

Chapter 9

Bakery Algorithm

9.1 Introduction

The bakery algorithm is one of the simplest known solutions to the N -process mutual exclusion problem on a platform that provides read-write atomicity only. The basic idea is that each non-thinking process has a variable that indicates the position of that process in a hypothetical queue of all the non-thinking processes. Each process in this queue scans the variables of the other processes, and enters the critical section only upon determining that it is at the head of the queue.

But the resulting algorithm is still not easy to understand. So below we first look at a simplified version of the bakery algorithm, one that uses a coarser grain of atomicity, namely, atomic read-modify-write (where the read is over N processes), and prove that it works. Then we look at the original version and prove that it works.

We use assertional reasoning to establish the key properties of the system, but we do not give all the mundane details needed for an assertional proof. In particular, we make assertions about the evolution of the hypothetical queue, and show that these assertions satisfy the proof rules assuming some trivial properties which are mentioned only informally rather than by assertions. For example, when we say predicate B is preserved by an event e assuming some obvious properties, we mean that $\{B, C\}e\{B\}$ holds where C is a trivial property that is not formally specified.

9.2 Simplified bakery algorithm

Consider a system of processes $0, 1, \dots, N-1$. Each process i has an integer variable $\text{num}[i]$, initially 0, that is readable by all processes but writeable by process i only. When process i is thinking, $\text{num}[i]$ equals zero. When process i becomes hungry, it sets $\text{num}[i]$ to a value higher than the num of every other process; this operation is assumed to be atomic in this simplified algorithm. Then process i scans the num values of the other processes in order. For each process j , process i waits until $\text{num}[j]$ is either zero or greater than $\text{num}[i]$. After going past every process, process i enters the critical section. Upon leaving the critical section, process i zeroes $\text{num}[i]$. In the system below, $\pi\text{id}[i]$ is an auxiliary variable that records the formal id for process i ; it is needed to state assertions.

```
system-program simplified-Bakery(int N) {
  atomicity-assumptions{ reads and writes of integers, and a1 below }
  init() {
    int[0..N-1] num := 0 ; // num[i]=0 iff process i is thinking
    for i := 0 to N-1 do {
      Pid  $\pi\text{id}[i]$  := StartThread(code(i)); //  $\pi\text{id}[i]$  is auxiliary
    }
  }
}

function code(int i) {
```

```

forever do {
  NCS: noncritical section;
  // start entry code
  a1: num[i] := max(num[0], num[1], ..., num[N-1]) + 1 ; // a1 is atomic
    for p := 0 to N-1 do { // p is local to (process executing) entry(i)
      a2: while num[p]≠0 and num[p]<num[i] do no-op ;
    }
  // end entry code
  CS: critical section
  // start exit code
  b1: num[i] := 0 ;
  // end exit code
}
} // end code(i)

fairness-assumptions{ [∀ 0..N-1 i :: Wfair(πid[i])]
}

```

The desired correctness properties are the usual safety and progress assertions:

- S_0 : $\square [\forall 0..N-1 i, j, i \neq j :: Z(i, j)]$, where
 $Z(i, j) : \neg(\pi id[i] \text{ on CS} \wedge \pi id[j] \text{ on CS})$
- P_0 : $[\forall 0..N-1 i :: P(i)] \Rightarrow [\forall 0..N-1 i :: Q(i)]$, where
 $P(i) : \pi id[i] \text{ on CS} \rightsquigarrow \pi id[i] \text{ at b1}$
 $Q(i) : \pi id[i] \text{ at a1} \rightsquigarrow \pi id[i] \text{ at CS}$

9.3 Proof of simplified algorithm

We show that the given system satisfies the desired assertions. Instead of jumping straight into the details of an arbitrary execution, it is worthwhile, and more pleasant, to reason about the system in terms of the hypothetical queue of non-thinking processes in increasing order of their `num`. This queue is empty iff every process is thinking. A process joins the queue at the tail upon becoming hungry and executing `a1`; because `a1` is atomic, no two non-thinking processes have the same `num`. A process leaves the queue upon completing eating and executing `b1`. These are the only two ways in which the queue changes.

