = \{\mu | (\mu \notin N)) \wedge (type(\mu) = \mathtt{W}) \wedge (i \in own(\mu))\}$.
2. $\Gamma^{\mathtt{W}}(N, \ell) = \{\mu \,|\, (\mu \notin N) \wedge (reg(\mu) = \ell) \wedge (type(\mu) = \mathtt{W})\}$.
3. $\Gamma^{\mathtt{W}}_i(N, \ell) = \{\mu \,|\, (\mu \notin N) \wedge (reg(\mu) = \ell) \wedge (type(\mu) = \mathtt{W}) \wedge (i \in own(\mu)))\}$.
4. $\Gamma^{\mathtt{R}}_i(N, \ell) = \{\mu \,|\, (\mu \notin N) \wedge (reg(\mu) = \ell) \wedge (type(\mu) = \mathtt{R}) \wedge (i \in own(\mu)))\}$.
5. $\gamma^{\mathtt{W}}(N, \ell) = \min_{\preceq} \Gamma^{\mathtt{W}}(N, \ell)$*, or* $\perp$ *if* $\Gamma^{\mathtt{W}}(m, \ell) = \emptyset$.
6. $\gamma^{\mathtt{W}}_i(N, \ell) = \min_{\preceq} \Gamma^{\mathtt{W}}_i(N, \ell)$*, or* $\perp$ *if* $\Gamma^{\mathtt{W}}_i(N, \ell) = \emptyset$.

These functions define various subsets of M related to N. For example, $\Gamma^{\mathtt{W}}_i(N)$ is the set of write metasteps not in N that contain p_i, $\Gamma^{\mathtt{W}}(N, \ell)$ is the set of write metasteps not in N accessing ℓ, and $\gamma^{\mathtt{W}}_i(N, \ell)$ is the minimum write metastep not in N containing p_i that accesses ℓ, if it exists, etc. We have the following lemmas.

LEMMA 5.8. *Let N be a prefix of M, and let $i \in [n]$. Define $m^* = \gamma^{\mathtt{W}}(N, \ell)$, $m^*_i = \gamma^{\mathtt{W}}_i(N, \ell)$, and suppose $m^* \neq \perp$, $m^*_i \neq \perp$, and $type(step(m^*_i, i)) = \mathtt{W}$. Then $m^* = m^*_i$.*

That is, if p_i does a write step in the first write metastep on ℓ not in N that contains p_i, then that metastep equals the first write metastep on ℓ not in N.

Proof. *(Sketch)* To prove this lemma, let $j = own(win(m^*))$. Notice that p_j is the minimum index process (w.r.t. π) contained in m^*. We claim that $i \geq_\pi j$. Indeed, suppose $i <_\pi j$. Then, procedure CONSTRUCT created the steps for process p_i in iteration $\pi^{-1}(i)$, before it created the steps for process p_j in iteration $\pi^{-1}(j)$. Consider iteration $\pi^{-1}(j)$ of CONSTRUCT, and the iteration of the main loop of GENERATE where p_j's step e in m^* was created; e is a write step. Since the write metastep m^*_i has already been constructed, then in $\langle 14 \rangle$, GENERATE will find m^*_i as the minimum write metastep on ℓ not in N, and sets $m_w = m^*_i$; in $\langle 16 \rangle$, it adds e to the write set of $m^*_i = m^*$. Thus, the minimum index of a process in m^* is at most i, a contradiction.

We have shown that $i \geq_\pi j$. Consider iteration $\pi^{-1}(i)$ of CONSTRUCT, and the iteration of the main repeat loop of GENERATE where p_i's step e in m^*_i was created. Since e is a write step, then in $\langle 14 \rangle$, GENERATE sets $m_w = m^*$, and in $\langle 16 \rangle$, it adds e to the write set of $m^* = m^*_i$. Thus, the lemma holds. □

