Mutual Exclusion Algorithms in the Shared Queue Model*

Junxing Wang and Zhengyu Wang

The Institute for Theoretical Computer Science (ITCS),
Institute for Interdisciplinary Information Sciences,
Tsinghua University, Beijing, China
thuwjx@andrew.cmu.edu, wangsincos@163.com

Abstract. Resource sharing for asynchronous processors with mutual exclusion property is a fundamental task in distributed computing. We investigate the problem in a natural setting: for the communications between processors, they only share several queues supporting enqueue and dequeue operations. It is well-known that there is a very simple algorithm using only one queue when the queue also supports the peek operation, but it is still open whether we could implement mutual exclusion distributed system without the peek operation. In this paper, we propose two mutual exclusion starvation-free algorithms for this more restricted setting. The first algorithm is a protocol for arbitrary number of processors which share 2 queues; the second one is a protocol for 2 processors sharing only one queue.

Keywords: Mutual Exclusion, Starvation Free, Queue, Distributed Algorithm.

1 Introduction

In a distributed system, processors often need to share a unique resource, while at the same time it is required that the processors are mutual exclusive, which means that at any time, there is at most one processor who uses the resource (i.e. enters the critical section). That fundamental problem is known as mutual exclusion problem. Dijkstra [1] gives the first formulation of the problem, and devices an algorithm with shared atomic registers for that purpose. However, the algorithm does not satisfy starvation free property, i.e., there might be some processor who wants to enter the critical section, but it never enters. Lamport [2] further proposes bakery algorithm obtaining starvation free property, and moreover, the processors share weaker registers called safe registers. There are a huge amount of literatures on the design of mutual exclusion algorithms, which can be even formed a book several decades ago [3].

* This work was supported in part by the National Basic Research Program of China Grant 2011CBA00300, 2011CBA00301, the National Natural Science Foundation of China Grant 61033001, 61061130540.

M. Chatterjee et al. (Eds.): ICDCN 2014, LNCS 8314, pp. 29–43, 2014.
© Springer-Verlag Berlin Heidelberg 2014

Shared queues have been studied as objects for communication between processors. Informally, the *enqueue* operation inserts an element to the tail of the queue; the *dequeue* operation returns as well as deletes the head element of the queue; the *peek* operation returns the head element of the queue without modifying it. Specially, if there is no element in the queue, it will return a symbol indicating the queue is empty upon the *dequeue* or *peek* operation. It is well known that if the queue object supports *enqueue*, *dequeue* and *peek* operations, we can use a single queue to implement a mutual exclusion system satisfying starvation free property: initially let the queue be empty; if one processor wants to use the resource, it will enqueue its *id* to the queue and then peek the queue over and over again until it finds that the top element of the queue is the id of itself so that it could enter the critical section; after one's critical section, it removes the top element of the queue by *dequeue* and then exits. Notice that the *peek* operation is the key for obtaining mutual exclusion property, and the queue structure enables starvation free property. Since *peek* is a more advanced operation of the queue object, one interesting question is that *can we succeed without the peek operation?* We have not found the answer to the question in the literature, and we found it to be non-trivial to solve. In this paper, we give a positive answer by showing two algorithms. The first algorithm uses two queues to support arbitrary number of processors, and the second one uses one queue to support 2 processors.

In our model, we assume that the processors are failure-free, and for every processor, if it is active, it will execute a step in finite time. More formally, the problem concerned here is how to design a protocol with 1 or 2 queues without peek operation such that the distributed system satisfies the following properties.

a) Mutual exclusion property: there is no time when two processors run in the critical section.

b) No starvation (starvation-free) property: every processor that wants to enter the critical section eventually enters the critical section.

Note that the problem studied in our paper is different from consensus in that (1) In the problem setting of making consensus, one processor might crash and never recover again; while in our model, one processor never crash (or might crash, but recover in finite time and return to the state before crash); (2) The consensus problem (or calculating the consensus number) typically concerns only making one consensus, while mutual exclusion deals with many critical sections. Furthermore, in the consensus problem, we have not found the concept of starvation-free. It might be interesting to consider that property, since it is natural and fair to make every proposer having the chance of being admitted in the period of infinite times of consensus making.

