1 Mutual Exclusion Using More Powerful Primitives

The previously described algorithms required $O(n)$ variables to provide mutual exclusion. In this section, a new set of primitives will be described called “read-modify-write” primitives. The basic steps are:

1. Read the value of the variable
2. Based on the value read, modify $v$
3. Write a result

These are performed in a single atomic step. As a simple example, the increment function is a read-modify-write primitive.

1.1 A Test&Set Object

A Test&Set object is a special case of a read-modify-write primitive. There are two functions which can be performed on them:

Test&Set($v$):

```c
if (v = 0)
    then
        v := 1
        return(0)
    else
        return(1)
```

Reset($v$): $v := 0$

This primitive provides a small amount of coordination between processors. We can use one Test&Set variable to achieve Mutual Exclusion. The idea is:

1. When the value of the variable changes from 0 to 1, enter the critical section
2. When leaving the critical section, reset the value back to 0
1.2 Test&Set Algorithm

Entry:
1. If Test&Set(v) = 1
2. Then goto line 1
<CS>

Exit:
3. Reset(v)
REM

Theorem 1 Test&Set provides Mutual Exclusion.

Proof: Assume by contradiction that Mutual Exclusion is violated, and hence, more than one processor is in the critical section at the same time. Consider the first time that this occurs, and call this time \( t \). At time \( t \), \( P_i \) and \( P_j \) are in the critical section together. Without loss of generality, assume \( P_i \) enters first and hence is already in the critical section at some \( t' < t \).

At \( t' \), \( P_i \) reads 0 and writes 1 in \( v \) to enter the critical section. Similarly, at time \( t \), \( P_j \) reads 0 and writes 1 in \( v \). Hence, it follows that at some time \( t'' > t' \), and \( t'' < t \), the value of \( v \) was set to 0. Hence, some processor must have set the value to 0 at \( t'' \). But note that it could not be \( P_i \), since by assumption \( P_i \) entered the critical section and remained there until \( t \). (It could not have left the critical section, because then \( t' \) would not have been the entry time prior to \( t \)). But it could not have been \( P_j \), because \( t \) is the first time that two processors are in the critical section at the same time, so \( P_j \) could not have been in the critical section at time \( t'' \).

Therefore, we have arrived at a contradiction and it follows that Test&Set provides Mutual Exclusion.

\[ \square \]

Theorem 2 Test&Set provides the No Deadlock condition.

Proof: Assume by contradiction that beginning at time \( t \), no processor enters the critical section, but \( P_i \) is in the entry section. There are two cases to consider:

1. No processor has entered the critical section up to time \( t \).
2. A processor has entered the critical section, and then exited prior to time \( t \).

In both of these cases, \( v = 0 \) since \( v \) initializes to 0 and \( v \) is set to 0 in step 3 of the algorithm (exit). But, if \( v = 0 \), then by step 1 of the algorithm, \( P_i \) can enter the critical section - A contradiction.

\[ \square \]

Note that the No Starvation condition is **not** satisfied. Consider the following execution:
p0  
1. Test&Set(v)  
<CS>  
3. Reset(v)  
1. Test&Set(v)  
<CS>  
...  

Note that in the above execution, p0 repeatedly enters the critical section, while p1 starves.

### 1.3 Read-Modify-Write Objects

In general, a RMW object is called by:

\[
\text{temp} := \text{RMW}(v, f(v))
\]

This performs the follow steps:

1. Read the value of \( v \)
2. Modify the value of \( \text{temp} \) to be the value of \( v \)
3. Write the value of \( v \) to be \( f(\text{temp}) \)

Test&Set is a special case of this more general object, in which \( f(0) = f(1) = 1 \), and the reset function is a special case in which \( f(0) = f(1) = 0 \). To provide a stronger algorithm which also provides no starvation, we can use a RMW register which represents a FIFO queue. The idea is that a processor gets a token upon entry to the critical section, and checks if the token is equal to its own value to enter. Processors are enqueued when they enter ENTRY and enter the critical section when they reach the head of the queue. Processors are dequeued upon exiting the critical section.

