Timing-Based Mutual Exclusion*

Nancy Lynch† Nir Shavit‡

Abstract

Known asynchronous n-process deadlock-free mutual exclusion algorithms require $O(n)$ read/write registers and $O(n)$ operations to access the critical region, rendering them impractical for large scale applications. Burns and Lynch have shown that n registers are necessary for solving this problem, in the asynchronous setting.

This paper examines the benefits that can be obtained by using *timing information* in mutual exclusion algorithms. First, a simple and efficient timing-based mutual exclusion algorithm is given. This algorithm always guarantees mutual exclusion (i.e., even when run asynchronously), and also avoids deadlock in case certain (realistic) inexact timing constraints are met. The algorithm uses only *two* shared read/write registers (a total of $\log n + 1$ bits), thus "overcoming" the n register lower bound for asynchronous algorithms. It is proved that the problem cannot be solved with only one shared register, so that this algorithm is optimal in terms of the number of registers.

Second, a lower bound is proved for the time complexity of any deadlock-free mutual exclusion protocol, as a function of the number of shared registers it employs. This bound shows that the algorithm described above is near optimal, in terms of time complexity. Finally, it is shown that two natural ways of weakening the timing assumptions lead (unfortunately) to an n register lower bound.

*This work was supported by ONR contracts N00014-85-K-0168 and N00014-91-J-1046, by NSF grants CCR-8611442 and CCR-8915206, by DARPA contracts N00014-87-K-0825 and N00014-89-J-1988, and by Draper Laboratories grant DL-H-441640

†Laboratory for Computer Science, MIT, Cambridge, MA 02139.

‡Laboratory for Computer Science, MIT, and Tel-Aviv University, Tel-Aviv 69978, Israel.

1 Introduction

Although current-day multiprocessors provide sophisticated synchronization primitives for mutual exclusion, situations that require a register-based exclusion algorithm continue to arise [1, 2]. Known mutual exclusion algorithms [3, 15, 16, 18] require that both the number of read/write registers and the number of operations a process performs to access the critical region be at least equal to the total number of processes in the system. The lower bound of Burns and Lynch [3] implies that this linear complexity is optimal for deterministic algorithms, even if multi-writer registers are used. This implies that such algorithms cannot be expected to perform well in systems that support thousands of concurrent processes [1, 2].

Recently, several researchers [4, 5] have proposed ways of overcoming the linear lower bound on the number of operations a process needs to perform. For example, employing operating system support in a sophisticated way at process creation time, Merritt and Taubenfeld provide an algorithm that limits the number of operations by a process to depend only on the number of processes concurrently accessing the critical region. However, in both of these cases, the number of shared registers the algorithm requires is still at least equal to the total number of processes in the system. Also, using randomization, and based on the work of Lynch and Saias [13], Kushilevits and Rabin [14] offer a new (but rather involved) algorithm that employs randomization to achieve fair mutual exclusion using a single $O(\log \log n)$-bit shared register. However, this solution uses a powerful n-process *Test-and-Set* shared variable, not a read/write register.

In this paper, we consider the mutual exclusion problem in a *real-time* setting. More particularly,

we examine the benefits that can be obtained by using available *timing information* in mutual exclusion algorithms.

Fischer [10] appears to have been the first to propose overcoming the Burns-Lynch lower bound of n shared registers for deadlock-free mutual exclusion by assuming timing constraints. He has presented a truly simple mutual exclusion algorithm that uses only a single shared multi-writer register. However, his algorithm has a major drawback: it fails to guarantee mutual exclusion if the timing constraints are not met. This is also the drawback of the elegant timing-based algorithm of Alur and Taubenfeld [6]. Ideally, a timing-based algorithm should not depend on its timing constraints to guarantee basic safety properties, but only to ensure liveness and performance properties.

In this paper, we give a simple and efficient timing-based mutual exclusion algorithm that *always* guarantees mutual exclusion (i.e., even when run asynchronously), and also avoids deadlock in case certain inexact timing constraints are met. Specifically, we assume that the times between successive steps of the same process are always in some known interval, $[c_1, c_2]$, of real times. This assumption is realistic in the real-time setting, if processes do not get swapped out during the time when they are executing the protocol, or even if they do get swapped out, but the underlying processor scheduling algorithm ensures approximately predictable execution speeds. (In fact, the algorithms we present require that processes do not get swapped out only between two specific lines of code, and not throughout the complete protocol.) The algorithm uses only *two* shared read/write registers (a total of $\log n + 1$ shared bits), thus "overcoming" the n register lower bound for asynchronous algorithms. We also prove that the problem cannot be solved with only one shared register, so that our algorithm is optimal in terms of the number of registers.

