1 Lamport’s Bakery Algorithm

Algorithm 1 Lamport’s Bakery Algorithm: Algorithm on Processor $p_i$

Code for Processor $P_i$

Entry Section
1 $CHOOSE[i] := 1$
2 $NUM[i] := \max(NUM[1], \ldots, NUM[n]) + 1$
3 $CHOOSE[i] := 0$
4 for $j := 1$ to $n$ do
5 begin
6 wait until $CHOOSE[j] = 0$
7 wait until $NUM[j] = 0$ or $(NUM[j], ID_j) \geq (NUM[i], ID_i)$
8 end
⟨Critical Section⟩

Exit Section
9 $NUM[i] := 0$
⟨Remainder section⟩

In the above algorithm lines 1-8 are entry section, while line 9 is the exit section.
We prove certain properties of this algorithm.

Lemma 1 If processor $P_i$ is in the critical section, then

$$\forall j : j \neq i : Num[j] = 0 \vee (NUM[j], ID_j) \geq (NUM[i], ID_i)$$

Proof: We need to argue that the blocking condition of step 7 continues to hold while $P_i$ is in CS. $Num[i]$ remains constant after $t_j$, where $t_j$ is the time-point where $P_i$ executes step 7 for the $j^{th}$ iteration of the loop. So now, is there any way that $P_j$ could change its NUM value to break the blocking condition? We will show that this is not possible as follows. Consider the possible NUM values of $P_j$ that $P_i$ could have read in the $j^{th}$ iteration of the loop.
• Case I: Lets say $P_i$ read $NUM[j] = 0$. Any changes to this value of $Num[j]$ can only happen in step 2 (after passing line 1). But since $P_i$ read $CHOOSE[j] = 0$ in line 6, $p_j$ sets $CHOOSE[j] = 1$ in line 1 after that. This means that when $P_i$ reads the $NUM$ array in step 2, $P_i$’s value $NUM[i]$ has already been set. So in step 2 the $NUM$ value chosen by $P_j$ has to be greater, which maintains the blocking condition.

• Case II: $p_i$ read that $(NUM[j], ID_j) \geq (NUM[i], ID_i)$ in line 6. Suppose $NUM[j]$ changes now. It can do thus only in the entry section or the exit section. If its modified in the exit section the blocking condition still holds since $Num[j]$ resets to 0. If $NUM[j]$ is modified in the entry section, by a similar argument as in Case I, $p_j$ reads $NUM[i]$ in line 2 after $p_i$ sets $NUM[i]$ in line 2. Therefore the new $NUM[j]$ is greater than $NUM[i]$. Thus the blocking condition still holds.

Therefore, the statement of the lemma holds in either case.

**Lemma 2** If $p_i$ is in the critical section, then $NUM[i] \neq 0$.

**Proof:** This is straightforward since $P_i$ has to choose a non-zero ticket $Num[i]$ in step 2 before entering the CS. And $Num[i]$ does not change while $P_i$ is in CS.

**Theorem 3** Lamport’s Bakery Algorithm satisfies Mutual Exclusion.

**Proof:** Suppose not, that is lets say $p_i$ and $p_j$ ($i \neq j$) are both in the critical section at the same time. We will now derive a contradiction.

By Lemma 2, $NUM[i] \neq 0$ and $NUM[j] \neq 0$ since they are both in CS.

By Lemma 1, 
$\text{(NUM}[j], ID_j) \geq (\text{NUM}[i], ID_i)$ as well as $(\text{NUM}[i], ID_i) \geq (\text{NUM}[j], ID_j)$

This implies that $(\text{NUM}[j], ID_j) = (\text{NUM}[i], ID_i)$ which is a contradiction since $P_i$ and $P_j$ are two different processors with different (unique) IDs.

**Theorem 4** Lamport’s Bakery Algorithm satisfies No Starvation.

**Proof:** Suppose not and pick a processor $P_i$ with the lowest ticket that is starved. Since $P_i$ can only starve by being stuck in the steps of the loop, its ticket is not going to change after it enters the loop. Since there are a finite number of processors, eventually all processors with lower tickets will enter and exit the CS, leading to a point after which all processors arriving in the entry section will now have tickets larger $P_i$. For processor $P_i$ to be starved, they will have to enter CS while $P_i$ does not, but this contradicts Lemma 1.