We can define a concept called mutual exclusion number, denoted by MEN, which is similar to the definition of consensus number [4]. We call a system has mutual exclusion number k, if it can support starvation-free mutual exclusion requirement of k processors, while it cannot support $k+1$ processors. If it can support arbitrary number of processors, we define that number to be ∞. For example, we have already known that $MEN\,(n$ atomic registers$) \geq n-1$ by

Bakery Algorithm. And the conclusion of our work can be written as

(1) MEN (1 queue with peek) $= \infty$;
(2) MEN (2 queues without peek operation) $= \infty$;
(3) MEN (1 queue without peek operation) ≥ 2.

The paper is organized as follows. Section 2 gives a protocol with 2 queues that supports arbitrary number of processors. In Section 3, we present a protocol with one queue that supports 2 processors. Finally we conclude and refer to further work in Section 4.

2 Algorithm with Two Queues

In this section, we present an algorithm using two queues that solves the mutual exclusion problem for arbitrary number of processors. In the first part, we exhibit some intuitions behind the design of the algorithm. Next, we give some data type notation which will be used in the algorithm. Then we show the proposed algorithm, and finally give a technical correctness justification in the last subsection.

2.1 Designing Idea

In order to satisfy mutual exclusion property, we could adopt a token-based approach: place a unique token in queues; any processor can get into critical section only if it obtains the token from queues by *dequeue*; when the processor leaves its critical section, it should return the token back to queues by *enqueue*. To satisfy no starvation property, we should design a rule to guarantee that all the processors with intent to enter critical section will get the token in a finite time. Since every active processor must *dequeue* from queues to get messages from other processors, it becomes a trouble that how to avoid the processors getting the token who are not supposed to do so. To tackle it, we could attach the token with additional information such as the processors which have the right to enter critical section, operations that each processors are executing, etc. As a result, the processor which gets the token "wrongly" can take some actions to "help" the processor which is supposed to get the token according to information attached with the token. At the same time, we should also make sure that the new processor with intent to enter critical section can notify other processors, the processors which have the ability to guide it to get the token.

2.2 Notation

Before describing the proposed algorithm, we need to define some notations that will be used in the following algorithm description and analysis. Suppose that there are n processors numbered with 1 to n, which have intent to get into critical section at any time they want (or will never have such intent). Mark two shared queues mentioned above with q_0 and q_1. To formalize the notation of operations,

we use $enqueue(q_i, x)$ to represent command that inserts the element x to the tail of q_i. Also, $x \leftarrow dequeue(q_i)$ means that the processor reads the head element of q_i and save it into variable x together with removing that element form q_i.

All the elements in queues are of type *number* or of type *order*. For element x, we can acquire the type of x by referring $x.type$. For instance, x's type is *number* if and only if $x.type = number$. Specially, if we execute an operation $x \leftarrow dequeue(q_i)$ while q_i is empty, we can obtain *NULL* while referring $x.type$.

While one processor executes the operation $x \leftarrow number(i)$, it can get an element x of type *number* with an attached value i which can be referred by $x.value$, which ranges from 1 to n. We can regard type *number* as an indicator telling the processor who gets this element that processor i wants to get into critical section. Consider the case that $x.type = order$, i.e., x is of type *order*. We can refer $x.queue$ to get a sequence of numbers, which can be empty (i.e., $x.queue = \phi$). We can obtain the head number of $x.queue$ by $x.head$. If $x.queue = \phi$, mark $x.head$ to be 0. We can also refer $x.wait$ to get an n-dimension vector, ith entry of which is a 0-or-1 value, acquired by notation $x.wait[i]$. Such an element x, of type *order*, is just the token we mentioned in Subsection 2.1.

2.3 Algorithm

To begin with, let q_0 contain only one element x of type *order*, i.e., $x.type = order$, where $x.queue = \phi$ and $x.wait[i] = 0$ for all $i \in \{1, 2, \cdots, n\}$. Then, if processor i wants to get into critical section, it should run the following algorithm DQB for processor i. In this algorithm, processor i has some local memory: a 0-or-1 variable k, a queue tmp containing at most n numbers, a Boolean variable $allow$ and two temporary variables x and y used to record the element dequeued from queues.

The proposed algorithm is depicted in Algorithm 1.

2.4 Properties

In this subsection, we will discuss two important properties of algorithm DQB: mutual exclusion and no starvation.