Next, we prove a lower bound for the time complexity of any deadlock-free mutual exclusion protocol, as a function of the number of shared registers it employs. The proof uses the techniques of [3], extended to the real-time setting. This bound shows that our algorithm is near optimal, in terms of time complexity.

```
Process i:
repeat forever
        remainder region
        trying region
        critical region
        exit region
end repeat;
```

Figure 1: Program of Process i.

Finally, we show that two natural ways of weakening the timing assumptions lead, unfortunately, to an n register lower bound. Specifically, if we assume that the time bounds $[c_1, c_2]$ are fixed, and always hold, but are *unknown to the processes*, or that the timing conditions are only *eventually* met, then we obtain a lower bound of n on the number of read/write registers required.

2 The Mutual Exclusion Problem

We consider a system of n sequential threads of control called *processes*, which communicate through shared *multi-writer, multi-reader* atomic registers. Communication consists of *read* and *write* operations, each of which is assumed in this paper to be executed instantaneously. In the usual model for atomic registers each operation is separated into an *invocation* event and a *response* event, concurrent invocations are allowed, and it is assumed that every invocation eventually terminates in a matching response, in such a way as to produce the illusion of instantaneous operation. Our work applies to this setting, but we do not need to deal with this extra complication here.

Formally, we model each such system as a single *timed automaton* [7] (A, b), where A is a composition of I/O automata [8], and b is a boundmap. We keep the discussion in this ab-

stract informal and refer the interested reader to [8, 9, 7, 20] for a complete presentation of the model and its properties.

We view the program of every process as consisting of two distinguished regions: a *remainder region* and a *critical region.* Each process alternates between executing its remainder and its critical region. No process remains forever within any particular execution of its critical region, although it may remain forever in its remainder region. The *mutual exclusion problem* is to guarantee that the system does not enter a global state in which more than one process is executing its critical region [15].

To coordinate the entrance to the critical region, a *trying region* and an *exit region* of code are added to the program of each process as in Figure 1.

The following safety property is required of any solution to the problem.

Mutual Exclusion: A system satisfies *mutual exclusion* if in any reachable system state, at most one user is in its critical region.

We also require that a process running alone will always get access to the critical region.

Weak Deadlock-Freedom: A system is *weakly deadlock-free* if in any execution, if a process' trying region (resp. exit region) is concurrent only with other processes' remainder regions, then its trying region (resp. exit region) terminates.

The following property guarantees progress, and will be required to hold only under certain conditions.

Deadlock-Freedom: A system is *deadlock-free* if the following are true in any execution.

1. If in some state, some user is in the trying region and no user is in the critical region, then subsequently some user enters the critical region.
2. If in some state, some user is in the exit region then subsequently some user enters the remainder region.

---

x, y: *shared registers, initially* $y = 0$;

repeat forever
0b: *remainder_exit*$_i$
L: $x := i$;
1: **if** $y \neq 0$ **then goto** L;
2: $y := 1$;
3: **if** $x \neq i$ **then goto** L;
4a: *critical_entry*$_i$
4b: *critical_exit*$_i$
5: $y := 0$;
0a: *remainder_entry*$_i$
end repeat;

Figure 2: Algorithm 1: Lamport Style Mutual Exclusion – Code for Process i.

We define a time-bounded version of the deadlock-freedom property. This involves putting an upper bound on the time processes can spend executing the added *trying* and *exit* regions of code. We define a number b to be an *upper bound* for a *deadlock-free* system providing that the deadlock freedom definition above holds if we replace "subsequently" with "within time at most b."

3 The Asynchronous Case

The following theorem is due to Burns and Lynch [3].

