The Mailbox Problem
(Extended Abstract)

Marcos K. Aguilera[1], Eli Gafni[1,2], and Leslie Lamport[1]

[1] Microsoft Research Silicon Valley
[2] UCLA

Abstract. We propose and solve a synchronization problem called the *mailbox problem*, motivated by the interaction between devices and processor in a computer. In this problem, a postman delivers letters to the mailbox of a housewife and uses a flag to signal a non-empty mailbox. The wife must remove all letters delivered to the mailbox and should not walk to the mailbox if it is empty. We present algorithms and an impossibility result for this problem.

1 Introduction

Computers typically use interrupts to synchronize communication between a processor and I/O devices. When a device has a new request, it raises an interrupt line to get the processor's attention. The processor periodically checks if the interrupt line has been raised and, if so, it interrupts its current task and executes an interrupt handler to process unhandled device requests. The interrupt line is then cleared so that it can be used when new requests come from the device. (This is a slight simplification, since there is typically an interrupt controller between the device and processor. In this case, we consider the interrupt controller as the "device" that interrupts the processor.) It is imperative that the processor eventually execute the interrupt handler if there are unhandled requests. Furthermore, it is desirable to avoid *spurious interrupts*, in which the processor executes the interrupt handler when there is no unhandled request. A closely related problem occurs in multi-threaded programming, in which the processor and the devices are separate threads and the interrupt is some type of software signal [8, 10].

In this paper, we study a theoretical synchronization problem that arises from this setting, which we call the *mailbox problem*. From time to time, a postman (the device) places letters (requests) for a housewife (the processor) in a mailbox by the street.[3] The mailbox has a flag that the wife can see from her house. She looks at the flag from time to time and, depending on what she sees, may decide to go to the mailbox to pick up its contents, perhaps changing the position of the flag. The wife and postman can leave notes for one another at the mailbox. (The notes cannot be read from the house.) We require a protocol to ensure that (i) the wife picks up every letter placed in the mailbox and (ii) the wife never goes to the mailbox when it is empty (corresponding

[3] This problem originated long ago, when all mail was delivered by men and only women stayed at home.

to a spurious interrupt). The protocol cannot leave the wife or the postman stuck at the mailbox, regardless of what the other does. For example, if the wife and postman are both at the mailbox when the postman decides to take a nap, the wife need not remain at the mailbox until the postman wakes up. We do not require the wife to receive letters that are still in the sleeping postman's bag. However, we interpret condition (i) to require that she be able to receive mail left by the postman in previous visits to the mailbox without waiting for him to wake up.

The following simple protocol was once used in computers. The postman/device raises the flag after he delivers a letter/request; the wife/processor goes to the mailbox if the flag is raised and lowers the flag after emptying the mailbox. It is easy to see that this can cause a spurious interrupt if the postman goes to the mailbox while the flag is still raised from a previous visit and falls asleep after putting a letter in the box and before raising the flag.

There are obviously no spurious interrupts with this protocol if the postman can deliver mail to the box and raise the flag in an indivisible atomic action, while the wife can remove mail from the box and lower the flag in an indivisible atomic action. Moreover, the problem is solvable if the wife and postman can leave notes for one another, and the reading or writing of a note and the raising or lowering of the flag can be performed atomically. Here is a simple algorithm that uses a single note written by the postman and read by the wife. The postman stacks letters in delivery order in the box. After delivering his letters, the postman as a single action writes the total number of letters he has delivered so far on his note and raises the flag. When she sees the flag up, the wife as a single action lowers the flag and reads the postman's note. Then, starting from the bottom of the stack, the wife removes only enough letters so the total number she has ever removed from the box equals the value she read on the note.

What if a single atomic action can only either read or write a note or read or write a flag? Then, we show that there are no algorithms that use only two Boolean flags, one writable by the wife and one by the postman. However, perhaps surprisingly, there is a wait-free algorithm that uses two 14-valued flags, as we show. We do not know if there is an algorithm that uses smaller flags.

The mailbox problem is an instance of a general class of problems called *bounded-signaling problems*. We give a general algorithm for any problem in this class. The algorithm is non-blocking but not wait-free. It is an open problem whether there are general wait-free algorithms in this case.