We will show that the process at the head of the queue, i.e., with the smallest `num`, is the only process that can eat. Furthermore, that process will not be blocked by any other process, and so it will eventually eat, after which all the processes in the queue move one step closer to eating.

Define the following for a process `i`:

- Let function `passed(i, j)` mean that `i` is non-thinking and has executed `a2` with `i.p = j` since it last became hungry. Formally, `passed(i, j) = ((i on CS, b1) ∨ ((i on a2) ∧ i.p > j))`
- Let function `pos(i)`, for a non-thinking process `i`, denote the position of `i` in the hypothetical queue. Formally, `pos(i) = |{j : 0..N-1 j, 0 < num[j] < num[i]}|`

We start with the safety proof. For any two non-thinking processes `i` and `j`, we expect that `i` has not passed `j` if `i` is behind `j` in the queue (i.e., `num(i) > num(j)`). Or equivalently, `i` has passed `j` only if `j` is either thinking (i.e., `num[j] = 0`) or `j` is behind `i` in the queue (i.e., `num[j] > num[i] > 0`). Formally,

$$A_1(i, j) : (i \neq j \wedge \text{passed}(i, j)) \Rightarrow (\text{num}[j] = 0 \vee \text{num}[j] > \text{num}[i] > 0)$$

We establish $\square A_1(i, j)$. $A_1(i, j)$ holds initially and is preserved by all events assuming some trivial properties. In particular: `πid[i].a1` falsifies `passed(i, j)`; `πid[i].a2` establishes the antecedent only if the consequent holds; `πid[i].b1` falsifies `passed(i, j)`; `πid[j].a1` establishes consequent; `πid[j].a2` establishes consequent; `πid[j].b1` establishes consequent; events of a process other than `i` or `j` do not affect $A_1(i, j)$.

The desired safety assertion S_0 follows from $A_1(i, j)$. Briefly, for any two processes i and j , $A_1(i, j)$ and $A_1(j, i)$ imply that both i and j cannot be in the CS (otherwise, $\text{passed}(i, j)$ and $\text{passed}(j, i)$ hold, which, together with $A_1(i, j)$ and $A_1(j, i)$ imply that $\text{num}[j] > \text{num}[i]$ and $\text{num}[i] > \text{num}[j]$, which is a contradiction).

We now deal with progress. A process i at the head of the queue, i.e., with the smallest non-zero num , eventually eats because it does not get blocked at a_2 (the only way that $\text{num}[j]$, $j \neq i$, can change is if j executes a_1 , in which case $\text{num}[j]$ becomes higher than $\text{num}[i]$). Formally, the following holds via $\pi \text{id}[i].a_2$ assuming some trivial properties:

$$L_1 : (\text{num}[i] \neq 0 \wedge \text{pos}(i) = 0) \rightsquigarrow (\pi \text{id}[i] \text{ on CS, } b_1)$$

Because eating is finite and b_1 is executed with weak fairness, L_1 implies the following:

$$L_2 : \text{pos}(i) = 0 \rightsquigarrow \pi \text{id}[i] \text{ on NCS}$$

Now consider a non-thinking process i that is not at the head of the queue. Its position does not worsen, because processes join the queue only at the tail. Its position eventually decreases because the process at the head eventually leaves the queue (from L_2). So we have

$$L_3 : (\text{num}[i] \neq 0 \wedge \text{pos}(i) = k > 0) \rightsquigarrow (\text{pos}(i) < k)$$

The desired progress assertion P_0 follows from the closure of L_3 and L_2 .

To flesh out the above to assertional proofs, one needs to explicitly specify and establish the trivial properties assumed above. For example, one would need to establish the invariance of $\pi \text{set} = \{\pi \text{id}[0], \dots, \pi \text{id}[N-1]\}$ and $\text{num}[i] = 0 \Leftrightarrow \pi \text{id}[i] \text{ on NCS}..a_1$.