Theorem 3.1 *There is no asynchronous algorithm providing mutual exclusion with deadlock-freedom for $n \geq 2$ processes, using fewer than n shared read/write registers.*

Since Theorem 3.1 rules out asynchronous solutions for deadlock-free mutual exclusion with fewer than n shared registers, we will consider timing-based algorithms. A natural requirement for a timing-based mutual exclusion algorithm is that mutual exclusion and weak deadlock-freedom be guaranteed always, even if the timing assumptions are violated. We thus begin our development by showing that there exists an

asynchronous 2-register algorithm for $n \geq 2$, giving mutual exclusion and weak deadlock-freedom. This algorithm, which we call Algorithm 1, has a structure similar to some described by Lamport in [17]. The algorithm uses two shared registers, x and y. Each process P_i writes its index i into x, to indicate that it wishes to enter the critical region, and then checks register y to see if it might have just overwritten the value of some other process P_j that is attempting to enter. If there is such a process (i.e., if $y = 1$), P_i gives up, otherwise, it sets y for other processes and tests x to see if it has not been overwritten by some later process P_j attempting to enter the critical region. If x is still set to i, P_i enters the critical region, setting y to 0 before returning to the remainder region.

The code is presented in Figure 2, in a standard psuedo-code style. Translating this code into timed automata is a straightforward but tedious exercise. The other algorithms in this paper are also presented in the same style.

To aid in the proof, we describe the program states as a combination of *program counter* values as above, local variable values and shared register values. In any state of the algorithm, we define:

$F1$: the set of process indices i such that $x = i$ and $pc(i) = 2$,

$F2$: the set of process indices i such that $x = i$ and $pc(i) = 3$,

CR: the set of process indices i such that $pc(i) \in \{4a, 4b, 5\}$.

RM: the set of process indices i such that $pc(i) \in \{0a, 0b\}$.

We prove the following lemma by induction on the length of the execution.

Lemma 3.2 *The following are true in any reachable state of Algorithm 1.*

$I1$: $|F1| + |F2| + |CR| \leq 1$.

$I2$: *If* $|F2| + |CR| > 0$ *then* $y = 1$.

$I3$: *If* $y = 1$ *then some process* i *is not in* RM.

By $I1$ of Lemma 3.2 we have:

Lemma 3.3 *Algorithm 1 has the mutual exclusion property.*

The following lemma follows from $I3$ and the fact that the executing the exit region takes a bounded nember of steps.

Lemma 3.4 *Algorithm 1 is weakly deadlock-free.*

We now show that Algorithm 1 optimal in terms of the number of registers.

Lemma 3.5 *In any (timed or asynchronous) algorithm providing mutual exclusion and weak deadlock-freedom for $n \geq 2$ processes, using a single shared register, every process that moves from its remainder to its critical region must perform at least one read followed by a write.*

The proof follows since if a process does not write, other processes cannot tell that it has left the remainder, and if it does not read, it cannot tell if any one of them left the remainder. Both cases would lead to a violation of mutual exclusion.

Theorem 3.6 *There is no asynchronous algorithm providing mutual exclusion with weak deadlock-freedom for $n \geq 2$ asynchronous processes, using only one shared read/write register.*

Proof: Assume by way of contradiction that there exists a one register algorithm for the problem. Consider two processes, say 1 and 2. From the system start state s, by the *weak deadlock-freedom* property, either process must be able, on its own, to complete an execution leading to a state in which it is in the critical region; call these executions e_1 and e_2 respectively. By Lemma 3.5, each of the two processes must write the register during its execution.

Now we construct a bad execution. Run process 1 in e_1 just until the first state s' at which it is about to write the register, and let it pause without doing so. States s and s' differ only in the internal state of process 1. Then extend the execution by e_2, in which process 2 behaves just as if

```
x: shared register initially 0;
delay: positive integer constant;

repeat forever
0b: remainder_exit_i
L:  if x ≠ 0 then goto L;
1:  x := i;
2:  pause(delay);
3:  if x ≠ i then goto L;
4a: critical_enter_i
4b: critical_exit_i
5:  x := 0;
0a: remainder_enter_i
end repeat;
```

Figure 3: Algorithm 2: Fischer's Timed Mutual Exclusion – Code for Process i.

it were on its own, eventually entering the critical region. Next, process 1 continues its execution of e_1, first performing its write to the register, which overwrites whatever process 2 has written. Following this operation, the state of the system differs from the corresponding state in e_1 only in the internal state of process 2, and so process 1 goes critical, violating *mutual exclusion.* This is a contradiction. ■

4 The Timing-Based Case

In this section, we consider the *real-time* version of the problem, where we assume bounds in the interval $[c_1, c_2]$ for the time between successive steps of a process, when it is in its trying or exit region. We assume that $0 < c_1 \leq c_2 < \infty$. Let $C = c_2/c_1$; C is a measure of the *timing uncertainty.* Define a process i to be *slow* (respectively, *fast*) in an execution or execution fragment if the time for any step when the process is in its trying or exit region is exactly c_2 (resp., c_1).

