Deconstructing the Bakery to Build a Distributed State Machine

Leslie Lamport

Expanded Version 7 January 2022

Abstract

We connect two well-known concurrent algorithms, the bakery algorithm and a distributed state-machine algorithm, by a sequence of three mutual exclusion algorithms. Each algorithm is derived from the preceding one.

Contents

1 Introduction

The reader and I, the author, will journey between two concurrent algorithms of the 1970s that are still studied today. We will start at the bakery algorithm [9] and end at an algorithm for implementing a distributed state machine [12]. I hope we will enjoy the voyage and perhaps even learn something.

The bakery algorithm ensures that processes execute a critical section of code one at a time. A process trying to execute that code chooses a number it believes to be higher than the numbers chosen by other such processes. The process with the lowest number goes first, with ties broken by process name. In the distributed state-machine algorithm of [12], each process maintains a logical clock, the clocks being synchronized by having a process include its clock value in the messages it sends. Commands to the state machine are ordered according to the value of a process's clock when it issues a command, with ties broken by process name.

The similarity of the bakery algorithm's numbers and the state-machine algorithm's clocks has been noticed, but I know of no previous rigorous connection between them. Our trip provides this connection, going from the bakery algorithm to the state-machine algorithm through a sequence of algorithms, each (except the first) derived from the preceding one.

The first algorithm on the journey is a straightforward generalization of the bakery algorithm, mainly by allowing a process to read other processes' numbers in an arbitrary order. We then deconstruct this algorithm by having each process maintain multiple copies of its number, one for each other process. Next is a distributed version of the deconstructed algorithm obtained by having each copy of a process i's number kept by the process that reads it, where i writes the value stored at another process by sending a message to that process. We then modify this distributed algorithm to ensure that numbers increase with each execution of the critical section. Finally, we arrive at the distributed state-machine algorithm by forgetting about critical sections and just using the numbers as logical clocks.

Not only do our algorithms date from the 1970s, but the path between them is one that could have been followed at that time. The large amount of related work done since then has neither influenced nor obviated any part of the route. At the end of our journey, a concluding section discusses that related work and why the algorithms that begin and end our path are still studied today. The correctness proofs in our journey are informal, much as they would have been in the 1970s. More modern rigorous proofs are discussed in the concluding section.

2 The Original Bakery Algorithm

The bakery algorithm solves the mutual exclusion problem, introduced and solved by Edsger Dijkstra [3]. The problem assumes a set of processes that alternate between executing a noncritical and a critical section of code. A process must eventually exit the critical section, but it may stay forever in the noncritical section. The basic requirement is that at most one process can be executing the critical section at any time. A solution to the mutual exclusion problem lies at the heart of almost all multiprocess programming.

The bakery algorithm assumes processes are named by numbers from 1 through N. Figure 1 contains the code for process number i, almost exactly as it appeared in the original paper. The values of the variables *number* and *choosing* are arrays indexed by process number, with $number[i]$ and $choosing[i]$ initially equal to 0 for every process i. The relation $\ll$ is lexicographical ordering on pairs of numbers, so $(1,3) \ll (2,2) \ll (2,4)$; it is an irreflexive total ordering on the set of all pairs of integers.

Mutual exclusion can be achieved very simply by not allowing any process ever to enter the critical section. A mutual exclusion algorithm needs to satisfy some progress condition as well. The condition Dijkstra's algorithm satisfies is *deadlock freedom*, meaning that if one or more processes try to enter the critical section, one of them must succeed. Most later algorithms satisfy the stronger requirement of *starvation freedom*, meaning that every process that tries to enter the critical section eventually does so. Before discussing mutual exclusion, we show that the bakery algorithm is starvation free. But first, some terminology.

We say that a process is *in the doorway* when it is executing statement M. After it finishes executing M until it exits its critical section, we say that it is *inside the bakery.* When it's at any other place in its code, we say that it is *outside the bakery.*

We first show that the algorithm is deadlock free. If it weren't, it would eventually reach a state in which every process is either forever in its noncritical section or forever inside the bakery. Eventually, $choosing[i]$ would equal 0 for all i, so every process inside the bakery would be waiting forever at statement $L3$. But this is impossible because the waiting process i with the smallest value of $(number[i], i)$ would eventually enter the critical section. Hence, the algorithm is deadlock free.

To show that the algorithm is starvation free, it suffices to obtain a contradiction by assuming that a process i remains forever inside the bakery and outside the critical section. By deadlock freedom, other processes must continually enter and leave the critical section, since they cannot halt there.

```
begin integer $k$ ;
$L1$: *noncritical section* ;
    $choosing[i]$ := 1 ;
$M$: $number[i]$ := $1 + maximum(number[1], \ldots, number[N])$ ;
    $choosing[i]$ := 0 ;
    for $k = 1$ step 1 until $N$ do
      begin
        $L2$: if $choosing[k] \neq 0$ then goto $L2$ ;
        $L3$: if $number[k] \neq 0$ and $(number[k], k) \ll (number[i], i)$
               then goto $L3$ ;
      end ;
    *critical section* ;
    $number[i]$ := 0 ;
    goto $L1$
end
```

Figure 1: Process i of the original bakery algorithm.

However, once a process j is outside the bakery, to enter the bakery again it must execute statement M and set *number*$[j]$ to be greater than *number*$[i]$. At that point, process j must remain forever inside the bakery because it will loop forever if it reaches $L3$ with $k = i$. Eventually, *number*$[i]$ will be less than *number*$[j]$ for every process j in the bakery, so i will enter its critical section. This is the contradiction that proves starvation freedom.

Essentially the same proof shows that the other mutual exclusion algorithms we derive from the bakery algorithm also satisfy starvation freedom. So, we will say little more about starvation freedom. We now explain why the bakery algorithm satisfies mutual exclusion. For brevity, we abbreviate $(number[i], i) \ll (number[j], j)$ as $i \ll j$.

Here is a naive proof that i and j cannot both be in their critical section at the same time. For i to enter the critical section, it must find $number[j] = 0$ or $i \ll j$ when executing $L3$ for $k = j$. Similarly, for j to enter the critical section, it must find $number[i] = 0$ or $j \ll i$ when executing $L3$ for $k = i$. Since a process's number is non-zero when it executes $L3$, this means that for i and j both to be in their critical sections, $i \ll j$ and $j \ll i$ must be true, which is impossible.

This argument is flawed because it assumes that both i and j were inside the bakery when the other process executed $L3$ for the appropriate value of k. Suppose process i read *number*$[j]$ while j was in the doorway (executing M) but had not yet set *number*$[j]$. It is possible for j to have read $number[i] = 0$ in $L3$ and entered the critical section, and for i then to have chosen $number[i]$ to make $i \ll j$ and also entered the critical section.

The flaw in the argument is corrected by statement $L2$. Since $choosing[j]$ equals 1 when j is in the doorway, process i executed $L3$ after $L2$ found that j was not in the doorway; and similarly, j executed $L3$ after finding i not in the doorway. If, in both cases, the two processes were inside the bakery when $L2$ was executed, then the naive argument is correct. If one of them, say j, was not inside the bakery, it must have been outside the bakery. Since i was then inside the bakery, with its current value of $number[i]$, process j must have chosen $number[j]$ to be greater than the current value of $number[i]$, making $i \ll j$ true. Hence, j could not have exited the $L3$ loop for $k = i$ and entered the critical section while i was still in the bakery. Therefore i and j cannot both be in the critical section.