Call processor i as p_i and mark p_i's local queue as tmp_i. For convenience, we divide the whole algorithm into three parts: trying section (from Line 1 to Line 10), control section (from Line 11 to Line 25) and exit section (from Line 26 to the end). If we say trying-control section, we mean the union of trying section and control section. In the same way, we say control-exit section to express the union of control section and exit section. To claim these sections clearly, if we say that processor i enters or gets into one section, we mean that processor i has executed one command of this section. To say that processor i leaves or gets out of one section, we mean that processor i has executed the last operation-command in this section, the command that would change the memory used in the whole algorithm including shared memory and local memory, and next operation-command p_i will execute is out of this section. For instance, if processor i with local variable $allow$ true has just executed the command at Line

Algorithm 1. Algorithm DQB for Processor i

```
1:  enqueue(q_1, number(i))                                   ▷ trying section
2:  k ← 0
3:  repeat
4:      tmp ← φ
5:      repeat
6:          x ← dequeue(q_k)
7:          if x.type = number then
8:              tmp ← (tmp, x.value)
9:          end if
10:     until x.type = order
11:     allow ← (x.head ∈ {0, i})                             ▷ control section
12:     x.queue ← (x.queue, tmp)
13:     repeat
14:         y ← dequeue(q_1)
15:         if y.type = number then
16:             x.queue ← (x.queue, y.value)
17:         end if
18:     until y.type = NULL
19:     if allow = false then
20:         x.wait[i] ← 1 − k
21:         enqueue(q_k, x)
22:         k ← 1 − k
23:     end if
24: until allow = true
25: Critical Section
26: Delete number i from x.queue                              ▷ exit section
27: x.wait[i] ← 0
28: if x.queue = φ then
29:     enqueue(q_0, x)
30: else
31:     enqueue(q_{x.wait[x.head]}, x)
32: end if
33: Remainder Section
```

14 and has obtained its local element y of type *NULL*, we can say that processor i leaves control section because the next several commands (at Line 15, 18, 19, 24, 25) processor i will execute make no chance on the memory used in the algorithm. If we say that processor i is running in one section, we mean that processor i has entered this section and has not leaved yet. For simplicity of the proof, we postulate that local queue *tmp* should be cleared up automatically after executing Line 12, and that local element x should be cleared up after executing $enqueue(\cdot, x)$ command. Such postulations make no chance of the function of algorithm.

Now, we will show some basic observations and lemmas. In the algorithm, a processor can get into control-exit section if and only if it acquires an element of type *order* at Line 6 and then executes the first command at Line 11. Whatever

its local variable *allow* returned by command at Line 11 is, the last command it will execute before leaving is *enqueue* command at Line 21 or 29 or 31, which is also the first command of *enqueue* operation it executes in control-exit section, i.e., an element of type *order* is enqueued back to queues if and only if this processor leaves control-exit section. These can be concluded in the observation below.

Obseravation 1. *A processor has an element of type* order *in its local memory if and only if it is running at control-exit section.*

Based on this observation, we will propose the following lemma about uniqueness of the element of type *order*, which will be used on whole analysis of this paper.

Lemma 1 (Conservation of *Order*)
There is exact one element of type order in queues and all processors' local memories.

Proof. Call the time as *free* when there is no processor in control-exit section. Oppositely, if there exists one processor in control-exit section, we mark this moment as *busy*.

Let's consider the case in *free* time first. If there is exact one element of type *order* in queues (we call this as one-element-in-queues condition), Lemma 1 can be proved because all the processors have no element of type *order* in their memories by Observation 1. We say beginning time is such a *free* time that satisfies one-element-in-queues condition.

From the beginning time on, Lemma 1 keeps correct until a processor i obtains an element of type *order* and will enter control-exit section later. After processor i gets into control-exit section, the time becomes *busy*. This time no elements of type *order* can be found in queues. Then, no processor can acquire an element of type *order* by *dequeue* unless one processor *enqueues* it. In *busy* time, the only case that the element of type *order* is *enqueued* to queues is that processor i executes its last command and gets out of control-exit section. Therefore, queues and all the processors except i can not get the element of type *order* in *busy* time. As processor i holds it during the whole *busy* time, Lemma 1 keeps correct.

After processor i leaves control-exit section, queues gains an element of type *order* again, which satisfies one-element-in-queues condition. Therefore, the case become the first one again. Lemma 1 can keep correct forever.

According to Lemma 1, if we say *order* in the following context, we refer to that unique element of type *order* placed in queues or one process's local memory. Combining Lemma 1 with Observation 1, we can directly get the following lemma.