Each processor \( P_i \) has two local variables \( pos_i \) and \( queue_i \). The shared memory variable is a RMW register \( v = (first, last) \)

- **first** - The ticket of the processor at the head of the queue
- **last** - The ticket of the next processor to enter the queue
1.4 RMW Algorithm for Processor $P_i$

Init: $v = (1, 1)$

Entry:
1. $pos_i = \text{RMW}(v, (\text{first}(v), \text{last}(v) + 1))$
2. $queue_i = \text{RMW}(v, v)$
3. if $\text{first}(queue_i) \neq \text{last}(pos_i)$
4. then goto line 2
5. $\text{RMW}(v, (\text{first}(v) + 1, \text{last}(v)))$

Note that in this algorithm, there are $n^2$ states for only one single variable. This algorithm also implements a strong fairness property: Processors enter the critical section in the same order they execute line 1.

**Lemma 3** The values of first($v$) and last($v$) are non-decreasing.

**Lemma 4** Two processors can never have the same ticket.

**Lemma 5** If there is a processor with ticket $L$, then there is a processor with ticket $L-1$ for all $L$.

**Theorem 6** If processor $P_i$ has ticket $k_i$ and processor $P_j$ has ticket $k_j$, where $k_i < k_j$, then $P_i$ will enter the critical section before $P_j$.

**Proof:** Assume by contradiction that processor $P_i$ has ticket $k_i$, processor $P_j$ has ticket $k_j$ where $k_i < k_j$, but $P_j$ enters the critical section before $P_i$. Note of course that last(pos$_i$) = $k_i$ and last(pos$_j$) = $k_j$. Consider the time $t_j$ at which $P_j$ enters the critical section with ticket $k_j$. To enter the critical section, $P_j$ had to check that first(queue$_j$) is equal to last(pos$_j$). So at time $t_j$, first(queue$_j$) = last(pos$_j$). Now consider the time $t_i$ at which $P_i$ enters the critical section. By assumption, $t_i > t_j$. To enter the critical section, $P_i$ had to check that first(queue$_i$) = last(pos$_i$). By Lemma 3, first(queue$_i$) $\geq$ first(queue$_j$) and last(pos$_i$) $\geq$ last(pos$_j$) since $t_i > t_j$ But this violates the assumption that $k_i < k_j$. A contradiction. $
$
**Theorem 7** This algorithm provides the No Starvation condition.

**Proof:** Assume by contradiction that there is a processor $P_i$ that is starved, and assume without loss of generality that it has the lowest ticket out of all starved processors. Call the ticket owned by $P_i$ ticket $k_i$. By Lemma 5, some processor $P_j$ has ticket $k_j = k_i - 1$. By assumption, $P_j$ has entered and exited the critical section. By Theorem 6, no processor with a ticket higher than $k_i$ will enter the critical section. Since first(queue$_j$) = last(pos$_j$) at the point where $P_j$ entered the critical section, it must be that first(v) = first(queue$_j$) + 1 after $P_j$ leaves the critical section, since $P_j$ increments first(v) when exiting. But then first(v) = last(pos$_j$) + 1 = $k_i$ at time $t$, and hence $P_i$ can enter the critical section, a contradiction.

Since we have arrived at a contradiction, this algorithm provides the No Deadlock condition.
Theorem 8 This algorithm provides Mutual Exclusion.

Proof: Assume by contradiction that Mutual Exclusion is violated, and hence, more than one processor is in the critical section at the same time. Consider the first time that this occurs, and call this time $t$. At time $t$, $P_i$ and $P_j$ are in the critical section together.

By Lemma 4, two processors can never have the same ticket. Hence, without loss of generality, assume $P_i$ has ticket $k_i$ and $P_j$ has ticket $k_j$ with $k_i < k_j$. Then $P_i$ enters the critical section before $P_j$ by Theorem 6. Let $t'$ be the time $P_i$ enters the critical section. Then at $t'$, $first(queue_i) = last(pos_i)$. It follows that $t$ is the time $P_j$ enters the critical section. Then at $t$, $first(queue_j) = last(pos_j)$ and since $last(pos_j) > last(pos_i)$, it must be that the value of $first(queue_j) > first(queue_i)$. Since $first(queue_j) > first(queue_i)$, it follows that $first(v)$ was incremented between time $t'$ and time $t$.

Then at some $t''$ where $t' < t'' < t$, some processor left the critical section to increment $first(v)$. It could not have been $P_i$ since $P_i$ enters at $t'$ and remains in the critical section at least until $t$. But it could not have been $P_j$ since $P_j$ does not enter until $t$. Since no other processor is in the critical section between $t'$ and $t$, and since $t$ is the first time when there is more than one processor in the critical section, we have arrived at a contradiction. It follows that this algorithm provides Mutual Exclusion.