9.4 Original bakery algorithm

We now consider the original bakery algorithm, that is, without assuming the max operation to be atomic. We will show that it works assuming only read-write atomicity of boolean variables and integer variables. Having a non-atomic max operation introduces two complications not present in the simplified bakery algorithm.

The first complication is that two processes can get non-zero nums with the same value. This is overcome by introducing a tie-breaking rule so that non-thinking processes with the same num value can resolve the tie in a consistent manner. The priority of a non-thinking process i is now given by the tuple $(\text{num}[i], i)$ in lexicographic ordering; that is, $(t, i) < (u, j)$ iff $t < u$ or $t = u$ and $i < j$. So process i goes past process j if it finds $\text{num}[j] = 0$ or $(\text{num}[i], i) < (\text{num}[j], j)$. It is important that processes i and j reach the same conclusion about their relative priority; otherwise i and j may each pass the other (and simultaneously enter CS) or i and j may each wait for the other (and stay deadlocked).

The second complication is that a process i , when deciding whether to pass another process j , may read $\text{num}[j]$ while j is in the middle of updating it, and thereby obtain some “intermediate” value of $\text{num}[j]$ that causes i to wrongly pass j . A danger here is that i passes j now but $(\text{num}[j], j)$ ends up being smaller than $(\text{num}[i], i)$, and so both end up in the CS at the same time. This complication is overcome by introducing a boolean flag $\text{choosing}[i]$ for each process i , indicating whether i is choosing a non-zero value for its num . When process i becomes hungry, it sets $\text{choosing}[i]$ to true, reads the nums of the other processes in some order, sets $\text{num}[i]$ to higher than the highest num value it has read, and sets $\text{choosing}[i]$ to false.

When process i wants to pass process j , it first busy waits until it finds $\text{choosing}[j]$ to be false, and only then does it start checking $\text{num}[j]$. So it is still possible for process i to read an unstable $\text{num}[j]$, but this can happen only if j started choosing its value after i finished choosing its value. So $\text{num}[j]$ will stabilize eventually to a value higher than $\text{num}[i]$. Hence there is no danger of i passing j now, j passing i later, and both ending up in the CS at the same time. As usual, process i enters the critical section after going past every process, and zeroes $\text{num}[i]$ upon leaving the critical section.

```

system-program Bakery(int N) {
  atomicity-assumptions{ reads and writes of integers and booleans }
  init() {
    boolean[0..N-1] choosing := false ; // choosing[i]=true iff i is choosing nonzero value for num
  }
}

```

```

int[0..N-1] num := 0 ; // num[i]=0 if process i is thinking
for i := 0 to N-1 do {
  Pid  $\pi$ id[i] := StartProcess(code(i)) ; //  $\pi$ id[i] is auxiliary
}
}

function code(int i) {
  forever do {
    NCS: noncritical section;
    // entry code
    a1: choosing[i] := true ;
    a2: num[i] := max(num[0], num[1], ..., num[N-1]) + 1 ;
    a3: choosing[i] := false ;
    for p := 0 to N-1 do { // p is local to entry(i)
    a4: while choosing[p] do skip ;
    a5: while num[p]  $\neq$  0 and (num[p],p) < (num[i],i) do skip ;
    }
    // end entry code
    CS: critical section;
    // exit code
    b1: num[i] := 0 ;
    // end exit code
  }
} // end code(i)

fairness-assumptions{  $[\forall 0..N-1 i :: Wfair(\pi id[i])]$ 
}

```

9.5 Proof of algorithm

We show that the given system satisfies the desired assertions. Again, instead of plunging into the details of an arbitrary execution, we reason about the system in terms of the hypothetical queue of non-thinking processes ordered by their extended timestamps. Unlike in the simplified algorithm, processes can join this queue at any position. In particular, if processes i and j have overlapping choosing durations, then j can complete choosing after i completes choosing and yet $num[j]$ is smaller than $num[i]$ (e.g., if $\pi id[j].a1$ read $num[i]$ before $\pi id[i].a1$ read $num[j]$). However, as we shall see, a process can leave the queue only from the head.