The paper is organized as follows. We first define the mailbox problem in Section 2. In Section 3, we give a wait-free algorithm for the problem. To do so, we first explain the *sussus* protocol in Section 3.1. We then give a non-blocking algorithm that uses flags with large timestamps in Section 3.2. We show how to shrink these timestamps in Section 3.3. We then explain how to change the non-blocking algorithm into a wait-free algorithm in Section 3.4. In Section 4, we show that there are no non-blocking (or wait-free) algorithms that use only two Boolean flags. Next, we consider general bounded-signaling problems in Section 5. We describe related work in Section 6. Because of space limitations, most proofs are omitted from the paper.

2 Problem definition

We now state the mailbox problem more precisely. For simplicity, we let only one letter at a time be delivered to or removed from the mailbox. It is easy to turn a solution to this problem into one in which multiple letters can be delivered or removed.

We assume a *postman* process and a *wife* process. There are three operations: the postman's $deliver$ operation, the wife's $check$ operation, which returns a Boolean value, and her $remove$ operation. The postman can invoke the $deliver$ operation at any time. The wife can invoke the $remove$ operation only if the last operation she invoked was $check$ and it returned TRUE. We describe the execution of these operations in terms of the mailbox metaphor—for example "checking the flag" means executing the $check$ operation. Remember that $deliver$ and $remove$ respectively delivers and removes only a single letter.

Safety properties. We must implement $deliver$, $check$, and $remove$ so that in every system execution in which the wife follows her protocol of checking and obtaining TRUE before removing, the following safety properties hold.

If the wife and postman never execute concurrently, then the value returned by an execution of $check$ is TRUE if and only if there are more $deliver$ than $remove$ executions before this execution of $check$. This is the *sequential specification of safety*.

Neither the wife nor the postman can execute multiple operations concurrently, but the wife can execute concurrently with the postman. The allowable behaviors are specified by requiring that they act as if each operation were executed atomically at some point between its invocation and its completion—a condition known as linearizability [4].

Liveness properties. A process executes an operation by performing a sequence of atomic steps. A solution should also satisfy a liveness property stating that, under some hypothesis, a process's operation executions complete. We now state two possible liveness properties we can require of an algorithm. We number the two processes, letting the wife be process 0 and the postman be process 1. Thus, for each process number i, the other process number is $1-i$.

- *(Non-blocking)* For each i, if process i keeps taking steps when executing an operation, then either that operation execution completes or process $1-i$ completes an infinite number of operations.
- *(Wait-free)* For each i, every operation execution begun by process i completes if i keeps taking steps—even if process $1-i$ halts in the middle of an operation execution [3]. The algorithm is said to be *bounded wait-free* [3] or *loop-free* [6] if each operation completes before the process executing it has taken N steps, for some fixed constant N.

Process communication and state. A solution requires the two processes to communicate and maintain state. For that, processes have *shared variables*. We assume that there are two shared variables: $Flag$ and $Notes$. It is desirable that $Flag$ assume only a small number of values, but $Notes$ can assume infinitely many values.

Operation $check$ should be efficient: its execution should access a small amount of persistent state. We consider two alternative interpretations of this requirement:

– *(Weak access restriction)* Operation $check$ accesses at most one shared variable, $Flag$, and it only accesses this variable by reading.
– *(Strong access restriction)* Operation $check$ accesses at most one shared variable, $Flag$, it only accesses this variable by reading, and it returns a value that depends only on what it reads from $Flag$.

With the weak access restriction, $check$ can remember and use process-local state across its executions, while with the strong access restriction, $check$ is a memoryless operation that is a function of $Flag$ alone.

We are interested in solutions in which variables are atomic registers or arrays of atomic registers, and an atomic step can read or write at most one atomic register.

3 Algorithms

We now give a solution to the mailbox problem with the strong access restriction and, a fortiori, with the weak access restriction as well. It is easy to find such a solution if $Flag$ can hold an unbounded number of values. For example, we can use the algorithm mentioned in the introduction in which the postman writes his note and raises the flag in one atomic step, except having him write his note in $Flag$. We now present a solution in which $Flag$ is an array $Flag[0..1]$ with two single-writer atomic registers (a single-writer atomic register is an atomic register writable by a single process), each of which can assume only 14 values. We do not know if there is a solution that uses fewer values.