LEMMA 5.9. *Let N be a prefix of M, and let $\ell \in L$ and $i \in [n]$. Define $m^*_i = \gamma^{\mathtt{W}}_i(N, \ell)$, and suppose $m^*_i \neq \perp$ and $type(step(m^*_i, i)) = \mathtt{R}$. Then $m^*_i = \min_{\preceq}\{\mu \,|\, (\mu \in \Gamma^{\mathtt{W}}(N, \ell)) \wedge \mathrm{SC}(\mathrm{PLIN}(M, \preceq, \mu), \mu, i)\}$.*

Here, we used the functions $\mathrm{PLIN}(M, \preceq, m)$ and $\mathrm{SC}(\alpha, m, i)$, as defined in Figure 1. Recall that the former function returns a partial linearization of $(M, \preceq)$, consisting of all metasteps $\preceq m$. The latter function returns true exactly when process p_i, whose state is as in the final state of α, will change its state upon reading the value written by write metastep m. This lemma states that, if p_i does a read step in the first write metastep on ℓ not in N that contains p_i, then that metastep equals the first write metastep on ℓ not in N whose value causes p_i to change its state.

The proof of this lemma is very similar to the proof for Lemma 5.8. If e is the step p_i takes in m^*_i, then in iteration $\pi^{-1}(i)$ of CONSTRUCT, and the iteration of GENERATE where e was created, $\langle 30 \rangle$ of GENERATE adds e to the read set of the minimum write metastep on ℓ not in N that causes p_i to change its state (if this metastep exists). We omit the details.

LEMMA 5.10. *Let N be a prefix of M, and let $\ell \in L$ and $i \in [n]$. Suppose $\Gamma^{\mathtt{W}}(N, \ell) \neq \emptyset$, $\Gamma^{\mathtt{W}}_i(N, \ell) = \emptyset$, and $\Gamma^{\mathtt{R}}_i(N, \ell) \neq \emptyset$. Then $\max_{\preceq} \Gamma^{\mathtt{R}}_i(N, \ell) \in pread(\gamma^{\mathtt{W}}(N, \ell))$.*

That is, if there exist write metasteps on ℓ not in N, but none of them contain p_i, and p_i is contained in read metasteps on ℓ not in N, then the largest read metastep on ℓ not in N containing p_i is contained in the preread set of the minimum write metastep on ℓ not in N. This can be verified by considering lines $\langle 21 - 24 \rangle$ of GENERATE.

6. THE ENCODING STEP

6.1 Description of the Encoding

We can show that all linearizations of $(M, \preceq)$ have the same cost in the state change cost model, say C. In this section, we describe an algorithm that uniquely encodes $(M, \preceq)$ as a string of length $O(C)$. The encoding uses a two dimensional grid of cells, with n columns and an infinite number of rows. We fill some of the cells with strings. The contents of the cell in column i and row j is denoted by $T(i, j)$. It represents the *type of operation* process p_i performs in its j'th metastep, and possibly a *count* of how many reads, writes and prereads are in p_i's j'th metastep. The complete encoding E_π is produced by concatenating all nonempty cells $T(1, \cdot)$ (in order), then appending all nonempty cells $T(2, \cdot)$, etc., and finally appending all nonempty cells $T(n, \cdot)$. The encoder uses the helper function $\mathrm{PC}(p, m)$, where p is a process and m is a metastep containing p. The function returns a number q, such that m is p's q'th metastep. We also use the helper function $nrows(T, i)$, which returns how many nonempty cells there are in column i of T. Please see Figure 2 for the pseudocode.

The encoder works by iterating over every metastep $m \in M$. Suppose first that m is a write metastep. Then for every step e in $read(m) \cup write(m)$, let p be the process that performs e, and let m be p's q'th metastep. The encoder writes e's type, either $\mathtt{R}$ or $\mathtt{W}$, in cell $T(p, q)$. For the winning step, we record in $T(p, q)$ not only that it is a write, but also the *signature* of the metastep $\langle 9 \rangle$. The signature of m records the *number* of read and write steps (including the winning step) in m, as well as the size of m's preread set $pread(m)$. It is a string of the form $\mathtt{PR}x\mathtt{R}y\mathtt{W}z$.

