

7.2 Mutual Exclusion with the Test&Set Registers

The Test&Set register is a binary variable which supports two atomic operations **test&set(v)** and **reset(v)**. The variable v is initialized to 0.

Algorithm 8 Mutual Exclusion Using test&set Register(Continue)

Entry Section

1 if ($test\&set(v) = 1$)

2 goto line 1;

Critical Section

EXIT

3 $reset(v)$;

REM

We have the following theorem to prove that algorithm.8 provides mutual exclusion, no deadlock property. But this algorithm is not starvation free.

Theorem 7.2. *The test&set algorithm provides mutual exclusion.*

Proof. By contradiction, suppose more than one processor is in the critical section at the same time. Let p_i and p_j be the first pair of processors in the critical section. Without loss of generality, suppose p_i did **test&set** first. According to the algorithm.7.1, it must read 0 in line L , then the value of v after the reading of p_i will be 1. Because our assumption implies that no other processor will enter and exit the critical section between p_i and p_j , so there will be no execution of *reset* operation, which means that the value of v will be kept as 1 when p_j tries to read it. So it is impossible for p_j to enter the critical section. This is the contradiction. \square

Theorem 7.3. *The test&set algorithm provides no deadlock.*

Proof. By contradiction, suppose that there is a time t such that p_i is in entry section at time t but no processor is in critical section after time t . Since a processor sets v to 1 only before entering the critical section, it follows that no processor sets v to 1 after time t . Now, either no processor ever entered the critical section or else the processor to enter the critical section exited at time $t' < t$. In the first case, $v = 0$ at all times and in the second case, the processor to exit reset v to 0 at time t' . In either case, $v = 0$ at all times after time t . So some processor in entry section must read $v = 0$, it can enter the critical section, this is a contradiction. \square

The algorithm **does not** provide **no starvation**. It is easy to build the following scenario in table.1, the p_0 is slow enough, so that it always read $v = 1$ at line 1.

Table 1: A scenario that might cause the starvation

p_0	p_1
Line 1: read 1 and being trapped	Line 1: read 0 and enter the CS
	Line 3: reset v
Line 1: read 1 and being trapped	Line 1: read 0 and enter the CS

7.3 Mutual Exclusion with the Read-Modify-Write (RMW) Registers

The *Read-Modify-Write* register V is a variable which allows the processor to read the current value of the variable, compute a new value as a function of the current value, and write the new value to the variable, all in one atomic operation. This can be described as $temp := RMW(v, f(v))$. The *test-set* register is a special case of RMW, where $f(0) = 1$ and $f(1) = 1$.

We now use *RMW* to devise an algorithm which achieves mutual exclusion with no starvation. using only one *RMW* register. The algorithm organizes processors into a FIFO queue, allowing the processor at the head of the queue to enter the critical section. The register v used by the algorithm is a two-tuple $(first, last)$, where $first$ denotes the ticket of the processor at the head of the queue, which is eligible to enter the critical section, and $last$ denotes the ticket of the last processor in the queue, which is the last processor having entered the entry section. Each processor p_i maintains two local variables,

Algorithm 9 Mutual Exclusion Using a RMW Register

Entry Section

```

1   $position_i := RMW(v, (first(v), last(v) + 1));$ 
2   $queue_i := RMW(v, v);$ 
3  if  $first(queue_i) \neq last(position_i);$ 
4    then goto  $line_{frm-e};$ 
```

Critical Section

EXIT

```

5   $RMW(v, (first(v) + 1, last(v)));$ 
```

REM

This algorithm uses one shared variable which can take n^2 shared memory states, where n is the number of processors. We talk in terms of number of shared memory states rather than the number of variables since we can encode any number of states in just 1 **RMW** variable (note that we could do this with **RM** variables). So talking about the number of shared memory states gives us a better measure for the complexity of the algorithm.

In the following, we will prove the safety and liveness properties formally. In particular, to prove the mutual exclusion and no starvation properties, it is easier to first prove the following fairness property:

Fairness: Processors enter the critical section in the same order as they execute the line 1 of Algorithm.9.