Observe that the *choosing* variable serves only to ensure that, when process i executes $L3$ for $k = j$, there had been an instant when i was already inside the bakery and j was not in the doorway. This will be important later.

The most surprising property of the bakery algorithm is that it does not require reading or writing a memory register to be an atomic action. Carefully examining the proof of mutual exclusion shows that it just requires that $number[i]$ and $choosing[i]$ are what were later called safe registers [13], ensuring only that a read not overlapping a write obtains the current value of the register. A read that does overlap a write can obtain any value the register might contain.

It is most convenient to describe a safe register in terms of atomic actions. We represent writing a value v to the register as two actions: the first sets its value to a special constant **¿** and the second sets it to v. We represent a read as a single atomic action that obtains the value of the register if that value does not equal **¿**. A read of $number[i]$ when it equals **¿** can return any natural number, and a read of $choosing[i]$ when it equals **¿** can return 0 or 1.

3 Generalization of the Original Algorithm

Two generalizations of the bakery algorithm were obvious when it was published. The first is that, in statement M, it is not necessary to set $number[i]$ to $1 + maximum(\ldots)$. It could be set to any number greater than that maximum. (It can also be set to the maximum if that makes $(number[j], j) \ll (number[i], i)$ for all j, but we won't bother with that generalization.) We rewrite statement M using :> to mean "is assigned a value greater than".

The second obvious generalization is that statements $L2$ and $L3$ for different values of k do not have to be executed in the order specified by the **for** statement. Since the proof of mutual exclusion considers each pair of processes by themselves, the only requirement is that, for any value of k, statement $L2$ must be executed before $L3$. For different values of k, those statements can be executed concurrently by different subprocesses. Also, there is no reason to execute them for $k = i$ because their **if** tests always equal false.

These two generalizations have appeared elsewhere [5, 10]. There is another, less obvious generalization that seems to be new: The assignment of 0 to $number[i]$ after the process leaves the critical section need not be completed before the process enters the noncritical section. In fact, that assignment need not even be completed if the process leaves the noncritical section to enter its critical section again. As long as that assignment is completed or aborted (leaving the register equal to ¿) before $number[i]$ is assigned a new value in statement M, it just appears to other processes as if process i is still in the critical section or is executing the assignment statement immediately after the critical section. Therefore, mutual exclusion is still satisfied. To maintain starvation freedom, the write of 0 must eventually be completed if i remains forever in the noncritical section. There seems to be no simple way to describe in pseudo-code these requirements for setting $number[i]$ to 0 upon completing the critical section. We simply add the mysterious keyword **asynchronously** and refer to this discussion for its explanation.

The generalized algorithm is in Figure 2. Processes are explicitly declared, the outer **process** statement indicates that there are processes numbered from 1 through N and shows the code for process number i. Variables are declared with their initial values. The inner **process** statement declares that process i has $N - 1$ subprocesses j with numbers from 1 through N, with none numbered i, and gives the code for subprocess j. That statement is executed by forking the subprocesses and continuing to the next statement (the critical section) when all subprocesses have terminated. Harmful or not, **goto**s have been eliminated. The outer loop is described as a **while** statement. The loops at $L2$ and $L3$ have been described with **await** statements, each of which repeatedly evaluates its predicate and terminates when it is true. The :> in statement M and the **asynchronously** statement are explained above.

```
process $i$ in $\{1, \ldots, N\}$
  variables $number[i] = 0,\ choosing[i] = 0$ ;
  while true do
      noncritical section ;
      $choosing[i] := 1$ ;
 $M$: $number[i] :> maximum(number[1],\ \ldots,\ number[N])$ ;
      $choosing[i] := 0$ ;
      process $j \neq i$ in $\{1, \ldots, N\}$
        $L2$: await $choosing[j] = 0$ ;
        $L3$: await $(number[j] = 0) \vee ((number[i],\ i) \ll (number[j],\ j))$
      end process ;
      critical section ;
      asynchronously $number[i] := 0$ ;    see explanation in text
  end while
end process
```

Figure 2: A generalization of the original bakery algorithm.

4 The Deconstructed Bakery Algorithm

We have assumed that $number[i]$ and $choosing[i]$ are safe registers, written only by i and read by multiple readers. Such a register is easily implemented with safe registers having a single reader by keeping a copy of the register's value in a separate register for each reader.

We deconstruct the generalized bakery algorithm by implementing the safe registers $choosing[i]$ and $number[i]$ with single-reader registers $localCh[j][i]$ and $localNum[j][i]$, for each $j \neq i$. Note the counterintuitive order of the subscripts, with $localCh[j][i]$ and $localNum[j][i]$ containing the copies of $choosing[i]$ and $number[i]$ read by process j.

The pseudo-code of the deconstructed algorithm is in Figure 3. The reads of $choosing[j]$ and $number[j]$ by process i in the generalized algorithm are replaced by reads of $localCh[i][j]$ and $localNum[i][j]$. The variable $number[i]$ is now read only by process i, and we have eliminated $choosing[i]$ because process i never reads it. *Ad hoc* notation is used in statement M to indicate that $number[i]$ is set to be greater than the values of all $localNum[j][i]$.

We have explicitly indicated the two atomic actions that represent writing a value v to the safe register $localNum[j][i]$, first setting its value to ¿ and then to v. We have not bothered doing that for the writes to $localCh[j][i]$. The writes of $localCh[j][i]$ and $localNum[j][i]$ are performed by subprocesses of process i, except that the $N - 1$ separate writes of ¿ to all the registers

```
process $i$ in $\{1, \ldots, N\}$
  variables $number[i] = 0$, $localNum[*][i] = 0$, $localCh[*][i] = 0$ ;
  while true do
     noncritical section ;
     process $j \neq i$ in $\{1, \ldots, N\}$
         $localCh[j][i] := 1$
     end process ;
 M: $number[i] :=$ any $v > 0$ with $\forall j \neq i : v > localNum[i][j]$ ;
     $localNum[*][i] :=$ ¿ ;
     process $j \neq i$ in $\{1, \ldots, N\}$
           $localNum[j][i] := number[i]$ ;
           $localCh[j][i] := 0$ ;
       L2: await $localCh[i][j] = 0$ ;
       L3: await $(localNum[i][j] = 0) \vee ((number[i], i) \ll (localNum[i][j], j))$
     end process ;
     critical section ;
     $number[i] := 0$ ;
     $localNum[*][i] :=$ ¿ ;
     asynchronously process $j \neq i$ in $\{1, \ldots, N\}$   see explanation in text
                           $localNum[j][i] := 0$
                        end process
  end while
end process
```

Figure 3: The deconstructed bakery algorithm.

$\mathit{localNum}[j][i]$ are represented by an assignment statement

$$\mathit{localNum}[*][i] \;:= \text{¿}$$

of the main process i. (This will be more convenient for our next version of the bakery algorithm.) To set $\mathit{number}[i]$ to 0 after i exits the critical section, all the registers $\mathit{localNum}[j][i]$ are set to ¿ by the main process, and each is set to 0 by a separate process. We require that the setting of $\mathit{localNum}[j][i]$ to 0 has been either completed or aborted when $\mathit{localNum}[j][i]$ is set to $\mathit{number}[i]$ by subprocess (i, j). Again, this is not made explicit in the pseudo-code.