Lemma 2. *At most one processor is running in control-exit section.*

Using the similar analysis, we can also acquire some observations as follows.

Obseravation 2 (Conservation of *Number*). *processor i is running in trying-control section if and only if exact one of following events happens.*

1. *q_1 contains number(i) exactly once;*
2. *order.queue contains i exactly once;*
3. *there exists exact one processor whose local queue contains i exactly once.*

Obseravation 3. *If order.head $\neq 0$, order is stored in $q_{order.wait[order.head]}$ or one process's local memory. Otherwise, order is stored in q_0 or one process's local memory.*

Obseravation 4. *processor i executes dequeue(q_k) (Line 6) in trying section only if order.wait[i] $= k$.*

Now, let's consider the property of Mutual Exclusion first.

Theorem 1. *Algorithm DQB satisfies Mutual Exclusion.*

Proof. In the algorithm, critical section is in control-exit section. Therefore, Mutual Exclusion is directed indicated by Lemma 2.

Next, let's discuss the property of No Starvation.

Lemma 3. *If one processor is running in control section, it will get out of control section in a finite time.*

Proof. If one processor stays in control section forever, it must keep repeating the commands in repeat-until paragraph from Line 13 to Line 18. Then we can only consider that processor i is running in control section before passing repeat-until paragraph. Suppose that the current time is t while there are m processors running the algorithm for them. By Observation 2, there are at most m elements of type *number* in q_1 at this time. By Lemma 2, no processor can enter exit section except p_i before p_i ends. Therefore, the *enqueue* command at Line 1 can be executed at most $n - m$ times before p_i gets out of control section, i.e., processor i can get non-empty local element y by *dequeue* operations at most $m + (n - m)$ times before it gets out of control section. Then, in a finite time, processor i will *dequeue* and obtain the element of type *NULL* stored in p_i's local memory y, and pass the until-condition at Line 18. Finally, it will get into critical section and exit section if its local variable *allow* is true, or it will return trying section in the case *allow* = *false*.

Here, we claim an extended version of Lemma 3, which can be directed demonstrated based on Lemma 3. It is that *if one processor is running in control-exit section, it will leave in a finite time.*

Lemma 4. *If order.head $= i \neq 0$, processor i has been in critical section or will get into critical section in a finite time.*

Proof. Because of the command at Line 26, processor i is impossible to run in exit section. Then we will discuss this lemma in two parts according to the section processor i is running in.

In first part, let's consider the case that p_i is running in control section but has not entered critical section now. processor i's local variable *allow* must be true because there is no operation in control section that can modify the head element in *order.queue*, i.e., $order.head = i$ or $order.head = 0$ (in the case that processor i inserts i into the empty *order.queue*) when p_i executed the first command (at Line 11) of control section. By Lemma 3, processor i will get out of control section in a finite time. As its local variable *allow* is true, it can pass until-condition at Line 24 and then get into critical section.

In second part, we will discuss the case that p_i is running in trying section at t_0. Let a dynamic set S_t for time t contain all the processors that will execute their next *dequeue* operation at $q_{order.wait[i]}$ (say q^* for short in following analysis). Before p_i gets into critical section, any other processor will enter control section with local variable $allow = false$, and will not be able to pass until-condition at Line 24. Therefore, there will be finite times for processors to *enqueue* the elements of type *number* to q^* before p_i gets into critical section. By Observation 2, there are finite elements of type *number* in q^*. According to the similar analysis in Lemma 3, it is impossible for processors to execute $dequeue(q^*)$ acquiring an element of type *number* forever. By Observation 3, *order* is placed in q^*. Then, from any time $t \geq t_0$ to the moment p_i enters critical section, there will be a processor $j \in S_{t+\Delta t}$ (Δt represents a finite period of time) acquiring *order* by *dequeue* operation. If $j \neq i$, processor j enters control section with local variable $allow = false$. In a finite time, processor j will execute the command at Line 20, which changes $order.wait[j]$. Suppose that p_j reaches Line 19 at time t_- and finishes the command at Line 20 at time t_+. The dynamic set S_t updates according to $S_{t_+} \leftarrow S_{t_-} \backslash \{j\}$. As time t goes on, the elements in the dynamic set S_t will be eliminated one by one. According to Observation 4, we guarantee $i \in S_t$ at time t. Therefore, there will be a moment p_i obtains *order*. In extreme case that all the elements except i will be eliminated from S_t as time t goes on, processor i will still acquire *order* in a finite time. Now, we can only consider the case $j = i$ which will happen in a finite time. In this case, processor i will enter control section and then it will critical section in finite time by the proof in first part.