Define the following for a process i :

- Let function $pos(i)$ denote the position of a non-thinking process i in the hypothetical queue of extended timestamps in increasing order. Formally, $pos(i) = |\{j : 0..N-1 j, (num[j] \neq 0) \wedge (num[j], j) < (num[i], i)\}|$
- Let auxiliary variable $peers[i]$ be the set of processes that were choosing when i stopped choosing. Formally, let the update $peers[i] := \{j : choosing[j]\}$ be done atomically with **a3**.
- Let function $passed(i, j)$ mean that i is non-thinking and has already executed **a5** with $i.p = j$ and the while condition false. Formally, $passed(i) = ((i \text{ on CS, b1}) \vee ((i \text{ on a4..a5}) \wedge i.p > j))$

The desired correctness properties are the same as for simplified bakery algorithm, that is, S_0 and P_0 . We start with safety. We expect that if i has passed j then either (1) j is thinking, or (2) j is choosing and started choosing after i finished choosing, or (3) j has a stable non-zero num and is behind i in the queue. We also expect that if i is on **a5** with $i.p = j$ and $num[j] = 0$, and j is choosing, then j started choosing only after i completed choosing. Formally,

$$C_1(i, j) : (i \neq j \wedge \text{passed}(i, j)) \Rightarrow \\ (((j \text{ on NCS, a1}) \vee \text{choosing}[j]) \wedge j \notin \text{peers}(i)) \vee (\text{num}[j] \neq 0 \wedge (\text{num}[j], j) > (\text{num}[i], i))$$

$$C_2(i, j) : (i \neq j \wedge (i \text{ on a5}) \wedge (\pi \text{id}[i].p = j) \wedge (\text{num}[j] = 0) \wedge \text{choosing}[j]) \Rightarrow (j \notin \text{peers}(i))$$

We first establish $\square C_1$. $C_1(i, j)$ holds initially and is preserved by all events assuming $C_2(i, j)$ and some trivial properties. In particular: $\pi \text{id}[i].a3$ zeros p and so falsifies antecedent; $\pi \text{id}[i].a5$ with p thinking establishes consequent; $\pi \text{id}[i].a5$ with $\text{num}[p] = 0$ and $\text{choosing}[j]$ establishes consequent because of $C_2(i, j)$; $\pi \text{id}[i].a5$ with $(\text{num}[p], p) > (\text{num}[i], i)$ establishes consequent; events of j where $j = \pi \text{id}[i].p$ preserves $C_1(i, j)$; $\pi \text{id}[j].a1$ preserves the first conjunct of the consequent.

$C_2(i, j)$ is invariant as follows: $\pi \text{id}[i].a4$ with $p = j$ preserves it vacuously ($\pi \text{id}[i]$ comes to $a5$ only if $\text{choosing}[j]$ is false); $\pi \text{id}[j].a1$ preserves it vacuously.

The desired safety assertion S_0 follows from $C_1(i, j)$ and $C_1(j, i)$, just as in the case of the simplified algorithm.

We now establish progress. Unlike in the simplified algorithm, $\text{pos}(i)$ for a non-thinking process i is not monotonically decreasing, and so it is not an adequate metric for progress. But the two-tuple $(\text{peers}[i], \text{pos}(i))$ is an adequate metric under lexicographic ordering. The following hold while process i is non-thinking:

$$D_1 : ((\text{peers}[i], \text{pos}(i)) = k > 0) \text{ unless } ((\text{peers}[i], \text{pos}(i)) < k)$$

$$D_2 : (\text{peers}[i] = k > 0) \rightsquigarrow (\text{peers}[i] < k)$$

$$D_3 : ((\text{peers}[i], \text{pos}(i)) = 0) \rightsquigarrow (i \text{ on NCS})$$

$$D_4 : (\text{pos}(i) = 0) \rightsquigarrow (i \text{ on NCS})$$

$$D_5 : ((\text{peers}[i], \text{pos}(i)) = (k_1, k_2) > (0, 0)) \rightsquigarrow ((\text{peers}[i], \text{pos}(i)) < (k_1, k_2))$$