1: **procedure** $\text{Encode}(M, \preceq)$
2: **for all** $m \in M$ **do**
3: **switch**
4: **case** $type(m) = \texttt{W}$:
5: **for all** $e \in read(m) \cup write(m)$
6: $p \leftarrow own(e)$; $q \leftarrow \text{Pc}(p, m)$; $T(p, q) \leftarrow type(e)$
7: **end for**
8: $p \leftarrow own(win(m))$; $q \leftarrow \text{Pc}(p, m)$
9: $T(p, q) \leftarrow \texttt{W},\texttt{PR}\,|pread(m)|\,\texttt{R}\,|read(m)|\,\texttt{W}\,|write(m) + 1|$
10: **case** $type(m) = \texttt{R}$:
11: $p \leftarrow own(read(m))$; $q \leftarrow \text{Pc}(p, m)$
12: **if** $\exists \mu \in M$ such that $m \in pread(\mu)$ **then**
13: $T(p, q) \leftarrow \texttt{PR}$
14: **else** $T(p, q) \leftarrow \texttt{SR}$ **end if**
15: **case** $type(m) = \texttt{C}$
16: $p \leftarrow own(crit(m))$; $q \leftarrow \text{PC}(p, m)$; $T(p, q) \leftarrow \texttt{C}$
17: **end switch**
18: **end for**

19: **for** $i \leftarrow 1, n$ **do**
20: **for** $j \leftarrow 1, nrows(T, i)$ **do**
21: $E_\pi \leftarrow E_\pi \circ T(i, j) \circ \texttt{\#}$
22: **end for**
23: $E_\pi \leftarrow E_\pi \circ \texttt{\$}$
24: **end for**
25: **return** E_π
26: **end procedure**

27: **procedure** $\text{Pc}(p, m)$
28: $N \leftarrow \{\mu \,|\, (\mu \in M) \wedge (p \in own(\mu))\}$
29: sort N in increasing order of $\preceq$ as $n_1, \ldots, n_{|N|}$
30: **return** $q \in 1, \ldots, |N|$ such that $n_q = m$
31: **end procedure**

Figure 2: Encoding M and $\preceq$ as a string E_π.

If m is a read metastep, then it contains only one step, in $read(m)$, say by process p. If m is a preread of any other (write) metastep μ, then we write PR in $T(p, q)$ $\langle 13\rangle$. Otherwise, we write SR $\langle 14\rangle$. Lastly, if m is a critical step, we write C in $T(p, q)$.

In the remainder of this paper, let E be the string Encode outputs given input M and $\preceq$.

6.2 Properties of the Encoding

We now state some results about the efficiency of the encoding.

Lemma 6.1. *Let α_1 and α_2 be two different linearizations of $(M, \preceq)$. Then α_1 and α_2 have the same cost in the state change cost model.*

Since all linearizations have the same cost, we let C be this cost.

Theorem 6.2. *The length of E_π is $O(C)$.*

Proof. *(Sketch)* For any metastep $m \in M$, we compare the number of bits used to encode m, and the cost for the algorithm to execute m in the state change cost model. The cost to encode a read metastep is $O(1)$, since we record either SR or PR for the metastep. Consider a write metastep m, and suppose m contains k processes. We first ignore the cost to encode the number of prereads in m. Let r be the number of reads in m, and w be the number of writes. If a process p is not the winner of m, then we use $O(1)$ bits to encode the type of p's step. If p is the winner, then we encode its type, and we also use $O(\log r) + O(\log w) = O(\log k) = O(k)$ bits encode the number of reads and writes. Now we count the bits used to encode the number of prereads. We notice that a read metastep can only appear as a preread in *one* write metastep. Thus, by counting every read metastep twice, we can account for the cost to encode all the prereads. Thus, if m is any metastep containing k processes, then the amortized (over all the metasteps of M) number of bits used to encode m is $O(k)$. We now claim that the algorithm incurs $O(k)$ cost in the state change model to execute m. Indeed, all write steps have unit cost. A read is only added to a write metastep if the metastep's value causes the reader to change its state. A read in a read metastep also has unit cost, as we have already argued the reader must change its state after the step. Therefore, $|E_\pi|$ is proportional to the cost to the algorithm C. □