We explain our algorithm in several steps. We first present an auxiliary protocol in Section 3.1. Then, in Section 3.2, we give a solution to the mailbox problem that is non-blocking and uses flags with unbounded timestamps. In Section 3.3, we show how to bound the timestamps. Finally, we show how to make the algorithm wait-free in Section 3.4.

3.1 The *sussus* protocol

The *sussus* protocol is defined in terms of an operation $sussus(v)$ that can be invoked at most once by each process i. Intuitively, when a process i invokes $sussus(v)$ with $v = v_i$, the process tries to communicate value v_i to the other process and learn any value communicated by the other process. The operation returns an outcome and a value to process i. This value is either $\perp$ or the value v_{1-i} with which the other process invokes $sussus$. The outcome is either *success* or *unknown*. A *success* outcome indicates that process i communicates its value successfully to the other process, provided the other process invokes operation $sussus$ and completes it. An *unknown* outcome indicates that process i does not know whether it communicates its value successfully. More precisely, the protocol is bounded wait-free and satisfies the following safety properties:

– *(SU1)* If both processes complete their operation execution,[4] then at least one obtains the outcome $success$.

[4] A process may not complete the operation execution if it stops taking steps.

```
    variables A = [i ∈ 0..1 ↦ ⊥],
              B = [i ∈ 0..1 ↦ ⊥];
     (* A and B are shared arrays indexed by 0..1 with A[i] = B[i] = ⊥ for each i *)

    procedure sussus(v)                          (* output: outcome, outvalue *)
    {
s1:   A[self] := v;                              (* self is the process id: 0 or 1*)
s2:   outvalue := A[1 − self];
      if (outvalue = ⊥)
        outcome := "success";                    (* Case A *)
      else {
s3:     B[self] := "done";
s4:     if (B[1 − self] = ⊥)
            outcome := "unknown";                (* Case B *)
        else outcome := "success";               (* Case C *)
      };
s5:   return;
    };

    process (Proc ∈ 0..1)
      (* process−local variables *)
      variables outcome, outvalue;
    {
m1:   with (v ∈ Int){ call sussus(v); }
    }
```

Fig. 1. The *sussus* protocol.

- *(SU2)* For each i, if process i completes the operation execution before process $1-i$ invokes the operation, then process i obtains the outcome $success$.
- *(SU3)* For each i, if both processes complete the operation execution and process i obtains the outcome $success$, then process $1-i$ obtains the value v_i with which process i invoked the operation.

Figure 1 shows the sussus protocol, written in $^{+}$CAL [7]. Procedure $sussus$ shows the code for operation $sussus$, while the code at the bottom shows an invocation to $sussus$ with a value v chosen non-deterministically from the set Int of all integers. The outcome and value returned by operation $sussus$ are placed in variables $outcome$ and $outvalue$, respectively. Labels in $^{+}$CAL indicate the grain of atomicity: an atomic step consists of executing all code from one label to the next. In the first step of procedure $sussus$, process i sets array element $A[i]$ of shared variable A to value v. In the next step, process i reads $A[1-i]$ and stores the result in local variable $outvalue$. If the value read is $\perp$ then process i sets $outcome$ to "success". Otherwise, in a third step, process i sets $B[i]$ to "done" and, in a fourth step, it reads $B[1-i]$; if the result is $\perp$, process i sets $outcome$ to "unknown", otherwise it sets $outcome$ to "success". Observe that each atomic step accesses at most one array element of one shared variable.

To see why the protocol satisfies properties SU1–SU3, observe that there are three possibilities for the values of variables $outcome$ and $outvalue$ when a process completes its operation:

Case A: $outcome =$ "success", $outvalue = \perp$
Case B: $outcome =$ "unknown", $outvalue \neq \perp$
Case C: $outcome =$ "success", $outvalue \neq \perp$

These cases are indicated by comments in the code.

Figure 2 shows these cases as six pairs, where each pair $\langle i, \rho \rangle$ represents process i ending up in case ρ. Beneath each such pair, we indicate the outcome that process i obtains, with S standing for $success$ and U for $unknown$. Two adjacent pairs indicate the results obtained by each process in some execution. For example, we see the adjacent pairs $\langle 1, B \rangle$ and $\langle 0, C \rangle$ and the letters U and S beneath them. This indicates that, in some execution, process 1 ends up in case B with outcome $unknown$, while process 0 ends up in case C with outcome $success$. It turns out that *every* execution in which both processes complete their execution of $sussus$ corresponds to some adjacent pair in the figure. It is easy to prove this by straightforward case analysis, and even easier by model checking the $^{+}$CAL code. Properties SU1–SU3 follow easily from this fact together with the observation that v_{1-i} is the only value other than $\perp$ that process i can possibly obtain. (Remember that each process invokes operation $sussus$ at most once.)