Lemma 5. *From any time on, queues will contain* order *in a finite time.*

Proof. We can only consider the case that queues do not contain *order* now. By Lemma 1, there must be a processor owning *order* in the local memory. According to 1, this processor is currently running in control-exit section. Using the extended version of Lemma 3, this processor will leaves control-exit section together with inserting *order* by *enqueue* operation in a finite time.

Lemma 6. *If order.queue contains i, processor i has been in critical section or will get into critical section in a finite time.*

Proof. Suppose the *order.queue* contains i currently at time t_0. Consider the first time t_1 at or after t_0, when *order* is placed in queues. By Lemma 5, t_1 is bound to be a finite number.

If number i is not in *order.queue* at time t_1, processor i must have entered critical section during the time t_0 to t_1, because the command to delete i from *order.queue* is right down the critical section in algorithm. In fact, this case happens when processor i is running in control-exit section with local variable $allow = true$ at time t_0. Otherwise, *order.queue* contains number i. Then it will keep non-empty till processor i enters critical section, i.e., $order.head \neq 0$ before processor i gets into critical section. Therefore, processor j enters critical section before p_i does only if $order.head = j$. In other words, the next processor to enter critical section is processor *order.head*. By Lemma 4, processor *order.head* will get into critical section in a finite time. Then it will delete the head element of *order.queue* at Line 26. Meanwhile, the rank of number i (i.e., the number of elements prior to i) in *order.queue* will be decreased by 1. Thereby, the rank will be 0 in a finite time. At that time, processor i, the processor indicated in the head element of *order.queue*, will get into critical section finally.

Lemma 7. *If q_1 contains number(i), there is one event of the following two that will happen in a finite time.*

1. *number i is in a certain processor j's local queue tmp_j;*
2. *order.queue contains number i.*

Proof. In the algorithm, the elements of type *number* can be dequeued only at Line 6 or Line 14. If a certain processor j acquires *number* at Line 6, it will store it in its local queue tmp_j. If one processor obtains it at Line 14, it will be inserted to the tail of *order.queue*. Therefore, we should only claim that these two events will happen in a finite time.

Suppose that q_1 contains *number*(i) at time t_0. Pick the first time t_1 at or after t_0 when *order* is placed in queues. By Lemma 5, t_1 is a finite time.

Now consider the case that *number*(i) is still in q_1 at time t_1. If $order.head = 0$ at time t_1, *order* should be contained in q_0 by Observation 3. At this time, there are at least one processor including p_i trying to dequeue at q_0. Therefore, a certain processor j will obtain *order* in a finite time. (We can get this result by the similar proof in second part of Lemma 4. Then processor j will keep executing *dequeue*(q_1) until it obtains an empty reply. This procedure will be finished in a finite time by Lemma 3. Before this procedure ends, one of two events must have happened. In the other case that $order.head = j \neq 0$, processor j will get into critical in a finite time by Lemma 4. Before processor j enters critical section, it will get *order* and clear up q_1, before which two events mentioned in this lemma must have happened.

To prove that the algorithm has no starvation property, we should consider such a dynamic graph model. For any time t, we can construct a directed graph $G_t = (V, E_t)$ where $V = \{0, 1, \cdots, n\}$, and where an edge from i to j exists in E_t if and only if one event of the following two happens.

1. tmp_i contains j
2. $i = 0$ and *order.queue* contains j

Lemma 8. *If there is a directed path from* 0 *to* i *in* G_t*, processor* i *will get into critical section in a finite time after time* t*.*

Proof. In a specific dynamic graph G_t, we shall prove this lemma using mathematical induction on the length of the directed path.

Clearly, the lemma holds when the length of a directed path is 1, since if there is a directed edge from 0 to i in G_t, i.e., *order.queue* contains i at time t, processor i will enter critical section in a finite time by Lemma 6.