We begin by presenting, in Figure 3, a timing-based mutual exclusion algorithm due to Mike Fischer [10]; we call this Algorithm 2. Algorithm 2 has the property that if the timing constraints are met, and the value of the *delay* parameter is chosen to be strictly greater than the timing uncertainty C, then mutual exclusion and deadlock-freedom are guaranteed. However, it has the drawback that if the constraints are not met, then many processes may enter the critical region concurrently.

The key idea behind Algorithm 2 is to delay each process i for a period greater than $C \cdot c_1$ after it has written x in Line 1 and before testing it in Line 3. Thus, by the time process i has reached Line 3, any process j that has passed the test in Line L and might overwrite $x = i$ with $x = j$, has already done so since $C \cdot c_1 \geq c_2$, the longest time such a step might take. If i reads $x = i$ in Line 3, it can safely enter the critical region, since all other processes are either before Line L and will not pass it, or after Line 1 with their index in x overwritten by process i, so they will fail the test on Line 3. To implement the delay, the procedure *pause*(*delay*) causes the process to delay by some number of steps, where a *delay* of 1 means skipping no steps at all, and a *delay* of k amounts to doing $k - 1$ dummy steps. Specifically, let *pause*(k) be a shorthand for a sequence of $(k - 1)$ lines of code, each a no-op operation, and let us accordingly assign program counter values $2.1, 2.2, \ldots, 2.(k-1)$ to those $(k - 1)$ lines.

We now outline the correctness proof of the algorithm. Let Algorithm 2' be the same as Algorithm 2 but asynchronous . The following lemma follows by induction on the length of the execution.

Lemma 4.1 *Then in any reachable state of Algorithm 2':*

$$\textit{If } x = i \neq 0 \textit{ then } pc_i \in \{2, 3, 4a, 4b, 5\}.$$

To aid in the main proof, let us define several state components involving time: *now*, representing the current real time, and for each process i, $ftime(i)$ and $ltime(i)$, representing the first and last (absolute) time that the next step of process i is allowed to occur. From our definitions, it follows that if $pc(i) \notin \{0b, 4b\}$ then $ltime(i) \leq now + c_2$.

Let CR be defined similarly to before. We prove the following collection of invariants by in-

duction on the number of steps in an execution. They are similar to those proposed in [19] for a weaker form of the mutual exclusion problem. Note that the only executions that are considered here are those in which the timing assumptions are satisfied; the timing assumptions are required to guarantee these key properties.

Lemma 4.2 *In any reachable state of Algorithm 2, the following are true.*

I1: *If* $i \in CR$ *then*

1. $x = i$, *and*
2. *for all* j, $pc(j) \neq 1$.

I2: *If* $pc(i) = 2.d$, $x = i$ *and* $pc(j) = 1$ *then* $ltime(j) < ftime(i) + (delay - d)c_1$.

I3: *If* $pc(i) = 3$, $x = i$ *and* $pc(j) = 1$ *then* $ltime(j) < ftime(i)$.

From *I*1 of Lemma 4.2 we have:

Lemma 4.3 *Algorithm 2 has the mutual exclusion property.*

We prove the following based on Lemma 4.1.

Lemma 4.4 *Algorithm 2' is deadlock-free.*

The following is an almost tight upper bound on the time b for some process to enter the critical region. (It can be made tighter by a slight modification to Algorithm 2, which we will describe in the full version of the paper.)

Lemma 4.5 *Suppose that* $delay = C + 1$. *Then Algorithm 2 has an upper bound of* $(2C + 7)c_2$.

The following theorem proves the near optimality of the time complexity of Algorithm 2. Our original bound of Cc_2 was strengthened by Faith Fich [11].