$\langle 0, A \rangle$	$\langle 1, B \rangle$	$\langle 0, C \rangle$	$\langle 1, C \rangle$	$\langle 0, B \rangle$	$\langle 1, A \rangle$
S	U	S	S	U	S

Fig. 2. Possibilities when both processes complete execution of the *sussus* protocol.

3.2 Non-blocking algorithm with large flag values

We now present a solution to the mailbox problem that is non-blocking and uses flags that keep large, unbounded timestamps. In this algorithm, the postman and wife each keep a private counter with the number of times that they have executed $deliver$ and $remove$, respectively. To deliver or remove a letter, a process increments its counter and executes a procedure to compare its counter with the other process's counter (see procedures $deliver$ and $remove$ in Figure 3). The comparison procedure is explained in detail below. Its effect is to write to $Flag[i]$ a record with two fields, Rel and $Timestamp$. Rel is either "=" or "$\neq$", according to the result of the comparison. $Timestamp$ indicates how recent the result in Rel is; this information is used elsewhere to determine which of $Flag[0]$ or $Flag[1]$ has the most recent result.

The wife checks if the mailbox has letters or not by reading $Flag[0]$ and $Flag[1]$, choosing the flag with highest timestamp, and verifying if that flag says "=" or "$\neq$". If it says "=" then the wife considers the mailbox to be empty, otherwise, to be non-empty (see procedure $check$ in Figure 3).

```
     variables                                                    (* shared variables *)
       A = [k ∈ Int, i ∈ 0..1 ↦ ⊥],        (* A is an array indexed by the integers and 0..1 *)
       B = [k ∈ Int, i ∈ 0..1 ↦ ⊥],
       Flag=[i ∈ 0..1 ↦ [Timestamp↦0, Rel↦"="]];   (* Flag is an array of records with
                                                      fields Timestamp and Rel initialized to 1 and "=" *)
     process (proc ∈ 0..1)
       variables                                                  (* process-local variables *)
         counter = 0,                                          (* # times removed/delivered *)
         round = 0,                                            (* current round number *)
         otherc = 0,                                  (* last known counter of other process *)
         outcome,                                     (* output of procedure multisussus *)
         outvalue,                                    (* output of procedure multisussus *)
         hasmail;                                     (* output of procedure check *)
     {
m1:    while (TRUE) {
         if (self = 0) {                                          (* wife-specific code *)
m2:        call check();
m3:        if (hasmail) call remove();
         }
         else call deliver();                                     (* postman-specific code *)
       } (* while *)
     }
     procedure deliver(){
d1:    counter := counter + 1;
d2:    call compare(counter);
d3:    return;
     };
     procedure remove(){
r1:    counter := counter + 1;
r2:    call compare(counter);
r3:    return;
     };
     procedure check()                                            (* output: hasmail *)
       variables t_f0, t_f1;                                  (* procedure-local variables *)
     {
c1:    t_f0 := Flag[0];
c2:    t_f1 := Flag[1];
c3:    if (t_f0.Timestamp > t_f1.Timestamp){
         if (t_f0.Rel = "=") hasmail := FALSE;
         else hasmail := TRUE;
       } else {
         if (t_f1.Rel = "=") hasmail := FALSE;
         else hasmail := TRUE;
       };
c4:    return;
     };
```

Fig. 3. Non-blocking algorithm with large flag values (1/2). Top: shared and global variable definitions. Middle: starting code. Bottom: procedures.

```
      procedure compare(c)
      {
s1:     outcome := "unknown";
s2:     while (outcome ≠ "success") {
          (* advance round *)
s6:       round := round + 1;
s7:       call multisussus(round, c);
s8:       if (outvalue ≠ ⊥) {
            otherc := outvalue;                          (* remember outvalue *)
          };
        }; (* while *)
s9:     if (c ≠ otherc)
          Flag[self] := [Timestamp ↦ round, Rel ↦ "≠"];
        else Flag[self] := [Timestamp ↦ round, Rel ↦ "="];
s10:    return;
      };

      procedure multisussus(rnd, v)          (* output: outcome and outvalue *)
      {
ss1:    A[rnd, self] := v;
ss2:    outvalue := A[rnd, 1 − self];
ss3:    if (outvalue = ⊥)
          outcome := "success";
        else {
ss4:      B[rnd, self] := "done";
ss5:      if (B[rnd, 1 − self] = ⊥)
            outcome := "unknown";
          else outcome := "success";
        };
ss6:    return;
      };
```

Fig. 4. Non-blocking algorithm with large flag values (2/2).

