1 Analysis of the LR Algorithm

Recall from before that we wish to illustrate the following theorem with respect to the LR Algorithm:

**Theorem 1** For \( n \geq 3 \), the LR algorithm satisfies \( T \xrightarrow{14l} \frac{1}{16} C \).

In order to do so, we prove a series of lemmas that taken together imply the above result. The lemmas are:

**Lemma 1**
1. If \( U \xrightarrow{t} U' \) and \( U' \xrightarrow{t'} U'' \), then \( U \xrightarrow{t+t'} U'' \)
2. If \( U \xrightarrow{t} U' \) then \( U \cup U'' \xrightarrow{t} U' \cup U'' \)

**Lemma 2** \( T \xrightarrow{3l} RT \cup C \)

**Lemma 3** \( RT \xrightarrow{3l} F \cup L \)

**Lemma 4** \( F \xrightarrow{2l} G \cup L \)

**Lemma 5** \( G \xrightarrow{4l} \frac{1}{4} L \)

**Lemma 6** \( L \xrightarrow{1} C \)

We will continue proving the previous lemmas.

**Proof:** [Lemma 3] If any processor is initially in \( F \cup L \), or enters \( L \) within \( 3l \) time, then we are done. So assume that this is not the case. Then, no processor enters the critical section within \( 3l \) time, so all processes are in \( R \cup T \) for time \( 3l \).

If any processor enters \( F \) within \( 3l \) time, then we are done. So again, assume this is not the case. In particular, this implies that no processor enters \( T \) from \( R \) within \( 3l \) time. Therefore,
all processors are initially in $S \cup D \cup R \cup W$ and no processor enters $F \cup L$ within $3l$ time. If any processor is initially in $S \cup D$, or reaches $S \cup D$ within time $l$, then within $3l$ time they reach $F \cup L$, a contradiction. So all processors must be in $R \cup W$ initially. But they can’t all be in $R$, since $RT \subset T$, so some processor must be in $W$.

It follows that no processor initially holds a fork, since they are all either in $R$ or $W$. As deduced earlier, no processor enters $T$ from $R$, so the processor in $W$ takes a step in time $l$ and obtains a fork, reaching $S$, which is a contradiction.

**Proof:** [Lemma 4] If any processor is initially in $L$, we are done, so assume this is not the case. Let $i$ be a particular processor in $F$. Then, there are three cases to consider:

1. $i - 1 \in \rightarrow \cup R \cup F$
2. $i + 1 \in \leftarrow \cup R \cup F$
3. $i - 1 \in \leftarrow, \ i + 1 \in \rightarrow$

In case 1, which is illustrated in Figure 1, with probability $\frac{1}{4}$, $i$’s next random choice is left, and $(i - 1)$’s next random choice is right. So assume this to be the case. Within time $l$, processor $i$ flips the coin and goes to a state in $\overrightarrow{W}$. There are two sub-cases to consider:

![Figure 1: Illustration of case 1.](image)

(a) In the meantime, processor $i - 1$ does not access the shared fork. Then we claim that processor $(i - 1)$’s state is still in $\rightarrow \cup R \cup F$. So the system configuration is in $G$. Hence, $i \in \overrightarrow{W}$ and $i - 1 \in \rightarrow \cup R \cup F$

(b) In the meantime, processor $i - 1$ does access the shared fork. Consider the first time it does so. The shared fork must be $(i - 1)$’s second fork by our assumption. Hence, $i - 1$ gets the second fork and goes to $L$, so the system configuration is in $L$.

Note that case 2 follows from symmetry. Consider case 3, illustrated in Figure 2, where $i - 1 \in \leftarrow, \ i + 1 \in \rightarrow$. Intuitively, this is the “bad” case. But note that we can say that somewhere in the ring of processors, $\exists j | j \in \rightarrow \cup R \cup F$, and $j + 1 \in \leftarrow$. If $j + 1 \in \overrightarrow{W} \cup \overrightarrow{S}$, then the system configuration is in $G$ and hence we are done. Otherwise, $j + 1 \in \overrightarrow{D}$ and $j \in \rightarrow \cup R \cup F$. Now, with probability $\frac{1}{4}$, $(j + 1)$’s next choice is left, and $j$’s next random choice is right. By the same argument used in case 1, we are done.

$\blacksquare$
Theorem 2 This algorithm provides Mutual Exclusion.

Proof: Assume by contradiction that Mutual Exclusion is violated, and hence, more than one processor that depend on the same fork are 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, and $P_i$ and $P_j$ each require fork $f_k$.

Without loss of generality, let $f_k$ be the fork to the right of $P_i$ and the left of $P_j$. Then it must be that $f(right)$ is true for $P_i$ at time $t$, and $f(left)$ is true for $P_j$ at time $t$. Assume, again without loss of generality, that $P_i$ entered the critical section at time $t' < t$. Then at time $t'$, $f(right)$ is true for $P_i$. But then at some time $t''$, where $t' < t'' < t$, $P_j$ checks the value of its $f(left)$ and it must have been false. So some other processor must have set this fork to be false. But $P_i$ and $P_j$ are the only processors with access to this fork $f_k$, and by assumption $P_i$ remained in the critical section until time $t$, a contradiction.

Hence, this algorithm provides mutual exclusion.

Note that with a worst case adversary, the no starvation property is not certain. But, we can ensure independent progress. That is, we can claim that some processor is able to make progress. Given that there are a set of processors in the trying region, then if there are contending processors in $R$, some processor can get into the critical section. The idea is that if I cannot take a fork, some other processor must have it, and the other processor will be eventually be able to enter. A condition cannot occur such that no processor is able to enter the critical section.

2 Shared Object Implementations

We have discussed some special cases of shared objects which help to coordinate distributed algorithms among processors. In general, these shared objects have some kind of input, some kind of output, and some kind of modification routine, like a function.

Figure 3 illustrates a high-level diagram of the asynchronous shared memory model. In this model, a set of processes interact with a set of objects. At the same time, users are interacting with the processes. In a sense, the users are really interacting with a single shared object which has some kind of read port, write port, modify routine, and acknowledgement. In this section, we discuss three types of shared memory objects, called registers.
2.1 Register Types

1. **Safe Register** - When a processor initiates a read from a safe register, the safe register will return the value of the last completed write, as long as it does not overlap the read. That is, a read from a safe register is only guaranteed to behave properly if it is not overlapping a write - If they overlap, the safe register can return garbage.

2. **Regular Register** - A regular register will return the value of either the last completed write, or an overlapping write. Note however that multiple reads overlapping a write need not return the same value, and hence two reads overlapping a write may appear out of order, with the earlier read returning the new value, and the later read returning the old value.

3. **Atomic Register** - An atomic register guarantees that the reads and writes, in theory, occur at a single point in time. That is, a read happening before a write will return the old value, and a read happening after the write will return the new value. This eliminates the problem of regular registers, in which two reads could return the values out of order - If the older read returns the new value, the newer one will as well.