A proof of correctness for the deconstructed algorithm can be obtained by simple modifications to the proof for the original algorithm. For the original algorithm, we defined process i to be in the doorway while executing statement M, which ended with assigning the value of $\mathit{number}[i]$. Since $\mathit{number}[i]$ has been replaced by the registers $\mathit{localNum}[j][i]$, process i now has a separate doorway for each other process j. We say that i is in the doorway *with respect to* (wrt) j from when it begins executing statement M until its subprocess j assigns $\mathit{number}[i]$ to $\mathit{localNum}[j][i]$. We say that i is inside the bakery wrt j from when it leaves the doorway wrt j until it exits the critical section. The definition of i outside the bakery is the same as before.

To transform the proof of correctness of the original bakery algorithm to a proof of correctness of the deconstructed algorithm, we replace every statement that i or j is in the doorway or inside the bakery with the statement that it is there wrt the other process. The modified proof shows that the function of statement $L2$ is to ensure that, sometime between i coming inside the bakery wrt j and executing $L3$ for j, process j was not in the doorway wrt i.

5 The Distributed Bakery Algorithm

We now implement the deconstructed bakery algorithm with a distributed algorithm. Each main process i is executed at a separate node, which we call node i, in a network of processes that communicate by message passing. The variable $\mathit{localNum}[j][i]$, which is process j's copy of $\mathit{number}[i]$, is kept at node j. It is set by process i to the value v by sending the message v to j. The setting of $\mathit{localNum}[j][i]$ to ¿ in the deconstructed bakery algorithm is implemented by the action of sending that message, and $\mathit{localNum}[j][i]$ is set to v by process j when it receives the message. Thus, we are implementing the deconstructed algorithm by having process j obtain a previous value of *localNum*$[j][i]$ on a read when *localNum*$[j][i]$ equals **¿**. Since the deconstructed algorithm allows such a read to obtain any value, this is a correct implementation.

For now, we assume that process i can write the value of *localCh*$[j][i]$ atomically by a magical action at a distance. We will remove this magic later.

We assume that messages sent from a process i to any other process j are received in the order that they are sent. We represent the messages in transit from i to j by a FIFO (first-in, first-out) queue $q[i][j]$. We let ϕ be the empty queue, and we define the following operations on a queue Q.

Append(Q, val) Appends the element *val* to the end of Q.

Head(Q) The value at the beginning of Q.

Behead(Q) Removes the element at the beginning of Q.

Head(Q) and *Behead*(Q) are undefined if Q equals ϕ.

The complete algorithm is in Figure 4. The shading highlights uses of *localCh*, whose magical properties need to be dealt with. Along with the main process i, there are concurrently executed processes (i, j) at node i, for each $j \neq i$. Process (i, j) receives and acts upon the messages sent to i by j.

The main process i of the distributed algorithm is obtained directly from the deconstructed algorithm by replacing the assignments of **¿** to each *localNum*$[j][i]$ with the sending of a message to j, except for two changes. The first is that statement M and the following sending of messages to other processes (represented by appending *number*$[i]$ to all the message queues $q[i][j]$) have been made a single atomic action. We can do this because we can view the end of each message queue $q[i][j]$, onto which messages are appended, to be part of process i's local state. A folk theorem [4] says that, for reasoning about a multiprocess algorithm, we can combine any number of actions that access only a process's local state into a single atomic action. That folk theorem has been formalized in a number of results starting with one by Lipton [15], perhaps the most directly applicable being [14]. In our algorithm, making this action appear atomic just requires preventing other processes at node i from acting on any incoming messages while the action is being executed.

The other significant change to the deconstructed algorithm is that the **asynchronously** statement has disappeared. The setting of *localNum*$[j][i]$ is performed by the receipt of messages sent by i to j. FIFO message delivery ensures that it is set to 0 before its subsequent setting to a non-zero value.

```
process $i$ in $\{1, \ldots, N\}$
variables $number[i] = 0$, $localNum[*][i] = 0$, $localCh[*][i] = 0$,
          $ackRcvd[i][*] = 0$, $q[*][i] = \phi$ ;
  while true do
     noncritical section ;
     process $j \neq i$ in $\{1, \ldots, N\}$
       $localCh[j][i] := 1$
     end process ;
     atomic $M$: $number[i]$ := any $v > 0$ with $\forall j \neq i : v > localNum[i][j]$ ;
                 $Append(q[i][*], number[i])$
     end atomic ;
     process $j \neq i$ in $\{1, \ldots, N\}$
       $L0$ : await $ackRcvd[i][j] = 1$ ;
             $localCh[j][i] := 0$ ;
       $L2$: await $localCh[i][j] = 0$ ;
       $L3$: await $(localNum[i][j] = 0) \vee (number[i], i) \ll (localNum[i][j], j)$
     end process ;
     critical section ;
     $ackRcvd[i][*] := 0$ ;
     $number[i] := 0$ ;
     $Append(q[i][*], 0)$
  end while
end process

process $i, j \neq i$ in $\{1, \ldots, N\}$
  while true do
    atomic await $q[j][i] \neq \phi$ ;
             if $Head(q[j][i]) = ack$
               then $ackRcvd[i][j] := 1$
               else $localNum[i][j] := Head(q[j][i])$ ;
                    if $Head(q[j][i]) \neq 0$ then $Append(q[j][i], ack)$
                    end if
             end if ;
             $Behead(q[j][i])$
    end atomic ;
  end while
end process
```

Figure 4: The Distributed Bakery Algorithm, with magic.

Also, since *localNum*$[j][i]$ is now set by process (j, i) upon receipt of the message, the assignment to it in subprocess j of i has been removed.

Correctness of the deconstructed algorithm also depends on the assignment to *localNum*$[j][i]$ being performed before process i sets *localCh*$[j][i]$ to 0. Since the assignment to *localNum*$[j][i]$ is now performed at node j, the ordering of those two operations is no longer trivially implied by the code. To maintain that ordering, subprocess j of i must learn that process (j, i) has set *localNum*$[j][i]$ to *number*$[i]$ before it can set *localCh*$[j][i]$ to 0. This is done by having (j, i) send a message to i with some value *ack* that is not a natural number. Process (j, i) sets the value of *localNum*$[j][i]$ and sends the *ack* message to i as a single atomic action. When process (i, j) at node i receives the *ack* message, it sets *ackRcvd*$[i][j]$ to 1 to notify subprocess j of process i that the *ack* has arrived. The setting of *localNum*$[j][i]$ to *number*$[i]$ in the deconstructed algorithm is replaced by statement $L0$ that waits for *ackRcvd*$[i][j]$ to equal 1.

The rest of the code for the main process i is the same as that of the corresponding process of the deconstructed algorithm, except that after i leaves the critical section, the asynchronous setting of all the registers *localNum*$[j][i]$ to 0 is replaced by sending the message 0 to all the processes j; and *ackRcvd*$[i][j]$ is reset to 0 for all j.

The asynchronously executed process (i, j) receives messages sent by j via $q[j][i]$. For an *ack* message, it sets *ackRcvd*$[i][j]$ to 1; for a message with a value of *number*$[j]$ it sets *localNum*$[i][j]$ and, if the value is non-zero, sends an *ack* to j.

The one remaining problem is the magical atomic reading and writing of the register *localCh*$[i][j]$. The value of that register is used only in statement $L2$. The purpose of $L2$ is to ensure that, before the execution of $L3$, there existed a time T when i was in the bakery wrt j and j was not in the doorway wrt i. We now show that statement $L2$ is unnecessary, because executing $L0$ ensures the existence of such a time T.