In the comparison procedure, a process i executes one or more rounds numbered $1, 2, \ldots$, starting with the smallest round it has not yet executed. In each round k, process i executes an instance of the *sussus* protocol to try to communicate the value of its counter and, possibly, learn the value of the other process's counter. If the outcome of *sussus* is *success*, process i compares its counter with the most recent value that it learned from the other process. The comparison result is written to $Flag[i]$ together with timestamp k, the process's current round. The process is now done executing the $compare$ procedure. If, on the other hand, the outcome of *sussus* is *unknown* then process i proceeds to the next round $k+1$. This continues until, in some round, the outcome of *sussus* is *success*.

The detailed code for the comparison procedure is shown in Figure 4. It invokes a multi-instance version of the *sussus* protocol in procedure $multisussus$, which is a trivial extension of the code in Figure 1. Shared variable $Notes$, used in the mailbox

problem definition, is not shown in the code: for clarity, we replaced it with two shared variables, A and B. These variables should be regarded as fields $Notes.A$ and $Notes.B$ of $Notes$. Procedure $check$ writes its return value to process-local variable $hasmail$, since in $^+$CAL, a procedure call has no mechanisms for returning a value.

Intuitively, the algorithm works because the rounds provide a way to order operation executions, ensuring linearizability. Roughly speaking, we can assign each operation execution to a round, as follows:

- An execution of $remove$ or $deliver$ by a process is assigned the first round in its execution in which the other process learns the process's value or the process obtains outcome *success* from *sussus*.
- An execution of $check$ is assigned the larger of the timestamps it reads from $Flag[0]$ and $Flag[1]$.

We now order operation executions according to their assigned round number. If two operation executions are assigned the same round number, we order $deliver$ before $remove$ before $check$ operations. This ordering ensures that if some operation execution op completes before another operation execution op' starts then op is ordered before op'. For example, if an execution of $deliver$ by the postman completes in round k then a subsequent execution of $remove$ by the wife cannot be assigned to round k or smaller. This is because it is impossible for the postman to learn the wife's new value in round k or smaller since the postman already executed them.

Theorem 1. *The algorithm in Figures 3 and 4 is a non-blocking algorithm that solves the mailbox problem with the strong access restriction.*

A fortiori, the algorithm is also a non-blocking algorithm that solves the mailbox problem with the weak access restriction.

3.3 Non-blocking algorithm with small flag values

We now give an algorithm that uses flags with small values. We do so by modifying the algorithm in the previous section, which uses unbounded timestamps, to use instead timestamps that assume only 7 different values.

In the new algorithm, as in the previous one, processes execute in (asynchronous) rounds. However, in the new algorithm, the timestamp that a process uses in round k is not k; it is a value chosen dynamically at the end of round $k-1$ according to what the process sees in that round.

Let $ts_{k,i}$ be the timestamp that process i uses in round k. To understand how $ts_{k,i}$ is chosen, we consider some properties that it must have. Let us assume that the *sussus* protocol in round k returns outcome *success* for process i—otherwise $ts_{k,i}$ does not get written to $Flag[i]$ and so it is irrelevant. In the previous algorithm of Section 3.2, $ts_{k,i}{=}k$. Such a timestamp has the property that it is larger than any timestamps from previous rounds. This is too strong a property to try to satisfy with bounded timestamps. However, closer inspection reveals that it is sufficient for $ts_{k,i}$ to be larger than previous-round timestamps that could appear in $Flag[1-i]$ at the same time that $ts_{k,i}$ appears in $Flag[i]$. It turns out that there are only two such timestamps: the timestamp

variables (* **shared variables** *)
same as before except for this minor change:
$Flag$=[$i \in 0\,..\,1 \mapsto$ [$Timestamp$↦1, Rel↦"="]];

process $(proc \in 0\,..\,1)$
variables (* **process-local variables** *)
same as before, with the following additions