Suppose the lemma holds for the directed path of length k in G_t. That is, if there is a directed path of length k from 0 to i in G_t, processor i will get into critical section in a finite time after time t. Consider the case of the length of directed path being $k + 1$. For any directed path of length $k + 1$ from 0 to i in G_t, there must be an unique node j which has an out-edge pointing to node i in G_t by Observation 2. In the same time, there is a directed path of length k from 0 to j in G_t. By the induction hypothesis, processor j will enter critical section in a finite time after time t. Before that, processor j must execute the command at Line 12 before entering critical section. Therefore, i must be added to *order.queue*. By Lemma 6, processor i will enter critical section in a finite time. Hence, if lemma holds for the directed path of length k, it also holds for the directed path of length $k + 1$.

By the principle of mathematical induction, we conclude that for all natural numbers k, if there is a directed path of length k from 0 to i in G_t, processor i will get into critical section in a finite time after time t.

Lemma 9. *If one node has an in-edge in* G_t*, there is a directed path from* 0 *to it.*

Proof. At the beginning time t_0, G_{t_0} is a graph with no edge, while the lemma is apparently correct. Suppose that the lemma is correct from time t_0 to time t_1, let's consider the case in time t_2, which is the atomically-next time of t_1, i.e., there is at most one command executed from t_1 to t_2. In fact, we should only consider the commands at Line 8, 12, 16 and 26, which insert or delete edges on G_{t_1} so that new dynamic graph G_{t_2} is different from G_{t_1}. Among these, the commands at Line 8 and 16 insert in-edges to the nodes without any in-edges. The command at Line 12 connects the nodes, which already have in-edges, with some other in-edges. For the command at Line 26, it makes a node isolated by deleting its in-edge. Now, we will discuss all these commands respectively.

If processor i executes the command at Line 8 during the time t_1 to t_2, we can construct G_{t_2} by inserting an edge from node i to j on G_{t_1} where j is the number indicated in a *number* element in q_1 at time t_1 and node j is isolated node in G_{t_1}. During this time, processor i's local variable k must be 1 because elements of type *number* can only be *dequeued* in q_1. Hence we can imply that processor i obtained at least one *order* before, because only if processor i has ever acquired *order*, can it have the opportunity to make a change on its local variable

k from 0, the initial value of k, to 1. In the algorithm, processor i must clear up q_1 before it enqueues *order*. Before p_i got an empty reply while dequeuing q_1, $number(i)$ must be read by some processor and then be stored in *order.queue* or one process's local queue, i.e., node i must have an in-edge before. As the only command which can make node i connected with no in-edges is the command at Line 26 in exit section and processor i is still running at trying section, it is impossible for it to run that command. Therefore, node i still has an in-edge at G_{t_1}. By the assumption that the lemma is correct in G_{t_1}, there exists a directed path from 0 to i in G_{t_1}. Hence, there must be a directed path 0 to j in G_{t_2}, and so the lemma holds on this case.

Next, let's consider the case that processor i executes the commands at Line 12 and 16. Whatever processors do during the time t_1 to t_2, all the nodes with in-edges to be changed or added in G_{t_1} will be connected to node 0 by in-edges pointing to them from 0 in G_{t_2}. Therefore, the lemma holds on this case too.

Finally, consider the case when one processor executes the command at Line 26 which deletes number i from *order.queue*. After executing this command, a directed edge from 0 to i is deleted from G_{t_1} and node i will become the node without any in-edge in G_{t_2}. In fact, node i will be isolated in G_{t_2} because processor i must execute the command at Line 12 before entering critical section, which deleted all the out-edges from node i. Therefore, this command makes no influence on other nodes in G_{t_2}, i.e., the lemma also holds on this case.

Theorem 2. *Algorithm DQB satisfies no starvation property.*

Proof. For any processor i which wants to get into critical section, it should enqueue an element of type *number* which contains the value of i to q_1. By Lemma 7, number i will be contained either in one process's local queue or *order.queue* in a finite time. At this moment t, there is an in-edge for node i on the dynamic graph G_t. By Lemma 9, there is a directed path from 0 to i on G_t. According to Lemma 8, processor i will get into critical section in a finite time after time t. Therefore, Algorithm DQB satisfies no starvation property.

3 Algorithm with One Queue

In this section, we show a protocol which implements a mutual exclusion system for two processors with one single queue. We first specify the data format for elements in the queue, together with the initial value of the queue. Then we describe the proposed protocol in pseudo-code in Algorithm 2. Due to the length limit, we leave the correctness proof to the online version.