The execution of statement M by j and the sending of *number*$[j]$ in a message to i are part of a single atomic action, and j enters the bakery wrt i when that message is received at node i. Therefore, j is in the doorway wrt i exactly when there is a message with a non-zero integer in $q[j][i]$. Let's call that message a *doorway* message. Process i enters the bakery wrt j when its message containing *number*$[i]$ is received at node j, an action that appends to $q[j][i]$ the *ack* that $L0$ is waiting to arrive. If there is no doorway message in $q[j][i]$ at that time, then immediately after execution of that action is the time T whose existence we need to show, since it occurred before the receipt of the *ack* that $L0$ was waiting for. If there is a doorway message in $q[j][i]$, then the required time T is right after that message was received at node i. Because of FIFO message delivery, that time was also before the receipt of the *ack* that $L0$ is waiting for. In both cases, executing $L0$ ensures that there was some time T after i entered inside the bakery wrt j when j was not in the doorway wrt i. Hence, statement $L2$ is redundant.

Because $L2$ is the only place where the value of $localCh[i][j]$ is read, we can eliminate *localCh* and all statements that set it. Removing all the grayed statements in Figure 4 gives us the distributed bakery algorithm, with no magic.

The first paper devoted to distributed mutual exclusion was apparently that of Ricart and Agrawala [19]. Their algorithm can be viewed as an optimization and simplification of our algorithm. It delays the sending of *ack* messages in such a way that a process can enter its critical section when it receives an *ack* from every other process, so it doesn't have to keep track of other processes' numbers. The number 0 messages sent upon exiting the critical section can therefore be eliminated, yielding an algorithm with fewer messages. Although nicer than our algorithm, the Ricart-Agrawala algorithm is not directly on the path we are traveling.

6 A Distributed State Machine

In a distributed state machine [12], there is a set of processes at separate nodes in a network, each wanting to execute state machine commands. The processes must agree on the order in which all the commands are executed. To execute a command, a process must know the entire sequence of preceding commands.

A distributed mutual exclusion algorithm can be used to implement a distributed state machine by having a process execute a single command in the critical section. The order in which processes enter the critical section determines the ordering of the commands. It's easy to devise a protocol that has a process in its critical section send its current command to all other processes, which order it after all preceding commands. Starting with this idea and the distributed bakery algorithm, we will obtain the distributed state-machine algorithm of [12] by eliminating the critical section.

The bakery algorithm is based on the idea that if two processes are trying to enter the critical section at about the same time, then the process i with the smaller value of $(number[i], i)$ enters first. We now make that true no matter when the two processes enter the critical section. Define a version of the bakery algorithm to be *number*-ordered if it satisfies this condition: If process i enters the critical section with $\mathit{number}[i] = n_i$ and process j later enters the critical section with $\mathit{number}[j] = n_j$, then $(n_i, i) \ll (n_j, j)$. We now make the distributed bakery *number*-ordered. We can do that because we have generalized the bakery algorithm to set $\mathit{number}[i]$ to *any* number greater than the maximum value of the values of $\mathit{number}[j]$ it reads, not just to the next larger number.

We add to the distributed bakery algorithm a variable *maxNum*, where $\mathit{maxNum}[i][j]$ is the largest value $\mathit{localNum}[i][j]$ has equaled, for $j \neq i$. We let $\mathit{maxNum}[i][i]$ be the largest value $\mathit{number}[i]$ has equaled. We then make two changes to the algorithm. First, we replace statement M with:

$$\begin{array}{ll} M: \mathit{number}[i] & :> \ \mathit{maximum}(\mathit{maxNum}[i][1], \ldots, \mathit{maxNum}[i][N]); \\ \quad \mathit{maxNum}[i][i] & := \ \mathit{number}[i] \end{array}$$

Second, in process (i, j), if $\mathit{localNum}[i][j]$ is assigned a non-zero value, then $\mathit{maxNum}[i][j]$ is assigned that same value. (The FIFO ordering of messages assures that the new value of $\mathit{maxNum}[i][j]$ will be greater than its previous value.) Clearly, $\mathit{localNum}[i][j]$ always equals $\mathit{maxNum}[i][j]$ or 0. The value of $\mathit{number}[i]$ chosen this way is therefore allowed by statement M of the distributed algorithm, so this is a correct implementation of that algorithm. We now show that it is *number*-ordered.

Suppose i enters the critical section with $\mathit{number}[i] = n_i$ and j later enters the critical section with $\mathit{number}[j] = n_j$. It's easy to see that $(n_i, i) \ll (n_j, j)$ if $i = j$, so we can assume $i \neq j$. The proof of mutual exclusion for the deconstructed algorithm shows that either: (i) $(n_i, i) \ll (n_j, j)$ or (ii) j chose n_j after reading a value of $\mathit{localNum}[i][j]$ written after i set it to n_i. In our modified version of the distributed algorithm, j reads $\mathit{maxNum}[j][i]$ rather than $\mathit{localNum}[i][j]$ to set $\mathit{number}[j]$, and $\mathit{maxNum}[j][i]$ never decreases. Therefore, $(n_i, i) \ll (n_j, j)$ is true also in case (ii), so the algorithm is *number*-ordered.

Since the algorithm is *number*-ordered, we don't need the critical section to implement a distributed state machine. We can order the commands by the value $(\mathit{number}[i], i)$ would have had when i entered the critical section to execute the command. Process i can send the command it's executing in the messages containing the value of $\mathit{number}[i]$ that it sends to other processes. In fact, we don't need $\mathit{number}[i]$ at all. When we send that message, $\mathit{number}[i]$ has the same value as $\mathit{maxNum}[i][i]$. We can eliminate everything in the main process i except the atomic statement containing statement M, which can now be written as follows, where *Cmd* is process i's current command:

```
**atomic**
  $M$: $maxNum[i][i] :> maximum(maxNum[i][1], \ldots, maxNum[i][N])$;
      $Append((maxNum[i][i], Cmd), q[i][*])$
**end atomic**
```

There is one remaining problem. Process i saves the messages containing commands that it sends and receives, accumulating a set of triples (v, j, Cmd) indicating that process j issued a command Cmd with $number[j]$ having the value v. It knows that those commands are ordered by (v, j). But to execute the command in (v, j, Cmd), it has to know that it has received all commands (w, k, Dmd) with $(w, k) \ll (v, j)$. Process i knows that, for each process k, it has received all commands (w, k, Dmd) with $w \leq maxNum[i][k]$. However, suppose i has received no commands from k. How can i be sure that k hasn't sent a command in a message that i hasn't yet received? The answer is to use the distributed bakery algorithm's *ack* messages. Here's how.

For convenience, we let process i keep $maxNum[i][i]$ always equal to the maximum of the values $maxNum[i][j]$ (including $j = i$). It does this by increasing $maxNum[i][i]$, if necessary, when receiving a message with the value of $maxNum[i][j]$ from another process j. Upon receiving a message (v, Cmd) from process j, process i sets $maxNum[i][j]$ to v (possibly increasing $maxNum[i][i]$) and sends back to j the message $(maxNum[i][i], ack)$. Upon receiving that message, j sets $maxNum[j][i]$ accordingly, (increasing $maxNum[j][j]$ if necessary). When i has received all the *ack* messages for a command it issued with $maxNum[i][i]$ equal to v, all its values of $maxNum[i][j]$ will be $\geq v$, so process i knows it has received all commands ordered before its current command. It can therefore execute all of them, in the appropriate order, and then execute its current command.