Lemma 7.4. *The values of both $first(v)$ and $last(v)$ are non-decreasing. Specifically, each is incremented by one after each time modified.*

Lemma 7.5. *Two Processors cannot have the same ticket.*

Lemma 7.6. *If there is a processor with ticket L , then there is a processor with ticket $L+1$.*

Theorem 7.7. *Let processor p_i and p_j have ticket k_i and k_j respectively, where $k_i < k_j$. Then p_i will enter the critical section before p_j . In other words, the algorithm satisfies the **Fairness Condition**.*

Proof. For p_j to enter the critical section, it must have read the value k_j in the component $first$ of the RMW object v at time t_j . Because of Lemma.7.4, the $first(v)$ is non-decreasing, and the only way that it is modified is that it is incremented by 1 in line 5 after it is found to be equal to some ticket. Consider the fact that the initial value of v is $(1, 1)$, during the increment process from 1 to k_j , it must pass the value of k_i at some t_i before t_j , since $k_i < k_j$. So p_i must enter the critical section before p_j . \square

Theorem 7.8. *The algorithm provides **mutual exclusion** property.*

Proof. By contradiction, suppose that two processors are in the critical section simultaneously. Let p_i and p_j be the first pair of processors in the critical section together, with the ticket values k_i and k_j , respectively. Without loss of generality, let $k_i < k_j$, let p_i and p_j enter the critical section at time t_i and t_j , respectively. Using the same idea behind the proof of Theorem.7.7, for p_i to enter the critical section, the $first(v)$ must increment itself from 1 to k_i , which means that $k_i - 1$ processors have entered the Exit section and executed line 5. So at time t_i , $first(v)$ must have increased itself from 1 to k_i . Since we assumed that p_i and p_j be the first pair of processors in the critical section together, no other processor can enter the critical section between t_i and t_j , as a result, none of the processors will enter the exit section and execute the line 5 to increase the number of $first(v)$ between t_i and t_j . So at time t_j , the value of $first(v)$ will still be k_i . This contradicts the fact that p_j would have to read k_j in $first(v)$ before it can enter the critical section. Therefore, p_i and p_j cannot be in the critical section together, so mutual exclusion is satisfied. \square

Theorem 7.9. *The algorithm provides **no deadlock** property.*

Proof. By contradiction, suppose there exists a time t after which there is some processor in the entry section but no processor in the critical section. It follows that there exists an integer k such that before time t there have been exactly k entries into and exits from the critical section. Since the value of $first(v)$ is changed only in line 6 when processors exit from the critical section, and it is incremented by 1 at that time, the value of $first(v)$ has been incremented k times and is therefore $k+1$ at time t . Further, since no more entries and exits take place after time t , by assumption, the value

of $first(v)$ is $k + 1$ at all times after t . By Theorem.7.7, the processors with ticket from 1 through k should have entered and exited the critical section by time t . By lemma.7.6, there must have a processor with ticket $k + 1$, and it must be in the entry section based on our assumption. Therefore, when it next reads $first(v)$, it will read the $k + 1$ and subsequently enters the critical section. This is a contradiction proving that the algorithm satisfies no deadlock. \square

Theorem 7.10. *The algorithm provides no starvation property.*

Proof. By contradiction, suppose that some processor is starved from entering the critical section. Let p_i , with ticket k_i , be the processor with the lowest ticket that is starved, which means that p_i is in the entry section at time t_i but never enters the critical section. Since the processors with ticket smaller than k_i all enter and exit the critical section, let t' be the time when the last of these processors exit the critical section. By the Theorem.7.7, no processor with ticket larger than k_i can enter the critical section. Therefore, after time t' , there is no processor in the critical section. Further, after time of $max(t, t')$, there is a processor in the entry section, which is p_i , but no processor is in the critical section. This contradicts to the no deadlock property. \square

We have therefore proven **Mutual Exclusion, No Deadlock and No Starvation** for this algorithm.

Some Discussion: Notice that the ticket numbers in the above algorithm can increase arbitrarily, therefore requiring an unbounded register. We can take care of this by making the addition of 1 to $first(v)$ and $last(v)$ in the algorithm be *modulo* n , thus allowing the variable v to be bounded. This will make our proof of correctness more difficult.

7.4 Lower Bound on the Number of States in Shared Memory

The same as what we discussed for the lower bound in the old lectures, there exists a lower bound of n shared memory states for any algorithm which provides **mutual exclusion** and **no starvation** using read-modify-write variables.