7. THE DECODING STEP

7.1 Description of the Decoding

We now describe the decoding algorithm. We first give an informal description of the algorithm, and then give a more detailed one. The decoder creates an execution α, and repeatedly appends steps to α, until α equals a linearization of $(M, \preceq)$. Note that this means there is a particular total order on M consistent with $\preceq$, and a particular way to expand each metastep in M via Seq, that produces an execution that equals α; that is, α equals the output of $\text{Lin}(M, \preceq)$, for some (nondeterministic) execution of $\text{Lin}(M, \preceq)$. Now, each time the decoder appends a sequence of steps, those steps are exactly the steps contained in some $m \in M$. We say the decoder has *executed* m. m has the property that it is a *minimal (w.r.t. $\preceq$) unexecuted metastep* in M. Thus, the decoding algorithm essentially runs a loop where in each iteration, it finds and executes a minimal unexecuted metastep of M.

We now describe the decoding algorithm in more detail. Please see Figure 3 for the pseudo-code. We first describe the variables in Decode. α is the execution that the decoder builds. *done* $\subseteq [n]$ is the set of processes that have completed their critical and exit sections. For $i \in [n]$, pc_i is the number of metasteps the decoder has executed that contain p_i, and e_i is p_i's step in the minimal unexecuted metastep containing p_i. We call e_i process p_i's *pending step.* At certain points in the decoding, the decoder may not yet know the pending steps of some processes. If the decoder knows the pending step of process p_i, then it places i in *wait*. For $\ell \in L$, R_ℓ (resp., W_ℓ) contains the set of processes whose pending step is a read step (resp., write step) on register ℓ. For any $i \in [n]$, if $i \in PR_\ell$, then p_i has performed its last read on ℓ in M. Lastly, if $sig_\ell \neq \varepsilon$, then sig_ℓ contains the signature of the minimum unexecuted write metastep on ℓ.

Each iteration of the main repeat loop of Decode consists of two sections, from $\langle 6 - 37\rangle$, and from $\langle 38 - 45\rangle$. The purpose of the first section is to find the pending step of each process. The purpose of the second section is to find a set of processes whose pending steps together form the steps of a minimal unexecuted metastep. Consider any $i \notin done \cup wait$. That is, p_i has not finished its critical and exit sections yet, and the decoder does not know its pending step. In $\langle 7\rangle$, the decoder increments pc_i, and calls the helper function $getStep(E, i, pc_i)$ to determine the type of p_i's pending step. Recall that E is stored as the concatenation of type and signature information about each process's steps in the metasteps containing that process. The decoder adds i to *wait*. It then switches based on the value of $step_i$.

Consider the case $step_i = \texttt{W}$. In $\langle 11 - 12\rangle$, the decoder loops until p_i's next step is a write step[9]. Let ℓ be the register that p_i's pending step e_i writes to. In $\langle 15\rangle$, the decoder adds i to W_ℓ. Also, if $step_i$ contains a signature sig, the decoder sets sig_ℓ to $makesig(sig, i)$. If $sig = \texttt{PR}pr\texttt{R}r\texttt{W}w$, then