Theorem 4.6 *There is no* $[c_1, c_2]$ *timed algorithm providing mutual exclusion with weak deadlock-freedom for* $n \geq 2$ *asynchronous processes, that uses only one shared read/write register, and that takes time* $b < (C + 3)c_2$.

Proof: (Sketch) Consider two processes, say 1 and 2. From the system start state s, by the *weak deadlock-freedom* property, either process must be able, on its own, to complete an execution leading to a state in which it is in the critical region. Call these executions e_1 and e_2, respectively. By Lemma 3.5 each process must first read and then write the shared register in its execution. Let r_1 and r_2 be the number of steps prior to the first write of the shared register in e_1 and e_2, respectively (by Lemma 3.5, r_1 and r_2 are greater than 0).

Suppose that one of these executions, say e_1, contains at most C steps after its first write to the shared register. Let $t = \max\{c_1(r_1 + 1), c_2(r_2)\}$ and consider the execution e formed by running e_1 fast starting at time $t - c_1(r_1 + 1)$ and running e_2 slowly starting at time $t - c_2(r_2)$. In e, the r_2th step of e_2 occurs at time t, followed immediately (at the same real time) by the first write of e_1, then the rest of e_1 until time $t + Cc_1$ and finally the rest of e_2, starting with the first write in e_2.

In e, all of e_1 occurs before the first write of e_2. Therefore process 1 cannot distinguish between executions e and e_1. Furthermore, all of the writes of e_1 occur between the first write of e_2 and the read that precedes it. Thus process 2 cannot distinguish between executions e and e_2. But e violates mutual exclusion.

It follows that both e_1 and e_2 contain at least $C+3$ steps. Then if either execution is run slowly by itself, at least $(C+3)c_2$ time will elapse before the critical region is entered. ■

5 The Combined Algorithm

In this section we give our first main result: we show how to overcome the drawbacks of the Fischer algorithm, Algorithm 2, providing an algorithm that always provides mutual exclusion and weak deadlock-freedom, and also provides deadlock-freedom if the timing constraints $[c_1, c_2]$ are met. The algorithm that accomplishes this uses only two shared registers and has a "nice" time bound close to that of Algorithm 2.

By combining Algorithms 1 and 2, one can cre-

```
x, y: shared registers initially 0;
delay: positive integer constant;

repeat forever
0b: remainder_exit_i
L:  if x ≠ 0 then goto L;
1:  x := i;
2:  pause(delay);
3:  if x ≠ i then goto L;
    % Start of Fischer Critical Region
4:  if y ≠ 0 then goto L;
5:  y := 1;
6:  if x ≠ i then goto L;
7a: critical_enter_i
7b: critical_exit_i
8:  y := 0;
    % End of Fischer Critical Region
9:  x := 0;
0a: remainder_enter_i
end repeat;
```

Figure 4: Algorithm 3: Combined Exclusion Algorithm – Code for process i.

ate an algorithm that uses three shared registers. The way to do this is to replace the critical region of Algorithm 2, by Algorithm 1. However, it is also possible to get a two register algorithm, by combining the two algorithms in a more elaborate way, as seen in Figure 4. The construction is based on the observation, implied immediately by Lemma 4.2, that in Algorithm 2, once a process i enters the critical region, the register x remains equal to i. This implies that the test of x in Algorithm 1 will not fail if it is embedded in Algorithm 2's critical region *even if Algorithm 2 allows processes to set the same register.*

To prove correctness of Algorithm 3, we let Algorithm 3' be the same as Algorithm 3, but without the timing constraints. We show the following by induction on the length of the computation.

Lemma 5.1 *In any reachable state of Algorithm 3', the following are true.*

1. *I1: If* $|RM| = n$ *then* $x = y = 0$.
2. *I2: If all processes* $i \notin RM$ *have* $pc(i) = 9$, *then* $y = 0$.

Then we define a forward simulation mapping [12] from Algorithm 3' to Algorithm 1. From this we prove that:

Lemma 5.2 *Algorithm 3' satisfies the mutual exclusion and weak deadlock-freedom properties.*

That is, Algorithm 3 has these properties even if the timing constraints are not satisfied.

We prove the following two lemmas by defining a timed forward simulation [12] from Algorithm 3 (with the timing constraints) to the Fischer algorithm, Algorithm 2, and then proving it and the following invariant by induction on the length of the computation:

I3: If for all i, $pc(i) \notin \{6, 7a, 7b, 8\}$ then $y = 0$.