$ts = 1$, (* current timestamp *)
$nextts = 2$, (* next timestamp to use *)
$otherts = 1$, (* last known timestamp of other process *)

{
same as before
}

procedure $deliver()$ ***same as before***

procedure $remove()$ ***same as before***

procedure $check()$
same as before, except replace
if $(t\_f0.Timestamp > t\_f1.Timestamp)$**{**
with
if $(t\_f0.Timestamp \succ t\_f1.Timestamp)$ **{**

procedure $multisussus(rnd, v)$ ***same as before***

Fig. 5. Non-blocking algorithm with small flag values (1/2). This part is very similar to Figure 3.

already in $Flag[1-i]$ when process i ends round $k-1$, and the last timestamp learned by process i when process i ends round $k-1$. Thus, at the end of round $k-1$, process i needs to pick $ts_{k,i}$ so that it dominates these two timestamps.

Therefore, to bound the number of timestamps, we must choose them from a finite set TS with an antisymmetric total relation $\succeq$ such that, for any two elements $t_1, t_2 \in TS$, there is an element $s \in TS$ that strictly dominates both t_1 and t_2 under $\succeq$. This would be impossible if we required the relation $\succeq$ to be transitive, but we do not. A computer search reveals that the smallest set with the requisite relation $\succeq$ contains 7 elements. We take $TS = 1\,..\,7$ to be our 7-element set and define

$$
\begin{array}{lcl}
Array & \triangleq & \langle\langle 1,0,1,1,1,0,0\rangle, \\
 & & \ \langle 1,1,1,0,0,0,1\rangle, \\
 & & \ \langle 0,0,1,0,1,1,1\rangle, \\
 & & \ \langle 0,1,1,1,0,1,0\rangle, \\
 & & \ \langle 0,1,0,1,1,0,1\rangle, \\
 & & \ \langle 1,1,0,0,1,1,0\rangle, \\
 & & \ \langle 1,0,0,1,0,1,1\rangle\rangle \\
v \succeq w & \triangleq & (Array[v][w] = 1) \\
v \succ w & \triangleq & v \succeq w \wedge v \neq w \\
dominate(v, w) & \triangleq & \text{CHOOSE } x \in 1\,..\,7 : x \succ v \wedge x \succ w
\end{array}
$$

```
      procedure compare(c)
      {
 s1:    outcome := "unknown";
 s2:    while (outcome ≠ "success") {
          (* advance round *)
 s6:      round := round + 1;
*           ts := nextts;                    (* use timestamp chosen at end of last round *)
* s7:      call multisussus(round, [Timestamp ↦ ts, Count ↦ c]);
                              (* record with Timestamp and Count fields set to ts and c *)
 s8:      if (outvalue ≠ ⊥) {
*           otherts := outvalue.Timestamp;   (* remember timestamp of other process *)
*           otherc := outvalue.Count;        (* remember counter of other process *)
          };
*         nextts := dominate(otherts, Flag[1 − self].Timestamp);   (* for next round *)
        }; (* while *)
 s9:    if (c ≠ otherc)
*         Flag[self] := [Timestamp ↦ ts, Rel ↦ "≠"];          (* use ts as timestamp *)
*       else Flag[self] :=[Timestamp ↦ ts, Rel ↦ "="];
 s10:   return;
      };
```

Fig. 6. Non-blocking algorithm with small flag values (2/2). Asterisks indicate changes relative to Figure 4.

Figures 5 and 6 shows the detailed code of the algorithm sketched above. Figure 5 is very similar to Figure 3. The significant changes to the algorithm are in Figure 6, where asterisks indicate a difference relative to Figure 4.

Theorem 2. *The algorithm in Figures 5 and 6 is a non-blocking algorithm that solves the mailbox problem with the strong access restriction. It uses a $Flag$ with two 14-valued single-writer atomic registers.*

3.4 Wait-free algorithm with small flag values

The algorithms of Sections 3.2 and 3.3 are non-blocking but not wait-free, because a process completes a $deliver$ or $remove$ operation only when it obtains outcome *success* from the *sussus* protocol. Thus, if the process keeps getting outcome *unknown* in every round, the process never completes its operation. Closer examination reveals this could only happen with the wife, because of the way processes invoke operations: if the postman got stuck forever in a $deliver$ execution, the wife would execute enough $remove$ operations for the mailbox to be empty, which would cause her to stop invoking $remove$ (since she invokes $remove$ only if $check$ returns TRUE), and this would allow the postman to eventually obtain outcome *success* and complete his operation.