D_1 holds because (1) $\text{peers}[i]$ can only decrease (no process joins $\text{peers}[i]$), and (2) any increase to $\text{pos}(i)$ is simultaneously accompanied by a decrease in $\text{peers}[i]$ (because only a process in $\text{peers}[i]$ can enter the queue ahead of i). D_2 holds because every process in $\text{peers}[i]$ eventually leaves (since $a2..a3$ is non-blocking and processes have weak fairness). D_3 holds because i does not get blocked at $a5$ for any p . D_4 holds from $D_{1,2,3}$. D_5 holds if $k_1 > 0$ because of D_2 and D_1 . D_5 holds if $k_1 = 0$ and $k_2 > 0$ because of D_4 for the process at the head of the queue. The desired progress assertion P_0 follows from D_5 and D_3 .

9.6 Concluding Remarks

As mentioned earlier, the proofs are not completely assertional. But it is easy to obtain the additional assertions to flesh out the proofs to assertional proofs. Some of the additional assertions have been indicated above.

Finally, a lock manager can be obtained from the bakery algorithm using the transformation described in the previous chapter.

9.7 Exercises

- 9.1 Give assertional proofs of safety and progress for the simplified bakery algorithm, by adding any additional assertions needed to instantiate the proof rules.
- 9.2 Repeat problem 9.1 for the original bakery algorithm.
- 9.3 In the (original) bakery algorithm, each of the “for” loops scans the process in the order $0, \dots, N-1$. Is any order acceptable?
- 9.4 Parallelize the bakery algorithm as much as you can. For example, can process i start $N - 1$ threads and scan the other processes in parallel.

- 9.5 The algorithm in its current form allows the $\{\text{num}_i\}$ to grow without bound. Come up with a way to bound them, modify the algorithm appropriately, and prove that it still works (try to augment the current proof).

Chapter 10

Timestamps and Mutual Exclusion

10.1 Introduction

This chapter describes the concept of logical timestamps and illustrates its application to distributed mutual exclusion. Logical timestamps, or **timestamps** for short, is a mechanism by which processes in a distributed system can determine a consistent ordering of event executions in the system. Such a mechanism enables a distributed solution to resource sharing problems, including mutual exclusion. Essentially, the mechanism is used to order requests for resources, and conflicting requests are served in this order. The bakery algorithm is an example of this.

Clearly, it is crucial that all contenders perceive the same ordering. Otherwise two conflicting processes may each think it is ahead of the other and so simultaneously access a shared resource. Or each may think it is behind the other, resulting in deadlock. It is also crucial that the mechanism not discriminate against a contender to the point of starvation. Ideally, contending requests should be served in real-time order, but this requires a global observer with an infinite-precision clock, or equivalently, perfect infinite-precision clocks at the processes. In reality, the processes of the distributed system have a much more limited view of the global state of their system.

Consider two event executions x and y in the distributed system. We say x **causally precedes** y if (1) x and y happened in that order at the same process, or (2) x sent a message that y received, or (3) there is a sequence of such causal precedences leading from x to y , i.e., transitive closure. We say x and y are **causally linked** if one of them causally precedes the other. If x and y are not causally linked, then there is no way for any process in the distributed system to determine which actually happened first (without processes using real-time clocks or communicating over channels outside the distributed system). The best one can hope for is for all processes to agree on some arbitrary ordering of x and y . This is what the timestamp mechanism provides.

Below, we first describe the timestamp mechanism and state its generic properties. We then show how it can be used to solve distributed mutual exclusion. Finally, we present a distributed lock manager obtained directly from the mutex solution.

10.2 Logical timestamp mechanism

Consider a system with processes $0, 1, \dots, N - 1$, completely interconnected by fifo channels. For every two event executions x and y , we want every process that learns of x and y to order x and y in the same way. This is achieved by tagging each event execution with an integer timestamp and the id of the executing process. Event execution x is ordered before event execution y iff (1) x 's timestamp is less than y 's timestamp, or (2) their timestamps are equal and x 's process id is lower than y 's process id. In other words, x is ordered before y if x 's timestamp-id pair is *lexicographically* smaller than y 's timestamp-id pair. We shall refer to a timestamp-id pair as an **extended timestamp**.