**Theorem 5** Lamport’s Bakery Algorithm satisfies No Deadlock.

**Proof:** No starvation—proven above—means that if a processor wishes to enter the CS, it will eventually succeed as long as no processor stays in the CS forever. No deadlock means that if one or more processors try to enter the CS, then one of them eventually succeeds as long as no processor stays in CS forever. Clearly, no starvation is a stronger property than no-deadlock.
2 Mutual exclusion with more powerful primitives

Requiring contention slows things down. Let us now consider a non-blocking protocol which relies on hardware support for stronger atomic primitives.

2.1 A Mutual Exclusion Algorithm Using Test&Set

The primitives, test&set and reset are defined as follows:

\[
\text{test&set}(v):
\begin{align*}
1. & \quad \text{if } v = 0 \\
2. & \quad \text{then } v := 1 \\
3. & \quad \text{return } 0 \\
4. & \quad \text{else return } 1
\end{align*}
\]

\[
\text{reset}(v): v := 0
\]

A mutual exclusion algorithm using test&set variables is given below.

Algorithm 2 Mutual Exclusion using Test&Set Register

Entry Section
1 if test&set\((v) = 1 \) //atomic step
2 then goto line 1
\(\langle\text{Critical Section}\rangle\)
Exit Section
3 reset\((v)\)
\(\langle\text{Remainder section}\rangle\)

Theorem 6 Test&Set algorithm provides mutual exclusion.

Proof: (Sketch) This can be shown by contradiction by considering the earliest time \(t\) when any two processors—call them \(P_i\) and \(P_j\)—were in CS together, i.e. both \(P_i\) and \(P_j\) have read \(v = 0\) in their executions of line 1 before this point \(t\). It can be argued using the definition of the test&set primitive that a reset is required between the time points of executions of line 1 by \(P_i\) and \(P_j\). This can then be shown to contradict the assumption that the time \(t\) was the earliest such time in which two processors were in CS simultaneously.

The algorithm does not provide no starvation. The following execution shows how processor \(p_1\) is just slow enough, such that it always reads \(v = 1\) and gets locked out.
\[ p_0 \quad \cdots \quad p_1 \]
\[
\begin{array}{l}
\text{read } v = 0 \\
\text{write } v = 1 \\
\{ \text{CS} \} \\
\text{write } v = 0 \\
\text{read } v = 0 \\
\text{write } v = 1 \\
\{ \text{CS} \}
end{array}
\]
\[ \text{read } v = 1 //\text{starved!} \]

\[ p_0 \quad \cdots \quad p_1 \]
\[
\begin{array}{l}
\text{read } v = 0 \\
\text{write } v = 1 \\
\{ \text{CS} \} \\
\text{write } v = 0 \\
\text{read } v = 0 \\
\text{write } v = 1 \\
\{ \text{CS} \}
end{array}
\]
\[ \text{read } v = 1 //\text{starved!} \]

### 2.2 Mutual Exclusion using Read-Modify-Write (RMW) Registers

The *Read-Modify-Write* object allows a 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 described as \( \text{temp} := \text{RMW}(v, f(v)) \), which translates to the atomic sequence:

1. \( \text{temp} := v \)
2. \( v := f(\text{temp}) \)

The *test&set* register is the weakest case of RMW, where \( f(0) = 1 \) and \( f(1) = 1 \).

Algorithm 3 uses a single shared RMW object to implement a FIFO queue which serializes the processors access to the CS thus providing Mutual Exclusion with No Starvation.\(^1\)

---

**Algorithm 3 Mutual Exclusion Using a RMW Register**

- **Initially** \( v = (1,1) \)
- **Code for Processor \( P_i \)**

**Entry Section**

1. \( \text{position}_i := \text{RMW}(v, (\text{first}(v), \text{last}(v) + 1)) \);
2. \( \text{queue}_i := \text{RMW}(v, v) \);
3. if \( \text{first}(\text{queue}_i) \neq \text{last}(\text{position}_i) \);
   4. then goto 2;

(Critical Section)

**Exit Section**

5. \( \text{RMW}(v, (\text{first}(v) + 1, \text{last}(v))) \);

(remainder section)

---

\(^1\)Actually it provides “fairness” which is stronger than No Starvation, as shown next.
**Definition 1** *Fairness Condition requires that processors enter the critical section in the same order as their requests for CS.*

The fairness condition is stronger than No Starvation. Algorithm 3 provides the fairness condition; in this case, the order of CS requests is the same order in which they execute line 1 of Algorithm 3.