[9]Note that we do this because p_i may do some critical steps before a write step.

```
1: procedure DECODE(E)
2:    ∀i ∈ [n] : pc_i ← 0, step_i ← ε
3:    ∀ℓ ∈ L : sig_ℓ ← ε, R_ℓ, PR_ℓ, W_ℓ ← ∅
4:    α ← try_1 ∘ try_2 ∘ ... ∘ try_n;   done ← ∅;   wait ← ∅
5:    repeat
6:       for all (i ∉ done) ∧ (i ∉ wait)) do
7:          pc_i ← pc_i + 1;   step_i ← getStep(E, i, pc_i)
8:          e_i ← δ(α, i);   wait ← wait ∪ {i}
9:          switch
10:            case step_i = W:
11:               while type(e_i) ≠ W do
12:                  α ← α ∘ e_i;   e_i ← δ(α, i) end while
13:               if step_i contains a signature sig then
14:                  sig_{reg(e_i)} ← makesig(sig, i) end if
15:               W_{reg(e_i)} ← W_{reg(e_i)} ∪ {i}
16:            case step_i = R:
17:               while type(e_i) ≠ R do
18:                  α ← α ∘ e_i;   e_i ← δ(α, i) end while
19:               if sig_{reg(e_i)} ≠ ε then
20:                  ℓ ← reg(e_i)
21:                  if st(α ∘ e_{sig_ℓ.v} ∘ read_i(ℓ), i) ≠ st(α, i) then
22:                     R_ℓ ← R_ℓ ∪ {i} end if
23:            case step_i = PR:
24:               while type(e_i) ≠ R do
25:                  α ← α ∘ e_i;   e_i ← δ(α, i) end while
26:               PR_{reg(e_i)} ← PR_{reg(e_i)} ∪ {i}
27:               α ← α ∘ e_i;   wait ← wait\{i}
28:            case step_i = SR:
29:               while type(e_i) ≠ R do
30:                  α ← α ∘ e_i;   e_i ← δ(α, i) end while
31:               α ← α ∘ e_i;   wait ← wait\{i}
32:            case step_i = C:
33:               α ← α ∘ e_i;   wait ← wait\{i}
34:            case step_i = $:
35:               done ← done ∪ {i}
36:          end switch
37:       end for

38:       for all ℓ such that sig_ℓ ≠ ε do
39:          if (|R_ℓ| = sig_ℓ.r) ∧ (|PR_ℓ| = sig_ℓ.pr) ∧ (|W_ℓ| = sig_ℓ.w)
40:             β ← concat(⋃_{i∈W_ℓ\{sig_ℓ.v}} e_i)
41:             γ ← concat(⋃_{i∈R_ℓ} e_i)
42:             α ← α ∘ β ∘ e_{sig_ℓ.v} ∘ γ
43:             wait ← wait\(R_ℓ ∪ W_ℓ)
44:             sig_ℓ ← ε;   R_ℓ, PR_ℓ, W_ℓ ← ∅
45:       end if end for
46:    until done = {1, ..., n}
47:    return α
48: end procedure
```

Figure 3: Decoding $E = E_\pi$ to produce a linearization of $(M, \preceq)$.

$makesig(sig, i)$ sets $sig_\ell.v \leftarrow i$ (indicating p_i is the winner of the metastep corresponding to this signature), $sig_\ell.r \leftarrow r$, $sig_\ell.w \leftarrow w$, and $sig_\ell.pr \leftarrow pr$.

Next, consider the case $step_i = \mathtt{R}$. The decoder loops until e_i is a read step $\langle 17-18\rangle$. Suppose e_i reads ℓ. Then the decoder checks whether $sig_\ell \neq \varepsilon$. If so, the decoder checks whether the (value of the) winning (write) step in the metastep corresponding to this signature, namely $e_{sig_\ell.v}$, would cause p_i to change its state $\langle 21\rangle$. If so, the decoder adds i to R_ℓ. Otherwise, it does nothing.

Next, consider the case $step_i = \mathtt{PR}$. Then, p_i's next unexecuted metastep is a read metastep. Let e_i be p_i's pending read step, and let ℓ be the register e_i reads. Since $step_i = \mathtt{PR}$, the decoder knows e_i is p_i's last read step on ℓ in M. So it adds i to PR_ℓ. The decoder then executes e_i $\langle 27\rangle$, and removes i from $wait$, indicating that it needs to compute a new pending step for p_i in the next iteration of the repeat loop.

Next, consider the case $step_i = \mathtt{SR}$. Then, p_i's next unexecuted metastep is a read metastep. The decoder executes this metastep, and removes i from $wait$ $\langle 31\rangle$.

If $step_i = \mathtt{C}$, then e_i is a critical step. The decoder appends e_i to α, and removes i from $wait$. Finally, if $step_i = \$$, then p_i has finished all its steps in M, and the decoder adds p_i to $done$.