3.1 Data Format and Initial Value

Every element in the queue is a record with several fields. The first field is *type*, whose value is either OFFER or APPLY, identifying the type of the record.

If the type is OFFER, meaning that it is an invitation for some processor (or both) to enter the critical section, then the second field will be *name* (whose

value is 1, 2 or ALL), indicating the processor to which the offer will give. The third field is *id*, which starts from 1 and goes larger, making the offer unique, (in fact, its concrete meaning in our algorithm is that after accepting that offer, it will be the *id*-th time for these two processors to use the resource), so that we can use this information to discern overdue offers. If *name* is not equal to ALL, there will be a fourth field *version* (a natural number, and in the following algorithm it will always be 1 or 2), which serves for identifying different versions of the same information (i.e. records with the same type, name and id), and in the following algorithm, you will find the use of the *version* field to convince one processor that its command (APPLY or OFFER) has been successfully received by the other, which is one crucial technique for our success.

If the type is APPLY, meaning that it is the application of a processor for using the resource (i.e., for entering the critical section), there will be two more fields, *name* (the one who applies) and *version* (the same meaning as above)

We will write the record as

$$(type, name[[, id], version]).$$

And if we *dequeue* when the queue is empty, we will get EMPTY as return.

At the beginning, the only element in the queue is (OFFER,ALL,1).

3.2 The Protocol

We give the proposed protocol in Algorithm 2. Because of the length of the protocol, we list the types and initial values of the variables used in the protocol here. Variable i is either 1 or 2, indicating the processor number. We use i' to represent $3 - i$, i.e. the other process, in a concise way. Variable j is a temporary variable for version number. Temporary variable t will be used for storing one record dequeued from the queue. The initial value of the temporary Boolean variable $flag$ is 0. It serves for deciding whether processor i should give offer to i' after finishing the critical section. Variable $m1, m2$ are also temporary ones storing version numbers, whose initial values are 2. *idmaker* is a persistent variable (its value will remain after exit), whose initial value is 0. It stores the maximum *id* value among all the offers known by processor i. *eaten* is a temporary variable for counting the number of applies of the other that are received by processor i, and its initial value is 0.

3.3 Explanations

The protocol consists of 3 parts, trying section (Line 1 to Line 16), critical section (Line 17) and exit section (Line 18 to Line 51).

a) The trying section contains two parts. In the first part (Line 1 to Line 3), processor i enqueues two coupled applies for itself (the first is of *version* 1 and the second is of *version* 2). The second part (Line 4 to Line 16) is an endless loop. It will jump out of the loop whenever it can enter critical section. For each loop, it firstly dequeues an element (Line 5), and then make decisions based on the dequeued element t.

Algorithm 2. Algorithm SQ2 for processor i ($i = 1$ or 2)

```
1: for j = 1 to 2 do                                          ▷ trying section starts
2:     enqueue(APPLY, i, j)
3: end for
4: repeat
5:     t ← dequeue()                          ▷ "∗" means ignoring the field when comparing
6:     if t = (APPLY, i′, ∗) then                  ▷ i′ refers to 3 − i, i.e., the other processor
7:         flag = 1
8:     else if t = (APPLY, i, ∗) then
9:         if m1 ≠ t.version then
10:            m1 ← t.version
11:            enqueue(t)
12:        end if
13:    else if t.type = OFFER and (t.name = ALL or t.name = i) and t.id ≥ idmaker
   then
14:        break()
15:    end if
16: until false                                               ▷ trying section ends
17: Critical Section                                          ▷ critical section
18: idmaker = t.id + 1                                        ▷ exit section starts
19: if t.name = i then
20:    flag = 0
21: end if
22: repeat
23:    t = dequeue()
24:    if t = (APPLY, i′, ∗) then
25:        flag ← 1
26:    end if
27: until t = EMPTY
28: if !flag then
29:    enqueue(OFFER, ALL, idmaker)
30: else
31:    for j = 1 to 2 do
32:        enqueue(OFFER, i′, j, idmaker)
33:    end for
34:    repeat
35:        t ← dequeue()
36:        if t = (OFFER, i′, ∗, ∗) then
37:            if t.version = m2 then
38:                exit()
39:            end if
40:            m2 ← t.version
41:            enqueue(t)
42:        else if t = (OFFER,ALL, ∗) then
43:            enqueue(t)
44:            exit()
45:        else if t = (APPLY, i′, ∗) and eaten < 2 then
46:            eaten ← eaten + 1
47:        else
48:            exit()
49:        end if
50:    until false
51: end if                                                    ▷ exit section ends
```

Case 1 (Line 6): $t.type$ = APPLY and $t.name = i'$, meaning that this is the other's apply. The processor will simply assign $flag$ to 1 (Line 7), marking the fact that the other is also applying.