Lemma 5.3 *Algorithm 3 (with the timing constraints) is deadlock-free.*

Lemma 5.4 *Algorithm 3 (with the timing constraints) has an upper bound of* $(2C + 10)c_2$ *time.*

6 Adding More Registers

In this section, we present our second main result, a lower bound on the time complexity of any deadlock-free mutual exclusion algorithm, in the timed setting. Our lower bound is expressed as a function of the number k of shared registers used by the algorithm. Specifically, it is of the form $e(k)Cc_2$, where $e(k)$ is a function of k. The value of $e(k)$ decreases rapidly with k. Thus, the bound is probably only interesting when k is small and when C, the timing uncertainty, is large. Note that we do *not* yet have any corresponding upper bounds that decrease similarly with increasing k; finding algorithms exhibiting such upper bounds (or removing the dependency of the lower bound on the number of registers) remains as future work.

For a particular k, we define $e(k)$ by means of a recursive definition of a fast-growing function $f(j, k)$, $1 \leq j \leq k$. Specifically, we define

$f(1,k) = 1$, and $f(j+1,k) = (\binom{k}{j} + 1)f(j,k) + \binom{k}{j}(2j+1)$, $1 \leq j \leq k-1$. Then we define $e(k) = 1/(f(k,k)+1)$. Thus, $e(k)$ is a rapidly-decreasing function of k.

Lemma 6.1 $1 \geq f(j,k)e(k) + e(k)$ *for all* j, $1 \leq j \leq k$.

Theorem 6.2 *Suppose* $C \geq 2$. *There is no* $[c_1, c_2]$ *timed algorithm providing mutual exclusion with deadlock-freedom for* $n \geq 2$ *asynchronous processes, that uses only* $k < n$ *shared read/write register, and that takes time* $b < e(k)Cc_2$.

Proof: (Sketch) Suppose there is such an algorithm, with time bound $b < e(k)Cc_2$. We follow the main ideas of the construction of Burns and Lynch [3], for the asynchronous impossibility proof (of our Theorem 3.1) to construct a bad execution. This time, however, the bad execution, as well as the auxiliary executions used along the way, must satisfy the timing constraints, which makes the construction considerably harder than before.

The construction we carry out involves only $k+1$ of the n processes, those numbered $p_1, p_2, \ldots, p_{k+1}$. For any $j = 1, \ldots, k$, starting from any configuration in which all processes are in their remainder regions, we obtain a finite timed execution of $p_1, \ldots, p_j$ only, satisfying the timing constraints, and leading to a point at which $p_1, \ldots, p_j$ "nullify" j distinct registers. (The definition of a process p_i "nullifying" a register v is as in [3]: p_i is about to write register v at its next step, and moreover, anything process p_i has written since last leaving its remainder region has already been overwritten without being read by other processes.) Moreover, the execution has the additional timing property that for all i, $1 \leq i \leq j$, $ltime(i) \geq now + e(k)c_2$ and $ftime(i) \leq now$. That is, the timing constraints permit process i to wait at least time $e(k)c_2$ before taking its next step, but do not impose any lower bound on the time at which it might take that step. Furthermore, the total time required for the execution is at most $f(j,k)e(k)c_2$.

We apply this result for the special case where $j = k$. Now let the last process, p_{k+1}, enter the system, taking steps as fast as it can. We claim it should be able to proceed to its critical region before any of the other processes take their next steps. This is because of the upper bound on the algorithm, the fact that p_{k+1} cannot tell that any other process is outside its remainder region, and the fact that $ltime(i) \geq now + e(k)c_2$ for all i, $1 \leq i \leq k$. (The upper bound is $e(k)Cc_2$ for all computations that satisfy the timing constraints; we can stretch the given fast computation of p_{k+1} so that the time between its steps is always c_2, and an upper bound of $e(k)Cc_2$ for this slower computation implies an upper bound of $e(k)c_2$ on the fast computation.) But then we can let p_{k+1} remain in its critical region while the rest of the processes proceed normally, still observing the timing assumptions. We first allow all the processes to write the registers they are nullifying. Once they have done this, they have hidden all information about p_{k+1} being in the critical region, so some other process will eventually proceed into its critical region, thereby violating mutual exclusion. This is a contradiction.