We now describe what DECODE does in $\langle 38-45\rangle$. In $\langle 38\rangle$, the decoder finds some ℓ for which it knows the signature. It then checks that the sizes of R_ℓ, W_ℓ and PR_ℓ match the signature $\langle 39\rangle$. If so, it sets β to be the concatenation, in an arbitrary order, of all the write steps e_i, for $i \in W_\ell \backslash \{sig_\ell.v\}$. It sets γ to be the concatenation of all read steps e_i, for $i \in R_\ell$. Then, it appends $\beta \circ e_{sig_\ell.v} \circ \gamma$ to α. The decoder removes $R_\ell \cup W_\ell$ from $wait$ $\langle 43\rangle$, to indicate it needs compute pending steps for these processes in the next iteration of the repeat loop. It also resets $sig_\ell, R_\ell, PR_\ell$ and W_ℓ.

The decoder performs the repeat loop until $done = [n]$, indicating all processes have finished their critical and exit sections.

7.2 Properties of the Decoding

In this section, we show that DECODE produces an execution that is a linearization of $(M, \preceq)$. In particular, we show that each time the decoder appends a sequence of steps to α, those steps are exactly the steps of some minimal unexecuted metastep m, and that all the steps in $write(m)$ are appended before $win(m)$, which is appended before all the steps in $read(m)$. Furthermore, we show that in each iteration of the main loop of DECODE, the decoder does append some steps to α. Thus, eventually α equals a linearization of $(M, \preceq)$. The proof uses induction on the execution of the decoder. Below, we define what it means for the decoder to behave correctly up to some point in its execution.

DEFINITION 7.1. *Let $j \geq 0$, and let $e_1 e_2 \ldots e_t$ be the value of α at $\langle 6\rangle$ in the j'th iteration of the main repeat loop of* DECODE. *Note that each $e_i, i \in [t]$, is a step. We say the decoder is* correct up to iteration j *if the following hold at $\langle 6\rangle$ of iteration j:*

1. *$\alpha \in \mathrm{SEQ}(m_1) \circ \ldots \circ \mathrm{SEQ}(m_u)$, where $m_1 \in \min\{\mu \mid \mu \in M\}$, and $\forall i \in [u-1] : m_{i+1} \in \min_\preceq\{\mu \mid (\mu \in M) \wedge (\mu \not\preceq m_i)\}$. Let $N = \bigcup_{i\in[u]} m_i$.*
2. *For any $\ell \in L$, let $m^\ell = \gamma^{\mathtt{W}}(N, \ell)$. Then we have $R_\ell \subseteq read(m^\ell)$, $W_\ell \subseteq write(m^\ell) \cup win(m^\ell)$, and $PR_\ell \subseteq pread(m^\ell)$.*
3. *For all $\ell \in L$, if $sig_\ell \neq \varepsilon$, then $sig_\ell.r = |read(m^\ell)|$, $sig_\ell.w = |write(m^\ell) + 1|$, and $sig_\ell.pr = |pread(m^\ell)|$.*

Thus, the decoder is correct up to iteration j if three types of conditions are satisfied. First, the sequence of steps α the decoder has produced up to iteration j is belongs to $\mathrm{SEQ}(m_1) \circ \ldots \circ \mathrm{SEQ}(m_u)$. Note that this means there are some (nondeterministic) executions of SEQ on inputs $m_1, \ldots, m_u$, whose concatenation equals α. In addition, m_1 is a minimal metastep of M, and each m_i is a minimal metastep not preceding m_{i-1}. This implies that α is a prefix of a linearization of $(M, \preceq)$. Also, we have that $N = \bigcup_{i\in[u]} m_i$ is a prefix of M. Second, for any $\ell \in L$, the sets R_ℓ, W_ℓ and PR_ℓ are subsets of the read, write and preread set of the minimum write metastep on ℓ not in N. Third, if a signature exists for ℓ, then it contains the sizes of the read, write and preread set of $\gamma^{\mathtt{W}}(N, \ell)$. We can verify that the decoder is correct up to iteration 0.

LEMMA 7.2. *Let $j \geq 0$, and suppose* DECODE *is correct up to iteration j. Then* DECODE *either terminates after iteration j, or it is correct up to iteration $j+1$.*

Proof. *(Sketch)* Suppose the decoder does not terminate in iteration j. We first verify that the conditions in the correctness definition continue to hold after lines $\langle 6 - 37\rangle$. Later, we verify they hold after $\langle 38 - 45\rangle$. Let $i \in [n]$. Fix any $\ell \in L$, and let $m^* = \gamma^{\mathtt{W}}(N, \ell)$. We consider four cases, depending on the type of $step_i$.