This algorithm is almost identical to the distributed state machine algorithm in [12], where $maxNum[i][i]$ is called process i's clock. (The sketch of the algorithm given there is not detailed enough to mention the other registers $maxNum[i][j]$.) The one difference is that, when process i receives a message from j with a new value v of $maxNum[i][j]$, the algorithm of [12] requires $maxNum[i][i]$ to be set to a value $> v$, whereas $\geq v$ suffices. The algorithm remains correct if the value of $maxNum[i][i]$ is increased by any amount at any time. Thus, the registers $maxNum[i][i]$ could be logical clocks that are used for other purposes as well.

We have described all the pieces of a distributed state-machine algorithm, but have not put them together into pseudo-code. To quote [12]: "The precise algorithm is straightforward, and we will not bother to describe it."

7 Ancient and Recent History

In addition to being the author of this paper, I am the author of the starting and ending algorithms of our journey. The bakery algorithm is among hundreds of algorithms that implement mutual exclusion using only read and write operations to shared memory [22]. A number of them improve the bakery algorithm, the most significant improvement being a bound on the chosen numbers [6, 21]. But all improvements seem to add impediments to our path, except for one: Moses and Patin [17] optimized the bakery algorithm by allowing process i to stop waiting for process j at statement $L3$ if it reads two different values of $number[j]$. However, it is irrelevant to our path because it optimizes a case that cannot occur in the distributed bakery algorithm.

Mutual exclusion algorithms based on read and write operations have been of no practical use for decades, since modern computers provide special instructions to implement mutual exclusion more efficiently. Now, they are studied mainly as concurrent programming exercises. The bakery algorithm is of interest because it was the first mutual exclusion algorithm not to assume lower-level mutual exclusion, which is implied by atomic reads and writes of shared memory.

The distributed state-machine algorithm is interesting because it preserves causality. But it too is less important than the problem it solves. The most important contribution of [12] was the observation that any desired form of cooperation in a network of computers can be obtained by implementing a distributed state machine. The obvious next step was to make the implementation fault tolerant. The work addressing that problem is too extensive to discuss here. Fault-tolerant state-machine algorithms have become the standard building block for implementing reliable distributed systems [20].

There was no direct connection between the creation of the bakery algorithm and of the state-machine algorithm. The bakery algorithm was inspired by a bakery in the neighborhood where I grew up. A machine dispensed numbers to its customers that determined the order in which they were served. The state-machine algorithm was inspired by an algorithm of Paul Johnson and Robert Thomas [7]. They used the $\ll$ relation and process identifiers to break ties, but I don't know if that was inspired by the bakery algorithm.

The path between the two algorithms that we followed is not the one I originally took. That path began when I was looking for an example of a distributed algorithm for notes I was writing. Stephan Merz suggested the mutual exclusion algorithm I had used in [12] to illustrate the state-machine algorithm. I found it to be too complicated, so I decided to simplify it. (I did not remember the Ricart-Agrawala algorithm and was only later reminded of it by a referee.) After stripping away things that were not needed for that particular state machine, I arrived at the distributed bakery algorithm. It was obviously related to the original bakery algorithm, but it was still not clear exactly how.

I wanted to make the distributed algorithm an implementation of the bakery algorithm. I started with the generalization of having subprocesses of each process interact independently with the other processes; that was essentially how I had been describing the bakery algorithm for years. Delaying the setting of $number[i]$ to 0 was required because the distributed algorithm's message that accomplished it could be arbitrarily delayed. It took me a while to realize that I should deconstruct the multi-reader register $number[i]$ into multiple single-reader registers, and that both the original bakery algorithm and the distributed algorithm implemented that deconstructed algorithm.

The path back from the distributed bakery algorithm to the distributed state-machine algorithm was easy. It may have helped that I had previously used the idea of modifying the bakery algorithm to make values of $number[i]$ keep increasing. Paradoxically, that was done to keep those values from getting too large [10].

Correctness of a concurrent algorithm is expressed with two classes of properties: *safety* properties like mutual exclusion that assert what the algorithm may do, and *liveness* properties like starvation freedom that assert what the algorithm must do [1]. Safety properties depend on the actions the algorithm can perform; liveness properties depend as well on assumptions, often implicit, about what actions the algorithm must perform.

The kind of informal behavioral reasoning I have used here is notoriously unreliable. I believe the best rigorous proofs of safety properties are usually based on invariants—predicates that are true of every state of every possible execution [2]. Invariance proofs that the bakery algorithm satisfies mutual exclusion have often been used to illustrate formalisms or tools [5, 11]. An informal sketch of such a proof for the decomposed bakery algorithm is in the appendix. Elegant rigorous proofs of progress properties can be written using temporal logic [18].

Rigorous proofs are longer than informal ones and can intimidate readers not used to them. I almost never write one until I believe that what I want to prove is true. For the correctness of our algorithms, that belief was based on the reasoning embodied in the informal proofs I presented—the same kind of reasoning I used when I discovered the bakery and distributed state-machine algorithms.

I understood the two algorithms well enough to be confident in the correctness of the non-distributed versions of the bakery algorithm and of the derivation of the state-machine algorithm from the distributed bakery algorithm. Model checking convinced me of the correctness of the distributed bakery algorithm and confirmed the confidence my informal invariance proof had given me that the deconstructed algorithm satisfies mutual exclusion.

More recently, Stephan Merz wrote a formal, machine-checked version of my informal invariance proof. He also wrote a machine-checked proof that the actions of the distributed bakery algorithm implement the actions of the deconstructed bakery algorithm under a suitable data refinement. These two proofs show that the deconstructed algorithm satisfies mutual exclusion. The proofs are available on the Web [16].

References

[1] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[2] E. A. Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10:110–135, February 1975.

[3] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[4] David Harel. On folk theorems. *Commun. ACM*, 23(7):379–389, 1980.

[5] Wim H. Hesselink. Mechanical verification of Lamport's bakery algorithm. *Sci. Comput. Program.*, 78(9):1622–1638, 2013.

[6] Prasad Jayanti, King Tan, Gregory Friedland, and Amir Katz. Bounding Lamport's bakery algorithm. In Leszek Pacholski and Peter Ruzicka, editors, *SOFSEM 2001: 28th Conference on Current Trends in Theory and Practice of Informatics*, volume 2234 of *Lecture Notes in Computer Science*, pages 261–270. Springer, 2001.

[7] P. R. Johnson and R. H. Thomas. The maintenance of duplicate data bases. Request for Comment RFC #677, NIC #31507, ARPANET Network Working Group, January 1975.

[8] Leslie Lamport. Supplemental material for Deconstructing the Bakery to Build a Distributed State Machine. Web page. http://lamport.azurewebsites.net/pubs/bakery/deconstruction.html.

[9] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[10] Leslie Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[11] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977.

[12] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[13] Leslie Lamport. On interprocess communication. *Distributed Computing*, 1:77–101, 1986.

[14] Leslie Lamport. A theorem on atomicity in distributed algorithms. *Distributed Computing*, 4(2):59–68, 1990.

[15] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, December 1975.

[16] Stephan Merz. TLA+ specifications and proofs for "deconstructing the bakery to build a distributed state machine". Web page. https://members.loria.fr/SMerz/papers/distributed-bakery.html.

[17] Yoram Moses and Katia Patkin. Mutual exclusion as a matter of priority. *Theor. Comput. Sci.*, 751:46–60, 2018.

[18] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on the Foundations of Computer Science*, pages 46–57. IEEE, November 1977.