The timestamp mechanism is as follows:

- Each process i has a non-decreasing integer counter, say clk , initialized to 0 (it can be an arbitrary value). This counter plays the role of a **logical clock**. An event execution at process i is identified by a tuple of the form (e, t, i) , where e identifies the event (e.g., the event signature) and t is the value of i 's clk when the event was executed. The tuple (t, i) is the extended timestamp of the event execution.
- Each process i also has, for every other process j , a variable $\text{hts}[j]$ indicating the highest timestamp received so far from j . Variable $\text{hts}[j]$ is initialized to 0 (or uninitialized if j 's clk is initially arbitrary).
- Whenever process i executes an event e that does not receive a timestamped message, it increments its clk before doing the event action and sends to all other processes the message (e, clk, i) .
- Whenever process i executes an event e that receives a timestamped message (f, t, j) , it updates its $\text{hts}[j]$ to t and its clk to $\max(\text{clk}, t) + 1$, thus ensuring that any future event execution at i will have a timestamp higher than t .

A process “learns” of a remote event execution when it receives the corresponding (e, t, i) tuple. Clearly, for any two event executions (e, t, i) and (f, u, j) , every process that learns of them will order them in the same way; specifically, $(e, t, i) < (f, u, j)$ iff $t < u$ or $t = u$ and $i < j$. Furthermore, if $(e, t, i) < (f, u, j)$ holds then either x causally precedes y , or x and y are not causally linked. So if x causally precedes y then $(x, t, i) < (y, u, j)$ holds. Finally, because clk is non-decreasing and the channels are fifo, a process i 's $\text{hts}[j]$ is a lower bound to j 's clk and any timestamp that i receives from j in the future will exceed $\text{hts}[j]$. So process i can be certain that the future will not bring an event execution with extended timestamp smaller than $\min_j(\text{hts}[j], j)$.

We express these properties assertionally. Let Q be the sequence of (e, t, j) tuples generated throughout the system, in order of increasing extended timestamp. Let Q_i be the sequence of (e, t, j) tuples generated or received by process i , also in order of increasing extended timestamp. Let H_i be the prefix of Q_i containing all tuples (e, t, j) such that $(t, j) < \min_j(\text{hts}[j], j)$. Then the following hold:

- \square (Q_i subsequence-of Q)
- \square (H_i prefix-of Q)

10.3 Mutual exclusion solution

A distributed solution to the mutual exclusion problem differs from the previous solutions we have seen in that the contending processes interact by message passing over fifo channels rather than via shared memory. Let the processes be identified 0 through $N - 1$. We have the usual requirements: each process cycles through thinking, hungry, or eating; at most one process should be eating at any time; and every hungry process should eventually eat provided every eating process eventually stops eating.

The timestamp mechanism can be applied to solve this problem as follows.

- In addition to the clock clk and the array hts , process i maintains an increasing sequence, say rqq , of extended timestamps of outstanding requests.
- Whenever process i becomes hungry, it increments its clk , sends the request message $(\text{req}, \text{clk}, i)$ to all other processes, and enters the extended timestamp (clk, i) into its rqq .
- Whenever process i receives a request message (req, t, j) , it updates $\text{hts}[j]$ and clk , enters the extended timestamp (t, j) into its rqq , and sends an ack message $(\text{ack}, \text{clk}, i)$ to j .
- Process i becomes eating when (1) rqq 's head is (t, i) for some t , and (2) $(t, i) < (\text{hts}[j], j)$ for every other process j . (This latter ensures that no request with a smaller extended timestamp can show up in the future.)
- When process i stops eating, it removes the $(*, i)$ entry from rqq (the entry would be at the head) and sends a release message (rel, i) to every other process (this message need not have a timestamp).

- When process i receives a release message (rel, j) , it removes the $(*, j)$ entry from rqq (the entry would be at the head).