If $step_i = \mathtt{W}$, then i is added to W_ℓ, where $\ell = reg(e_i)$, and e_i is p_i's pending step. Since we do not append any steps to α in this case, condition 1 holds. To verify condition 2, note that the minimum write metastep on ℓ containing p_i and not in N, namely $\gamma_i^{\mathtt{W}}(N, \ell)$, is equal to m^*, by Lemma 5.8. Thus, p_i performs a write step in m^*, and so $W_\ell \cup \{i\} \subseteq write(m^*) \cup win(m^*)$. If $step_i$ contains a signature, then by the same argument, this is the signature for metastep m^*, so condition 3 holds.

If $step_i = \mathtt{R}$ and i is added to R_ℓ, then $\ell = reg(e_i)$, where e_i is p_i's pending step. Let $m' = \gamma^{\mathtt{W}}_{sig_\ell.v}(N, \ell)$. Then $m' = m^*$, by Lemma 5.8. Also, since reading $val(e_{sig_\ell.v})$ causes p_i to change its state, we have by Lemma 5.9 that $m^* = \gamma_i^{\mathtt{W}}(N, \ell)$. Thus, p_i takes a read step in m^*, so $R_\ell \cup \{i\} \subseteq read(m^*)$, and condition 2 (and also 1 and 3) holds.

If $step_i = \mathtt{PR}$, then i is added to PR_ℓ, where $\ell = reg(e_i)$ and e_i is p_i's pending step. By the correctness of the decoder, e_i is p_i's final read on ℓ in M. Thus by Lemma 5.10, i belongs to $pread(m^*)$, and condition 2 holds. Lastly, if $step_i = \mathtt{SR}$, it is easy to show that condition 1 (and 2, 3) continues to hold.

We now verify that the correctness conditions hold after lines $\langle 38 - 45\rangle$. Suppose that the test on $\langle 39\rangle$ succeeds. Let $k = sig_{\ell.v}$; then k is the winner of m^*. Because of conditions 2 and 3, we must have $R_\ell = read(m^*)$, $W_\ell = write(m^*) \cup win(m^*)$, and $PR_\ell = pread(m^*)$. Thus, $R_\ell \cup W_\ell = own(m^*)$, and $concat(\bigcup_{i \in W_\ell \setminus \{k\}} e_i) \circ e_k \circ concat(\bigcup_{i \in R_\ell} e_i) \in \textsc{Seq}(m^*)$. Lastly, we have that $m^* \in \min_{\preceq}\{\mu \mid (\mu \in M) \wedge (\mu \not\preceq m_u)\}$, because for every process in $own(m^*)$, their pending step is their step in m^*, and also, every metastep in $pread(m^*)$ has been executed . Thus, condition 1 (and 2, 3) continues to hold, and the lemma is proved. ☐

Lemma 7.2 shows a safety property of Decode; that is, the decoder never executes a nonminimal unexecuted metastep. We now show a liveness property, that in every iteration of the main loop of the decoder, it executes some metastep.

Lemma 7.3. *Suppose* Decode *is correct up to iteration j, where $j \geq 1$. Then* Decode *either terminates or it executes some metastep in iteration j.*

Let m_u be the last metastep the decoder executed before iteration j. Then we can easily show that the decoder executes some minimal m not preceding m_u in iteration j. Informally, this is because the pending step of every process contained in m is its step in m, and so Decode either appends a read or critical step in $\langle 27\rangle$, $\langle 31\rangle$ or $\langle 33\rangle$, or the test on $\langle 39\rangle$ succeeds and the decoder appends to α in $\langle 42\rangle$. Together, Lemmas 7.2 and 7.3 imply the following.