[19] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1):9–17, 1981.

[20] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[21] Gadi Taubenfeld. The black-white bakery algorithm and related bounded-space, adaptive, local-spinning and FIFO algorithms. In Rachid Guerraoui, editor, *Distributed Computing, 18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004, Proceedings*, volume 3274 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2004.

[22] Gadi Taubenfeld. Concurrent programming, mutual exclusion. In Ming-Yang Kao, editor, *Encyclopedia of Algorithms—2016 Edition*, pages 421–425. Springer, 2016.

Appendix: Proof of Mutual Exclusion

We now prove that the deconstructed bakery algorithm satisfies mutual exclusion. As we explained, our other mutual exclusion algorithms implement this algorithm, so they also satisfy mutual exclusion.

The proof is based on a formalism in which an execution of the algorithm is represented by a sequence of states. The transition from one state to the next, which we call an *event*, represents the execution of a single atomic action of the algorithm. We call the state before the event the *old* state and the state after it the *new* state. For simplicity, we assume that in executing statement M, process i reads $\mathit{localNum}[i][j]$ once for each process j. Because of how we represent safe registers, that read is a single event.

We first define several state predicates—formulas that are true or false of a state of an execution. The significant events of the algorithm are ones that can change the value of one or more of these predicates. Precise definitions of these predicates would require precisely specifying all the atomic actions of the algorithm. We describe these state predicates informally. We also state their properties that are used in the proof, which indicate how they should be defined for a precise definition of the algorithm. In all of these descriptions, i and j are assumed to be distinct processes.

$\mathit{inBakery}(i, j)$ True when i is in the bakery wrt j. This means that subprocess j of i has finished writing the value $\mathit{number}[i]$ computed in statement M to $\mathit{localNum}[j][i]$. Property: $\mathit{inBakery}(i, j)$ implies that $\mathit{localNum}[j][i]$ equals a number (not ¿) that is greater than 0.

$\mathit{inCS}(i)$ True when i has reached the critical section and has not finished executing it. Property: $\mathit{inCS}(i)$ implies $\mathit{inBakery}(i, j)$ for all j.

$\mathit{inDoorway}(i, j, v)$ Becomes true when i reads $\mathit{localNum}[i][j]$ in executing statement M and obtains the number v, and it becomes false when $\mathit{inBakery}(i, j)$ becomes true. Properties: implies that $\mathit{localCh}[j][i] = 1$, that statement M will make $\mathit{number}[i] > v$ true, and that $\mathit{inBakery}(i, j)$ will become true when $\mathit{inDoorway}(i, j, v)$ becomes false.

$\mathit{inDoorway}(i, j)$ Asserts that $\mathit{inDoorway}(i, j, v)$ is true for some v. (This is not quite the same as the definition of i being in the doorway wrt j in Section 4.)

$\mathit{Outside}(i, j)$ True when neither $\mathit{inBakery}(i)$ nor $\mathit{inDoorWay}(i, j)$ is true. Properties: implies that $\mathit{number}[i]$ eventually equals 0 and remains equal to 0 unless $\mathit{Outside}(i, j)$ becomes false (which will never happen if process i remains forever in the critical section). Exactly one of $\mathit{Outside}(i, j)$, $\mathit{inDoorway}(i, j)$, and $\mathit{inBakery}(i)$ is always true.

$\mathit{passed}(i, j, L)$ where L is statement $L2$ or $L3$. It becomes true when i reads a value of $\mathit{localCh}[i][j]$ (for $L2$) or $\mathit{localNum}[i][j]$ (for $L3$) that causes i to exit statement L. It becomes false when i exits the critical section (making $\mathit{inBakery}(i, j)$ false). Properties: In the state in which it becomes true, L's **await** condition is true; $\mathit{inCS}(i)$ implies $\mathit{passed}(i, j, L3)$.

Mutual exclusion asserts that, for any distinct processes i and j, the state predicate $\neg(\mathit{inCS}(i) \wedge \mathit{inCS}(j))$ is an invariant of the algorithm, meaning that it is true in every state of every possible execution. We prove that a state predicate I is an invariant by induction on the number of events that occur before reaching the state. This means proving:

I1. I is true in the initial state of any execution.

I2. For every event, if I is true in the old state then it is true in the new state.

I2 is the induction step of the proof. Proving something by induction often requires proving something stronger that satisfies the induction step. Proving that a state predicate P is an invariant often requires finding a state predicate I that implies P and satisfies I1 and I2.

To prove $\neg(\mathit{inCS}(i) \wedge \mathit{inCS}(j))$ is an invariant, we need an invariant I satisfying I1 and I2 that implies it. A key part of defining I is defining a state predicate $\mathit{Before}(i, j)$ that implies j cannot enter the critical section until i has finished executing the critical section. Here is its definition, which assigns the names 1, 2, 2a, 2b, and 2c to subformulas. The symbol $\neg$ is logical negation; $\wedge$ and $\vee$ are logical *and* and *or*, respectively. Recall that $i \ll j$ is an abbreviation for $(\mathit{number}[i], i) \ll (\mathit{number}[j], j)$.

$$\begin{array}{lll}
\mathit{Before}(i, j) \triangleq & & \\
 & & 1.\ \mathit{inBakery}(i, j) \\
 & \wedge & 2.\ (\quad a.\ \mathit{Outside}(j, i) \\
 & & \qquad \vee\ b.\ \mathit{inDoorway}(j, i, \mathit{number}[i]) \\
 & & \qquad \vee\ c.\ \mathit{inBakery}(j, i)\ \wedge\ (i \ll j)\ \wedge\ \neg \mathit{passed}(j, i, L3)\ )
\end{array}$$

We let $\mathit{Before}(i, j).1$ denote subformula 1 of $\mathit{Before}(i, j)$, and we name other subformulas similarly.

Understanding this definition is a first step to understanding the correctness proof. The basic idea behind the algorithm is that mutual exclusion is guaranteed because if i and j are competing to enter the critical section, then i will enter first iff $i \ll j$. So $Before(i,j)$ has to imply that j can't have or be able to choose $number[j]$ to make $j \ll i$ true. If i has not yet set $localNum[j][i]$ after step M, then it will be possible for j to read 0 as its value and choose a value for $number[j]$ that will make $j \ll i$. Hence, $Before(i,j).1$ is needed.

$Before(i,j).2$ states what must be true about process j to keep it from making $j \ll i$ true when $inBakery(i,j)$ is true. It is the disjunction of three subformulas that apply to the three possibilities of $Outside(j,i)$, $inDoorway(j,i)$, or $inBakery(j,i)$ being true. Considering these possibilities in the reverse order explains the three subformulas.

2c If $inBakery(j,i)$ is true, then j has chosen $number[j]$, so $i \ll j$ must be true. But that won't keep j from entering the critical section if $passed(j,i,L3)$ is true, so it must be false.

2b If $inDoorway(j,i)$ is true, then j has read $number[i]$ in executing statement M. Subformula 2b asserts that the value it read is the current value of $number[i]$, ensuring that j will choose $number[j]$ greater than $number[i]$ and thus make $i \ll j$ true.

2a If $Outside(j,i)$ is true, then j has not yet read $number[i]$. It can't enter the critical section before i because, to do so, it must first read the current value of $number[j]$ in statement M, which makes it choose $number[j]$ to make $i \ll j$ true.