Case 2 (Line 8): $t.type$ = APPLY and $t.name = i$, meaning that this is its own apply. The processor will make decisions based on $t.version$ (Line 9). If $t.version = m1$, the processor will do nothing because in that case, it knows the same apply with a different $version$ number is received by the other processor. Otherwise, the processor updates $m1$ as $t.version$ (Line 10), and then enqueues t back (Line 11).

Case 3: $t.type$ = OFFER, meaning that this is an offer. If the offer is for processor i or both, and the offer is not overdue (Line 13), the processor will jump out of the loop (Line 14) to finish the trying section. Otherwise, it will do nothing.

Case 4: $t = EMPTY$. Do nothing.

b) After finishing trying section, The processor enters into critical section, where it will stay for finite time and then go to exit section.

c) The exit section is more complicated. It can be divided into two phases: Clear-up Phase (Line 18 to Line 27) and Release-offer Phase (Line 28 to Line 51).

i) In Clear-up Phase, the processor will clear up the queue, and update some local variables. On Line 18, processor i updates its $idmaker$ according to the offer, which will become the id number of the offer it soon releases. If the offer comes from the other, i.e., $t.name = i$, the processor will clear $flag$ (Line 19 to Line 21), because the applies sent by the other processor before in fact meaningless, since processor i' is able to give offer to processor i. Then processor i enters a loop (Line 22 to Line 27), dequeuing until it reads the EMPTY symbol. Meanwhile, it keeps track on whether there are applies from the other processor. If so, it will assign 1 to $flag$.

ii) In Release-offer Phase, the processor will release an offer whose name depends on $flag$ (Line 28), i.e., whether the other processor has been applied before. If the other has not applied, the processor simply enqueues an offer for all whose id is $idmaker$ (Line 29), and then exits. Otherwise, the processor will give offer to the other (Run Line 31 to Line 50). It will enqueue two coupled offers to processor i' (Line 31 to Line 33). From Line 34 to Line 50, processor i execute an endless loop until it can make sure that processor i' really receives the offer. For each loop, processor i will dequeue one element first (Line 35). And then its decision will depend on that element. We discuss it in several cases as follows.

Case 1 (Line 36): t is an offer for processor i'. In this case, if the version of the offer is the same with $m2$ (Line 37), then processor i will exit (Line 38) because it will be convinced that processor i' has already received the offer. Otherwise, it will update $m2$ (Line 40), and dequeue t back (Line 41).

Case 2 (Line 42): t is an offer for all. In this case, t is produced by processor i' after critical section. processor i will enqueue t back (Line 43), and exit (Line 44).

Case 3 (Line 45): t is an apply of i'. If processor i has read such apply for more than 2 times before, it will exit. Otherwise, it will increase *eaten* by 1, which is the number of times processor i has read $(\text{APPLY}, i', *)$.

Case 4: $t = EMPTY$. processor i will exit. Notice that it is impossible that $t.name = i$ in this loop.

4 Conclusion

In this paper, we investigate the classical mutual exclusion problem in a new setting, by which communication is made through shared queues. Without the *peek* operation, the protocol becomes quite hard to design since every processor has to modify the shared objects in order to make contact with others. We propose two novel algorithms for the problem. The designing ideas of both algorithms give new insights for the mutual exclusion problem, as well as the usage of the shared queue objects in the distributed computing setting. We leave the question of finding a protocol or proving impossibility results for more than 2 processors using one single shared queue as our further research work.

Acknowledgments. We would like to thank Wei Chen for telling us the open question concerned here. We also would like to thank Danny Dolev, Wei Chen and Lingqing Ai for helpful discussions.

References

1. Dijkstra, E.W.: Solution of a problem in concurrent programming control. CACM 8(9), 569 (1965)
2. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. CACM 17(8), 453–455 (1974)
3. Raynal, M.: Algorithms for mutual exclusion (1986)
4. Herlihy, M.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems (TOPLAS) 13(1), 124–149 (1991)