Theorem 7.4. *Let α be the execution produced by* Decode*. Then α is a linearization of $(M, \preceq)$.*

7.3 A Lower Bound for Mutual Exclusion

Theorem 7.5. *Let $\mathcal{A}$ be any livelock-free mutual exclusion algorithm. Then in some $\alpha \in execs(A)$ in which processes $p_1, \ldots, p_n$ all complete their critical sections once, we have $C(\alpha) = \Omega(n \log n)$.*

Proof. For each $\pi \in S_n$, let $(M^\pi, \preceq^\pi) = \textsc{Construct}(\pi)$, $E_\pi = \textsc{Encode}(M^\pi, \preceq^\pi)$, and $\alpha_\pi = \textsc{Decode}(E_\pi)$. By Theorem 7.4, α_π is a linearization of $(M^\pi, \preceq^\pi)$. Thus, by Theorems 5.5, we have that $p_1, \ldots, p_n$ all complete their critical sections once in α_π, and they complete them in the order π. Therefore, for $\pi_1, \pi_2 \in S_n$, $\pi_1 \neq \pi_2$, we have $\alpha_{\pi_1} \neq \alpha_{\pi_2}$. Thus, the (deterministic) algorithm Decode produces $n!$ different outputs, on input from the set $\{E_\pi\}_{\pi \in S_n}$. Therefore, there exists $\pi \in S_n$ such that $|E_\pi| = \Omega(\log(n!)) = \Omega(n \log n)$[10]. By Theorem 6.2, we have that $|E_\pi| = O(C(\alpha_\pi))$. Thus, we have $C(\alpha_\pi) = \Omega(n \log n)$, and the theorem is proved. ☐

8. CONCLUSIONS

In this paper, we have established a lower bound of $\Omega(n \log n)$ memory accesses in the state change cost model for solving n process mutual exclusion. Our proof technique uses an information theoretic characterization of a necessary condition for solving mutual exclusion, and relates this to the information processes can gain through access to shared registers. Our proof technique can be extended to accommodate stronger memory primitives. We believe it also extends with minor modifications to the cache coherent cost model. We believe our proof technique is intuitive and flexible, and may be used to establish lower bounds for other problems for which current techniques are complex or insufficient.

9. REFERENCES

[1] R. Alur and G. Taubenfeld. Results about fast mutual exclusion. In *Proceedings of the 13th IEEE Real-time Systems Symposium*, pages 12–21. IEEE, 1992.

[2] James H. Anderson and Yong-Jik Kim. An improved lower bound for the time complexity of mutual exclusion. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 90–99, New York, NY, USA, 2001. ACM Press.

[3] James H. Anderson and Yong-Jik Kim. Nonatomic mutual exclusion with local spinning. In *PODC '02: Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 3–12, New York, NY, USA, 2002. ACM Press.

[4] James H. Anderson, Yong-Jik Kim, and Ted Herman. Shared-memory mutual exclusion: major research trends since 1986. *Distributed Computing*, 2003.

[5] Hagit Attiya and Danny Hendler. Time and space lower bounds for implementations using -cas. In *DISC*, pages 169–183, 2005.

[6] James E. Burns and Nancy A. Lynch. Bounds on shared memory for mutual exclusion. *Information and Compututation*, 107(2):171–184, 1993.

[7] Robert Cypher. The communication requirements of mutual exclusion. In *SPAA '95: Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 147–156, New York, NY, USA, 1995. ACM Press.

[8] G. Graunke and S. Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 1990.

[9] Prasad Jayanti. A time complexity lower bound for randomized implementations of some shared objects. In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 201–210, New York, NY, USA, 1998. ACM Press.

[10] Patrick Keane and Mark Moir. A simple local-spin group mutual exclusion algorithm. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 23–32, New York, NY, USA, 1999. ACM Press.

[11] J. Mellor-Crummey and M. Scott. Algorithms for scalable sychronization on shared-memory multicomputers. *ACM Transations on Computer Systems*, 1991.

[12] Michael Raynal. *Algorithms for Mutual Exclusion.* The MIT Press, Cambridge, Massachusetts, 1986.

[13] Y.-H. Yang and J. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 1995.

[10] Note in fact that $\frac{\sum_{\pi \in S_n} |E_\pi|}{|S_n|} = \Omega(n \log n)$.