The meaning of $Before(i,j)$ implies that once it becomes true, it should remain true until i exits its critical section, which makes $inBakery(i,j)$ false. Therefore, this should be true:

Lemma 1 For any distinct processes i and j, any event of the algorithm in which $Before(i,j)$ is true in the old state either leaves $Before(i,j)$ true or makes $inBakery(i,j)$ false in the new state.

PROOF: By definition of *Before*, it suffices to assume that an event E with $Before(i,j)$ true in the old state leaves $inBakery(i,j)$ true, and prove that the event also leaves $Before(i,j).2$ true. This requires showing that if one of the subformulas 2a, 2b, or 2c is true in the old state, then one of them is true in the new state. Here is the proof for the three cases in which the indicated one of these subformulas is true in the old state.

2a The definition of $\mathit{Outside}(j,i)$ implies $\mathit{Before}(i,j).2a$ remains true unless E is the event of process j reading $\mathit{localNum}[j][i]$ in statement M. Because $\mathit{inBakery}(i,j)$ is true, $\mathit{localNum}[j][i] \neq$ ¿, so that read obtains the value $\mathit{number}[i]$. Therefore, $\mathit{Before}(i,j).2b$ is true in the new state.

2b $\mathit{Before}(i,j).2b$ can become false in two ways: (i) $\mathit{inDoorway}(j,i)$ becomes false or (ii) the current value of $\mathit{number}[i]$ changes. Case (ii) is impossible because $\mathit{number}[i]$ does not change while $\mathit{inBakery}(i,j)$ remains true. In case (i), the definition of *inDoorway* implies that E must be the process j event that completes the execution of statement M, and condition 2b holding in the old state and the definition of $\ll$ imply that E makes $i \ll j$ true in the new state. The definitions of *inBakery* and *passed* imply that the other two conjuncts of 2c are true in the new state because $\mathit{passed}(i,j,L2)$ and $\mathit{passed}(i,j,L3)$ are false.

2c Only an event in process i or j can falsify $i \ll j$. Since the action leaves $\mathit{inBakery}(i,j)$ true and 2c is true in the old state, an action of neither i nor j can make $i \ll j$ false. An event with $\neg\mathit{passed}(j,i,L3)$ true in the old state cannot make $\mathit{inBakery}(j,i)$ false. Therefore, the only way 2c can be made false is by subprocess j making $\neg\mathit{passed}(j,i,L3)$ false by executing $L3$. But since $\mathit{inBakery}(i,j)$ implies $\mathit{localNum}[j][i]$ equals $\mathit{number}[i]$, which is a number greater than 0, the condition $i \ll j$ implies that such an execution of $L3$ by j is impossible.

This completes the proof of the lemma.

We can now define the invariant I that satisfies I1 and I2 and implies mutual exclusion. We first define $\mathit{Inv}(i,j)$, numbering and naming subformulas as for $\mathit{Before}(i,j)$:

$$\begin{aligned}
&\mathit{Inv}(i,j) \;\triangleq\\
&\qquad 1.\ \mathit{inBakery}(i,j) \Rightarrow (\mathit{Before}(i,j) \lor \mathit{Before}(j,i) \lor \mathit{inDoorway}(j,i))\\
&\quad\land\ \ 2.\ \mathit{passed}(i,j,L2) \Rightarrow (\mathit{Before}(i,j) \lor \mathit{Before}(j,i))\\
&\quad\land\ \ 3.\ \mathit{passed}(i,j,L3) \Rightarrow \mathit{Before}(i,j)
\end{aligned}$$

Here is the definition of I, where the quantification is over all distinct processes i and j:

$$I \;\triangleq\; \forall\, i,j \,:\, \mathit{Inv}(i,j)$$

The proof of the following lemma is hierarchically structured, where the proof of a step can be a sequence of substeps ending with a Q.E.D. substep whose proof proves the step. A step of the form

SUFFICES ASSUME: P

PROVE: Q

asserts that to prove the current goal, it suffices to assume P and prove Q. It makes Q the current goal and allows us to assume P in its proof. Omitting the PROVE clause is equivalent to letting Q be the current goal. The step CASE: P asserts that the current goal is true if P is true.

Lemma 2 *I is an invariant of the algorithm.*

1. SUFFICES ASSUME: i and j are distinct processes and E is an event of process i whose old state satisfies $Inv(i,j) \wedge Inv(j,i)$.

   PROVE: $Inv(i,j) \wedge Inv(j,i)$ is true in the new state.

   PROOF: The conjunction of invariants is an invariant, so it suffices to prove that $Inv(i,j) \wedge Inv(j,i)$ is an invariant for distinct processes i and j, which we do by proving conditions I1 and I2. Condition I1 is trivially true because $inBakery(p,q)$ is false in the initial state for $q \neq p$, which implies that $passed(p,q,L2)$ and $passed(p,q,L3)$ are also false. To prove I2, it suffices to prove this step's goal for an arbitrary event E. We can assume E is an event of process i or j, because those are the only processes whose actions can falsify $Inv(i,j) \wedge Inv(j,i)$. By symmetry, we need only prove it for E an event of i.

2. $Inv(i,j)$ is true in the new state.

   2.1. CASE: $inBakery(i,j)$ is true in the old state.

   2.1.1. SUFFICES ASSUME: $inBakery(i,j)$ is true in the new state.

   PROOF: $Inv(i,j)$ is trivially true if $inBakery(i,j)$ is false, which implies $passed(i,j,L2)$ and $passed(i,j,L3)$ are false.

   2.1.2. E does not falsify $Before(i,j)$ or $Before(j,i)$.

   PROOF: By Lemma 1 and the step 2.1.1 assumption, since E is an event of process i.

   2.1.3. $Inv(i,j).1$ is true in the new state.

   PROOF: $inBakery(i,j)$ is true in the old and new states by case assumption 2.1 and assumption 2.1.1. Whichever of the three disjuncts of $Inv(i,j).1$ is true in the old state must remain true in the new state for the following reasons:

$Before(i,j)$	By step 2.1.2.
$Before(j,i)$	By step 2.1.2.
$inDoorway(j,i)$	Its value can't be changed by an action of process i.

2.1.4. $Inv(i,j).2$ is true in the new state.

PROOF: By step 2.1.2, $Inv(i,j).2$ can be falsified only by the value of $passed(i,j,L2)$ changing from false to true, which implies that the event must be an execution of $L2$ that reads $localCh[i][j]$ and obtains the value 0. But this implies that $inDoorway(j,i)$ is false in the old state, so the truth of $Inv(i,j).1$ implies that $Before(i,j) \vee Before(j,i)$ is true in the old state, which by 2.1.2 implies that it, and therefore $Inv(i,j).2$ are true in the new state.

2.1.5. $Inv(i,j).3$ is true in the new state.

PROOF: The same reasoning as in the proof of step 2.1.4 shows that this subformula can be falsified only if the action is an execution of statement $L3$ that reads a value v of $localNum[i][j]$ that equals 0 or for which $(number[i], i) \ll (v, j)$ is true. It suffices to show that this implies $Before(j,i)$ is false, since then the truth of $Inv(i,j).2$ implies the truth of $Inv(i,j).3$.

We prove $Before(j,i)$ is false by assuming it to be true and obtaining a contradiction. By definition of $Before$, this assumption implies $inBakery(j,i)$, which implies that $localNum[i][j]$ does not equal ¿ and that the value v read when executing the action equals $number[j]$, which is not equal to 0. Therefore, $i \ll j$ must be true. However, $Before(j,i)$ and the step 2.1 case assumption imply $j \ll i$. This is the required contradiction because $j \ll i$ and $i \ll j$ cannot both be true.