It remains to carry out the recursive construction. Basis: $j = 1$. Just let p_1 enter, going fast. Within time at most $e(k)c_2$, it must go to the critical region, and in order to do so, it must write some register, say v. Let p_1 stop at a particular point in this interval, when it is about to write some v for the first time. It is hidden, and ready to write, so it nullifies v. Moreover, stopping just when time c_1 has elapsed after the prior step ensures that $ftime(1) \leq now$, and since $ltime(1) = now + c_2 - c_1$, since $C \geq 2$ we also have $ltime(1) \geq now + e(k)c_2$. The total time required for this is at most $e(k)c_2 = f(1,k)e(k)c_2$, which suffices.

Inductive step: Suppose we have the result for $j \leq k-1$. Starting from any configuration in which all processes are in their remainder regions, we wish to obtain the analogous timed execution for $p_1, \ldots, p_{j+1}$. As in [3], we construct a "spine" execution; this time, the spine execution will satisfy the timing constraints. Constructing

the spine involves applying the inductive hypothesis to get $p_1, \ldots, p_j$ nullifying j registers, with all *ltime* values at least $now + e(k)c_2$ and all *ftime* at most *now*, all within time at most $f(j,k)e(k)c_2$. Then we repeatedly do the following:

Allow all of $p_1, \ldots, p_j$ to write at time exactly $e(k)c_2$ after the stopping point, with no other intervening steps. Then use the deadlock-freedom hypothesis to allow all these processes to proceed to their critical regions, one at a time, and from there, to their remainder regions. (Each will only spend time 0 in the critical region.) Do this using a fast execution fragment, so the upper bound for each process to go critical is $e(k)c_2$, and then to go to its remainder region is another $e(k)c_2$. After some time, all will be back in their remainder region. Then apply the inductive hypothesis once again to get $p_1, \ldots, p_j$ nullifying j registers, with all *ltime* values at least $now + e(k)c_2$ and all *ftime* values at most *now*, again within time at most $f(j,k)e(k)c_2$.

So, we repeatedly have $p_1, \ldots, p_j$ nullifying a set of j registers. Sometime within the first $\binom{k}{j}+1$ times this happens, the same set of j registers will be nullified. Call this set V. We claim that the time until the second point where V is nullified is at most $(\binom{k}{j}+1)f(j,k)e(k)c_2 + \binom{k}{j}(2j+1)e(k)c_2$. The first term in this sum describes the total time taken for all the uses of the inductive hypothesis. The second term describes the total time for the processes to overwrite and then to proceed back to their remainder regions. Note that by definition of f, this time bound expression is equal to $f(j+1,k)e(k)c_2$.

Now we use this spine to help us construct the desired execution of $p_1, \ldots, p_{j+1}$. We run the spine just until the first time V described above is nullified. Then we insert some steps of p_{j+1}, running fast, just until it is about to write something not in V. This must happen within time $e(k)c_2$, since p_{j+1} doesn't know there is anyone else there and must go critical by $e(k)c_2$ in a fast execution in which it operates alone. It must write something not in V, since otherwise its writes could be overwritten by the nullifying processes $p_1, \ldots, p_j$, who would thereby hide p_{j+1} in the critical region and lead to a violation of mutual exclusion. Just before it writes something not in V, we stop p_{j+1} and now allow the rest of the spine to proceed as before, just until the second time that V is nullified. We claim that we are allowed to stop p_{j+1} for that length of time, because the length of time for the spine is at most $f(j+1,k)e(k)c_2$, the upper bound on step time for p_{j+1} is c_2, and $f(j+1,k)e(k)c_2 \leq c_2$. (This latter fact follows from the fact that $1 \geq f(j+1,k)e(k)$, which in turn follows from Lemma 6.1.)

We claim that the resulting execution has the required properties. The inductive hypothesis already tells us that the *ftime* and *ltime* values for processes $p_1, \ldots p_j$ are as required. We must show that the values for p_{j+1} are also appropriate. Since at least two steps have occurred for some particular process since p_{j+1} was stopped, it must be that $ftime(j+1) \leq now$. It remains to consider the value of $ltime(j+1)$. To see that $ltime(j+1) \geq now + e(k)c_2$, it suffices to show that $c_2 \geq f(j+1,k)e(k)c_2 + e(k)c_2$. But Lemma 6.1 implies that $1 \geq f(j+1,k)e(k) + e(k)$, which immediately implies this inequality. ■

7 Weaker Timing Assumptions

We show that weakening the timing model by assuming that the bounds $[c_1, c_2]$ are unknown, or that the timing conditions are only *eventually* met, leads to a lower bound of n on the number of read/write registers required, just as in the asynchronous case.