Theorem 7.11. *Any algorithm that use **RMW** variables and provides mutual exclusion and no starvation, must have at least n least shared memory states (or $\lceil \log n \rceil$ bits to represent n shared memory states).*

Instead of proving this theorem, which is quite difficult, we prove a similar but weaker result. First, we define some notation to make some concepts more clear.

Definition 7.1. *An algorithm satisfies **k-bounded waiting** if for every valid schedule, there is no processor which remains in the **entry section** while some other processor enter **critical section** more than k times.*

To satisfy k -bounded waiting, a processor q can enter critical section at most k times while p is in the entry section waiting to enter critical section. On the other hand, the no starvation condition only requires that p must wait a finite amount of time in the entry section before it enters critical

section. So we can have an execution where p enters the entry section an infinite number of times, and the i th time p enters the entry section, it waits for q to enter and exit the critical section $i + 1$ times before it enters the critical section. Here, each waiting is finite, so the execution satisfies no starvation. However, for every value of k , the k -th entry of p into the critical section does not satisfy k -bounded waiting. Generally, no starvation and k -bounded waiting are conditions where each does not imply the other. k -bounded waiting by itself is a trivial condition because even an algorithm that violates no-deadlock trivially satisfies k -bounded waiting (if no processor enters the critical section, then it is never the case that one processor enters $k + 1$ times while another processor is waiting). On the other hand, **the k -bounded waiting condition along with No Deadlock is stronger than No Starvation**. Therefore, Theorem 7.12 is weaker than Theorem 7.11 since the lower bound here applies to only a subset of all starvation-free mutual exclusion algorithms.

Theorem 7.12. *Any algorithm that uses **RMW** variables and provides Mutual Exclusion and No Deadlock, and k -bound waiting must have at least n shared memory states.*

Proof. The basic idea for this proof is similar to what we have discussed for lower bound proof. Let C be a quiescent configuration, in which all processors are in remainder region. Assume the algorithm is **deadlock free** and **k -bounded**, which implies **no lockout**. By contradiction, we have less than n memory states. As shown in figure 7.4, based on the fact of no deadlock property, there exists a p_1 -only schedule τ_1 starting from initial configuration C and resulting in configuration C_1 , where p_1 is in the critical section. From C_1 , we have a p_2 -only schedule τ_2 that results in the configuration C_2 , where p_2 is in the entry section, note that the p_1 is still in the critical section in C_2 . Using the similar argument, we have p_i -only schedule of τ_i starting from C_{i-1} and ending in C_i , where p_i is in critical section as well as $p_2 \dots p_i$ are in the entry section.

By assumption, we have less than n shared memory states, by the **Pigeonhold Principle**, there are at least two configurations among C_1, \dots, C_n with the same shared memory state. Let C_i and C_j be these two configurations, without loss of generality, let $i < j$. Note that, for all l where $1 \leq l \leq i$, $C_i \stackrel{p_l}{\approx} C_j$, since p_l does not take any steps between p_i and p_j .

Apply a $\{p_1, \dots, p_i\}$ admissible schedule ρ' to C_i . The no deadlock property implies that some processor $p_l, 1 \leq l \leq i$ enters the critical section an infinite number of times in $exec(C_i, \rho')$.

Let ρ be some finite prefix of ρ' in which p_l enters the critical section $k + 1$ times. Since $C_i \stackrel{p_l}{\approx} C_j$ and ρ is $\{p_1, \dots, p_i\}$ only, the same $\{p_1, \dots, p_i\}$ -only schedule ρ can be applied to C_j , and each configuration in the execution from C_j to $F' = \rho(C_j)$ is similar to the corresponding configuration in the execution from C_i to $F = \rho(C_i)$ with respect to the processor p_1, \dots, p_i . Therefore, since p_l enters critical section $k + 1$ times in (C_i, ρ) , then p_l will enter the critical section $k + 1$ times in (C_j, ρ) as well. However, as stated earlier, processors p_{i+1}, \dots, p_j have been waiting in entry section in configuration C_j . Hence, in (C_j, ρ) , the processors p_{i+1}, \dots, p_j have been waiting in entry section while p_l enters critical section $k + 1$ times, which violates k -bounded waiting, a contradiction. \square

Note that nowhere in this proof do we use the fact that we are using **RMW** registers. This makes the proof a general proof that holds on other types of shared memory variables. If we are only allowed **read/write** registers, we can use this fact in the proof to get a stronger lower bound result.