2.1.6. Q.E.D.

PROOF: Steps 2.1.3, 2.1.4, and 2.1.5 imply that $Inv(i,j)$ is true in the new state.

2.2. CASE: $inBakery(i,j)$ is false.

2.2.1. SUFFICES ASSUME: $inBakery(i,j)$ is true in the new state.
PROVE: $Inv(i,j).1$ is true in the new state.

PROOF: Case assumption 2.2 implies that $Inv(i,j)$ can be falsified only if E is an action that makes $inBakery(i,j)$ true, so we can assume $inBakery(i,j)$ is true in the new state. To make $inBakery(i,j)$ true, action E must be the event in which i sets $localNum[j][i]$ to the value of $number[i]$ computed in statement M, which implies that $Passed(i,j,L2)$ and $Passed(i,j,L3)$ are false in the new state. Thus, $Inv(i,j).2$ and $Inv(i,j).3$ are trivially true in the new state, so we need only show that $Inv(i,j).1$ is also true then.

2.2.2. CASE: $Outside(j,i)$ is true in the new state.

PROOF: In this case, the definition of Before implies $\mathit{Before}(i, j)$ is true in the new state, making $\mathit{Inv}(i, j).1$ true.

2.2.3. CASE: $\mathit{inDoorway}(j, i)$ is true in the new state.

PROOF: This trivially implies that $\mathit{Inv}(i, j).1$ is true.

2.2.4. CASE: $\mathit{inBakery}(j, i)$ is true in the new state.

2.2.4.1. Either $i \ll j$ or $j \ll i$ is true in the new state.

The step 2.2.1 assumption and the step 2.2.4 case assumption assert that $\mathit{inBakery}(i, j)$ and $\mathit{inBakery}(j, i)$ are both true. This implies that both $\mathit{number}[i]$ and $\mathit{number}[j]$ do not equal ¿ or 0. Since $\ll$ is a total order on pairs of integers, either $i \ll j$ or $j \ll i$ is true in the new state.

2.2.4.2. CASE: $i \ll j$

PROOF: We do a case split on the value of $\mathit{passed}(j, i, L3)$. If it is false, then $i \ll j$, the step 2.2.4 case assumption, the step 2.2.1 assumption, and the definition of Before imply that $\mathit{Before}(i, j)$ is true, so subformula $\mathit{Inv}(i, j).1$ is true in the new state.

If $\mathit{passed}(j, i, L3)$ is true, then the truth of $\mathit{Inv}(j, i)$ in the old state implies $\mathit{Before}(j, i)$ is true in the old state, which by Lemma 1 implies it is true in the new state, because $\mathit{inBakery}(i, j)$ is true. By definition of $\mathit{Before}(j, i)$, this implies $j \ll i$, which is impossible by the case assumption since $\ll$ is an irreflexive order.

2.2.4.3. CASE: $j \ll i$

PROOF: Since the event makes $\mathit{inBakery}(i, j)$ true (by steps 2.2 and 2.2.1), it leaves $\neg\mathit{passed}(i, j, L3)$ true. The case assumption and the definition of $\mathit{Before}(j, i)$ therefore imply that $\mathit{Before}(j, i)$ is true in the new state, so $\mathit{Inv}(i, j).1$ is true in that state.

2.2.4.4. Q.E.D.

PROOF: By steps 2.2.4.1, 2.2.4.2, and 2.2.4.3.

2.2.5. Q.E.D.

PROOF: By steps 2.2.2, 2.2.3, and 2.2.4, since either $\mathit{Outside}(j, i)$, $\mathit{inDoorway}(j, i)$, or $\mathit{inBakery}(j, i)$ must be true.

2.3. Q.E.D.

PROOF: By 2.1 and 2.2

3. $\mathit{Inv}(j, i)$ is true in the new state.

3.1. SUFFICES ASSUME: $\mathit{inBakery}(j, i)$ is true in the new state.

PROOF: $\mathit{Inv}(j, i)$ is trivially true if $\mathit{inBakery}(j, i)$ is false.

3.2. SUFFICES ASSUME: Event E makes $\mathit{Before}(i, j)$ or $\mathit{inDoorway}(i, j)$ false.
PROVE: $\mathit{Before}(i, j) \vee \mathit{Before}(j, i)$ is true in the new state.

PROOF: We need to show that the i event does not falsify any of the three subformulas of $\mathit{Inv}(j, i)$. An event of i obviously cannot change $\mathit{passed}(j, i, L2)$ or $\mathit{passed}(j, i, L3)$, and by Lemma 1 it cannot make $\mathit{Before}(j, i)$ false. Therefore, the only way the event can make any of the three subformulas false is by making $\mathit{Inv}(j, i).1$ or $\mathit{Inv}(j, i).2$ false by making $\mathit{inDoorway}(i, j)$ or $\mathit{Before}(i, j)$ false. This won't make $\mathit{Inv}(j, i).1$ or $\mathit{Inv}(j, i).2$ false if $\mathit{Before}(i, j) \vee \mathit{Before}(j, i)$ is true in the new state.

3.3. CASE: The event makes $\mathit{Before}(i, j)$ false.

PROOF: By Lemma 1, an i event that makes $\mathit{Before}(i, j)$ false makes $\mathit{inBakery}(i, j)$ false, which makes $\mathit{Outside}(i, j)$ true. This in turn makes $\mathit{Before}(j, i)$ true, which of course makes $\mathit{Before}(i, j) \vee \mathit{Before}(j, i)$ true in the new state.

3.4. CASE: The event makes $\mathit{inDoorway}(i, j)$ false.

PROOF: The case assumption implies that $\mathit{inBakery}(i, j)$ is true in the new state. By step 2, $\mathit{Inv}(i, j).1$ is true in that state. By step 3.1, $\mathit{inBakery}(j, i)$ is true so $\mathit{inDoorway}(j, i)$ is false. Therefore, the truth of $\mathit{Inv}(i, j).1$ in the new state implies that $\mathit{Before}(i, j) \vee \mathit{Before}(j, i)$ is true in the new state.

3.5. Q.E.D.

PROOF: By steps 3.2, 3.3, and 3.4.

4. Q.E.D.

PROOF: Steps 2 and 3 prove the goal introduced by step 1.

With these two lemmas, the proof of mutual exclusion is simple.

Theorem The algorithm satisfies mutual exclusion.

1. SUFFICES ASSUME: $\mathit{Inv}(i, j) \wedge \mathit{Inv}(j, i) \wedge \mathit{inCS}(i) \wedge \mathit{inCS}(j)$
PROVE: $\neg \mathit{inCS}(j)$

PROOF: By Lemma 2, the definitions of I and mutual exclusion, and propositional logic.

2. $\mathit{Before}(i, j)$

PROOF: By the step 1 assumption and the definition of $Inv(i, j)$, since $inCS(i)$ implies $passed(i, j, L3)$.

3. $\neg passed(j, i, L3)$
   PROOF: By the step 1 assumption, step 2, and the definition of *Before*, since $inCS(j)$ implies $inBakery(j, i)$.

4. Q.E.D.
   PROOF: By step 3, since $\neg passed(j, i, L3)$ implies $\neg inCS(j)$.

Our definitions and the two lemmas can also be used to make the proof of starvation freedom more rigorous. Doing that is a straightforward matter of making the statements of the informal proof more precise.