The desired properties are the usual: two processes are not eating simultaneously, and every hungry process eventually eats provided every eating process eventually thinks.

For analysis, it is convenient to define some hypothetical quantities. We proceed a bit differently than in the bakery algorithm. We will have a global queue that consists of all processes, rather than only the non-thinking processes. The position of a thinking process in this queue will be the position it would have if it immediately became hungry. With this motivation, we define the following quantities. As usual, we use $\pi id[i]$ to reference process i .

- For any process i , let $rqt_s(i)$ denote the timestamp of i 's last request (which would be the current request if i is non-thinking).
- For any process i , let $val(i)$ equal $\pi id[i].clk + 1$ if i is thinking and $rqt_s(i)$ if i is non-thinking.
- For any process i , let $pos(i)$ denote $|\{j : (val(j), j) < (val(i), i)\}|$. [That is, the number of processes ahead of i where j is ahead of i if (1) j is non-thinking with a smaller extended timestamp, or (2) j is thinking and its clock is such that it will be ahead of i if it immediately becomes hungry.]
- Let function $passed(i, j)$ mean that i is non-thinking and has overtaken j . Formally, $passed(i, j) = ((\pi id[i]$ non-thinking) and $((\pi id[i].rqt_s, i) < (hts[j], j))$ and $((t, j) \text{ in } \pi id[i].rqq) \Rightarrow (\pi id[i].rqt_s, i) < (t, j))$)

We start with the safety proof. For any two non-thinking processes i and j , we expect that i has not passed j if i is behind j in gq . Or equivalently, i has passed j only if j is either thinking or j is behind i in the queue. Formally,

$$A_1(i) : (\pi id[j] \text{ eating}) \Rightarrow [\text{forall } j :: \text{passed}(i, j)]$$

$$A_2(i, j) : (i \neq j \wedge \text{passed}(i, j)) \Rightarrow (\text{pos}(j) > \text{pos}(i))$$

It is left as an exercise to establish $\Box A_1(i)$ and $\Box A_2(i, j)$. Basically, $A_1(i)$ holds initially and is preserved by every event assuming trivial properties. $A_2(i, j)$ holds initially and is preserved by every event assuming $A_2(j, i)$ and trivial properties.

We now deal with progress.

$$B_1 : ((\pi id[i] \text{ non-thinking}) \text{ and } \text{passed}(i, j)) \text{ unless } (\pi id[i] \text{ thinking})$$

$$B_2 : ((\pi id[i] \text{ hungry}) \text{ and } \text{pos}(i)=k) \text{ unless } ((\pi id[i] \text{ hungry}) \text{ and } \text{pos}(i)>k)$$

$$P_1 : ((\pi id[i] \text{ hungry}) \text{ and } \text{pos}(i)=0) \rightsquigarrow (\pi id[i] \text{ thinking})$$

$$P_2(k) : ((\pi id[i] \text{ hungry}) \text{ and } \text{pos}(i)=k) \rightsquigarrow (\pi id[i] \text{ thinking})$$

B_1 and B_2 hold by the unless proof rule assuming trivial properties.

We now establish P_1 . note that i hungry with $\text{pos}(i) = 0$ leads-to $\text{passed}(i, j)$ for every other process j , because i will eventually receive a timestamp from j that is higher than $rqt_s(i)$. So i will eventually pass all other processes, at which point its eating event is continuously enabled and will eventually occur.

We now establish $P_2(k)$. The proof is by induction over k . P_1 provides the base case. Assume P_2 holds for all $k < K$. Suppose i is hungry and $\text{pos}(i) = K$. If there is a hungry process j with $\text{pos}(j) < \text{pos}(i)$, then j will eventually eat and become thinking (by induction hypothesis), at which point $\text{pos}(i)$ becomes less than K (and we are done). If all the processes ahead of i stay thinking, one of them will receive i 's request, at which point its position becomes higher than i 's (and we are done).