### 2.2.1 Lower Bound on the Number of States in Shared Memory

The number of required shared memory states, rather than the number of actual RMW variables, are important since any number of states can be encoded in just 1 RMW variable. So the number of required shared memory states gives a better measure for the complexity of the algorithm.

**Theorem 7** *Any algorithm that uses RMW variables and provides Mutual Exclusion and No Starvation, must have at least* $n$ *shared memory states (or* $\lceil \log n \rceil$ *bits to represent* $n$ *shared memory states)*

Instead of proving the above theorem, we will sketch the proof of a similar but weaker result.

**Definition 2** *In a k-bounded waiting algorithm, a particular processor will wait in the ES only as long as some other processor enters the CS at most k times.*

$k$-bounded waiting is quite trivial in absence of No Deadlock. But together with No Deadlock, $k$-bounded implies No Starvation.

The weaker result we prove is stated as follows:

**Theorem 8** *Any algorithm that uses RMW variables and provides Mutual Exclusion and No Deadlock, and k-bounded waiting must have at least n shared memory states.*

**Proof:** (Semi-formal) The key idea here is that processors that do not take steps between one configuration and another configuration reachable from the previous one—such that both have identical shared memory states—cannot distinguish between the two configurations. We will construct an execution which exploits this similarity of two identical shared memory state configurations, (one reachable from the other) to processors that are inactive between the two configurations, so as to derive a contradiction to $k$-bounded waiting.

By the pigeonhole principle, if we have less than $n$ shared memory states and $n$ different configurations, then at least two configurations will have an identical shared memory state.

Referring to Figure 1, $C_0$ is a quiescent configuration, in which all processors are in remainder region. Each $\tau_k$ is a schedule in which only $P_k$ takes steps. Clearly the configurations $C_0$ to $C_n$ can form a chain of reachable configurations. By our assumption of less than $n$ shared memory
Figure 1: Configuration chain for lower bound of $n$ states
states and the pigeonhole principle there are at least two configurations in this chain, with identical shared memory states. Let these two configurations be $C_i$ and $C_j$. By our definitions of $\tau_k$, only processors $P_{i+1},...P_j$ take steps within $C_i$ and $C_j$. Thus for all other processors, configurations $C_i$ and $C_j$ are indistinguishable, written as $C_i \sim P_{i+1},...P_j P_{j+1},...P_n C_j$.

Due to No Deadlock, $P_1$ can be in CS at the end of $\tau_1$. Each of the other $\tau_k$ schedules (i.e. $k \neq 1$) result in $P_k$ being “added to” the Entry Section (ES), while the processors $P_{k+1},...P_n$ remain in the Remainder Section (REM). Thus the main difference between $C_i$ and $C_j$ is that in $C_j$ there are additional processors $P_{i+1},...P_j$ in ES. However as we will show, $P_1$ is unable to “see” this difference and can therefore violate $k$-bounded waiting.

Referring to Figure 1, starting at $C_i$, $\alpha_1$ is a $P_1$-only schedule where $P_1$ exits CS and enters REM. The resulting configuration is $D$, in which processors $P_2,...,P_i$ are in ES and all the other processors are in REM. Now, starting at $D$, $\alpha_2$ is a schedule in which only $P_2,...,P_i$ take steps, so as to enter the CS and exit to REM. Clearly $E = \alpha_2(D)$ is a quiescent configuration since everybody is in REM. Starting from $E$, $\alpha_3$ is a $P_1$-only schedule where $P_1$ cycles through ES, CS, REM $k+1$ times. (Note that $\alpha_3$ is possible due to No Deadlock)

Because $C_i \sim P_{i+1},...P_i C_j$, the schedules $\alpha_1, \alpha_2$ and $\alpha_3$—in which only $P_1,...,P_i$ take steps—can be applied in succession starting from $C_j$ with the same result of $P_1$ cycling through ES, CS, REM $k+1$ times. But now it is a violation of $k$-bounded waiting since $P_{i+1},...P_j$ were waiting in ES while $P_1$ cycled through CS more than $k$ times. This is a contradiction. ■