In the *resilient* timing model, we again consider a fixed pair, $[c_1, c_2]$, of bounds for the time between steps in the trying or exit region, as in the timed model. However, now we only consider those executions in which the time bounds hold *eventually*. We prove the following theorem by a variation of the [3] proof.

Theorem 7.1 *There is no $[c_1, c_2]$ resilient algorithm providing mutual exclusion with deadlock-freedom for $n \geq 2$ asynchronous processes that uses fewer than n shared read/write registers.*

In the *unknown bound* timing model, we con-

sider those executions in which there exist some time bounds $[c_1, c_2]$, $0 < c_1 \leq c_2 < \infty$, that hold throughout the execution. However, the time bounds are allowed to be *different* in different executions, in other words, unknown to the processes. We prove the following theorem by a reduction from the previous result.

Theorem 7.2 *There is no unknown bound algorithm providing mutual exclusion with deadlock-freedom for $n \geq 2$ asynchronous processes that uses fewer than n shared read/write registers.*

8 Acknowledgments

Many thanks go to Faith Fich for her interest and attention, and for many helpful suggestions on the results and proofs. We also thank Victor Luchangco who helped with the proof of Lemma 4.2.

References

[1] M. R. MacBlane. Source level atomic test-and-set for the TUXEDO system source product. UNIX System Laboratories, May 14, 1991.

[2] *TUXEDO System Release 4.0 – Product Overview.* AT&T, 1990.

[3] J. Burns and N. Lynch. Mutual exclusion using indivisible reads and writes. In *Proceedings of 18th Annual Allerton Conference on Communications, Control and Computing*, 1989, pages 833-842.

[4] E. Styer. Improving fast mutual exclusion. Manuscript.

[5] M. Merritt and G. Taubenfeld. Speeding Lamport's fast mutual exclusion algorithm. Manuscript.

[6] R. Alur and G. Taubenfeld. Results about fast mutual exclusion In *Proceedings of Real Time Systems Symposium*, Pheonix, Arizona, 1992.

[7] M. Merritt, F. Modugno and M. Tuttle. Time-Constrained Automata. In *Proceedings of 2nd CONCUR*, Amsterdam, The Netherland, August, 1991, Springer-Verlag LNCS 527, pp. 408-423.

[8] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th PODC*, August 1987, pp. 137-151.

[9] N. Lynch and M. Tuttle. An introduction to Input/Output Automata. *CWI-Quarterly*, No. 3, Vol. 2, September, 1989, pp. 219-246.

[10] M. Fischer. Personal communication.

[11] F. Fich. Personal communication.

[12] N. Lynch and F. Vaandrager. Forward and backward simulations for timing-based systems. In *Proceedings of REX Workshop "Real-Time: Theory in Practice,"* Mook, The Netherland, 1992. Springer-Verlag LNCS 600.

[13] N. Lynch and I. Saias. Proving probabilistic correctness statements: the case of Rabin's algorithm for mutual exclusion. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, 1992.

[14] E. Kushilevits and M. Rabin. Randomized mutual exclusion algorithms revisited. In *Proceedings of the 11th PODC*, 1992.

[15] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications Of The ACM*, 8:165, 1965.

[16] H. Katseff. A new solution to the critical section problem. In *Proceedings of the 10^{th} Annual ACM Symposium on Theory of Computing*, pages 86–88. ACM, 1978.

[17] L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. on Comp. Systems*, 5(1):1-11, Feb. 87.

[18] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 78(8):453–455, 1974.

[19] M. Abadi and L. Lamport. An old fashioned Recipe for Real Time In *Real Time: Theory in Practice, REX Workshop*, Springer Verlag, pp. 1-27, 1991.

[20] H. Attiya and N. Lynch. Time bounds for real-time process control in the presence of timing uncertainty. In *Proceedings of the 10th RTSS*, Santa-Monica, December 1989, pp. 268–284.