10.4 Lock manager

The above can be easily converted to a lock manager that offers the lock service using a fifo message passing service as follows. Below, `Fifo` denotes the message passing service and `Msg` is a record with fields `kind`, `ts`, and `id`, where `ts` and `id` are integers and `kind` is either `req`, `rel`, or `ack`.

```

system-program LockX(int i) {
  init() {
    type Xts = record{int ts; int id};
    int clk := 0;
    int[0..N-1] hts := 0;
    XtsSeq rqq := ⟨⟩;
    type Ttx = record{int to; Msg m};
    TtxSeq sbuff := ⟨⟩; // queue of messages to be transmitted
  }

  xc-event Req() {
    maps LockService.Req(i)
    ec true
    ac clk := clk+1;
    enter (clk,i) in rqq;
    for j := 0 to N-1 except i do append(sbuff, (j,(req,clk,i)));
  }

  xc-event Rel() {
    maps LockService.Rel(i)
    ec true
    ac remove head from rqq;
    for j := 0 to N-1 except i do append(sbuff, (j,(rel,i)));
  }

  xc-event Rx(Msg m) {
    maps Fifo.Rx(m)
    ec true
    ac clk := max( clk, m.ts ) + 1;
    hts[m.id] := m.ts;
    if (m.kind = req) then {
      enter (m.ts,m.id) in rqq;
      append(sbuff, (m.id, (ack,clk,i)));
    }
    else if (m.kind = rel) then {
      remove head from rqq;
    }
    // else m.kind=ack, nothing more to do
  }

  lc-event Tx() {
    ec sbuff not empty
    ac Ttx x := remove(sbuff);
    Fifo.Tx(x.to, x.m)
  }

  lc-event Grant() {
    ec (rqq not empty) and (rqq[0].id = i)
    ac LockSrv.Acq(i)
  }

```

```

}

fairness-assumptions { lc-events are weak fair}

} // end LockX(i)

```

Let $\text{LockMgr}(N)$ be the composite system of $\text{LockX}(0), \dots, \text{LockX}(N-1)$. It is left as an exercise to show that $\text{LockMgr}(N)$ offers $\text{LockSrv}(N)$ using $\text{Fifo}(0..N-1, \text{Msg})$.

10.5 Concluding Remarks

The timestamp mechanism provides a general framework for distributed resource-sharing problems. Each request is associated with an extended timestamp. Conflicting requests are served in their extended timestamp order. Starvation does not occur because extended timestamps are consistent with causal precedence and because requests and releases are broadcast to all processes. So once a request is propagated to all processes, that request will be served ahead of any future requests.

It is easy to see how the timestamp mechanism can be used to solve other resource sharing problems. For example, one can solve the readers-writers problem with just a few changes to the above mutual exclusion algorithm: (1) there are read-requests and write-requests; (2) a process starts writing when, as in the mutex case, its rqq has its write-request extended timestamp (t, i) at the head, and, for all other j , $(\text{hts}[j], j) > (t, i)$ holds. (3) a process starts reading when its rqq has no write-requests ahead of its read-request, and, for all other j , $(\text{hts}[j], j) > (t, i)$ holds.

The main disadvantage of the logical timestamp mechanism is that every process talks to every other process. This is acceptable for mutual exclusion and readers-writers, when such global communication is inherent in the problem. But it is not good for a dining philosophers problem, where one would hope for a solution in which only neighbors talk. We will see such solutions later.

One optimization to the mutual exclusion algorithm is as follows: when a non-thinking process i receives a request of lower priority from process j , process i withholds sending the ack until it finishes eating (instead of sending an ack immediately and a release after eating). This optimization results in the Ricart-Agrawala algorithm.

10.6 Exercises

- 10.1 Give assertional proofs of safety and progress for the mutex algorithm, by adding any additional assertions needed to instantiate the proof rules.
- 10.2 Obtain a proof for the Ricart-Agrawala algorithm by modifying the proof for Lamport's algorithm.
- 10.3 The algorithm in its current form allows the clocks to grow without bound. Come up with a way to bound them, modify the algorithm appropriately, and prove that it still works (try to augment the current proof).