Therefore, the algorithm fails to be wait-free only in executions in which the postman executes infinitely many $deliver$ operations while the wife gets stuck executing

$remove$. But there is a simple mechanism for the wife to complete her operation. Because the postman's counter is monotonically increasing, if the wife knows that the postman's counter is larger than her own, she can simply complete her operation and leave her flag unchanged, since her flag already indicates that her counter is smaller than the postman's — otherwise she would not be executing $remove$ in the first place. This mechanism is shown in Figure 7 in the statement labeled "s3".

```
        max(x, y) ≜ IF x > y THEN x ELSE y

        procedure compare(c)
          variables t_round, t_otherround;
        {
  s1:     outcome := "unknown";
  s2:     while (outcome ≠ "success") {
* s3:       if (self = 0 ∧ c < otherc) return;          (* wife process *)

            (* advance or skip round *)
* s4:       t_otherround := Round[1 − self];
* s5:       t_round := max(Round[self] + 1, t_otherround − 1);
* s6:       Round[self] := t_round;

            ts := nextts;
  s7:       call multisussus(t_round, [Timestamp ↦ ts, Count ↦ c]);
  s8:       if (outvalue ≠ ⊥) {
              otherts := outvalue.Timestamp;
              otherc := outvalue.Count;
            };
            nextts := dominate(otherts, Flag[1 − self].Timestamp);
          }; (* while *)
  s9:     if (c ≠ otherc)
            Flag[self] := [Timestamp ↦ ts, Rel ↦ "≠"];
          else Flag[self] :=[Timestamp ↦ ts, Rel ↦ "="];
  s10:    return;
        };
```

Fig. 7. Wait-free algorithm with small flag values: $compare$ procedure. Asterisks indicate changes relative to the non-blocking algorithm with small flag values.

We have also included a simple optimization in which, if process i sees that its round r_i is lagging behind the other process's round r_{1-i}, then process i jumps to round $r_{1-i}-1$. The reason it is possible to jump in this case is that process i will obtain an outcome *unknown* from the *sussus* protocol in every round from r_i to $r_{1-i}-1$. In each of these rounds, the process would learn the value of the other process, but what it learns in a round is subsumed by what it learns in a higher round. Therefore, the process only needs to execute round $r_{1-i}-1$. This optimization is shown in Figure 7 in the statements labeled "s4" through "s7". It uses an additional shared array $Round[i]$ that

stores the current round of process i (this used to be in process-local variable $round$, which no longer is used), where initially $Round[i] = 0$ for $i = 0, 1$.

Theorem 3. *The algorithm in Figures 5 and 7 is a wait-free algorithm that solves the mailbox problem with the strong access restriction. It uses a $Flag$ with two 14-valued single-writer atomic registers.*

4 Impossibility

We now show that it is impossible to solve the mailbox problem when $Flag$ has only two bits, each writable by a single process. This result holds even if $Notes$ can hold unbounded values.

Theorem 4. *There is no non-blocking algorithm that solves the mailbox problem with the strong access restriction when $Flag$ is an array with two 2-valued single-writer atomic registers.*

Proof sketch. We show the result by contradiction: suppose there is such an algorithm $\mathcal{A}$. Let $Flag[0]$ and $Flag[1]$ denote the two 2-valued single-writer atomic registers. We show how to use $\mathcal{A}$ to solve consensus using only registers, which is impossible [2, 9].

If $Flag[0]$ and $Flag[1]$ are writable by the same process, it is easy to get a contradiction. Without loss of generality we can assume $Flag[0]$ is writable by the wife (process 0) and $Flag[1]$ is writable by the postman (process 1).

A *solo execution* of an operation is one where only one process takes steps (the other does nothing).

We define a function C such that $C(F_0, F_1)$ is the value returned by a solo execution of $check$ when $Flag[i] = F_i$ at the beginning of the execution. This is well-defined because (1) with the strong access restriction, operation $check$ returns a value that depends only on what it reads from $Flag$, and (2) in a solo execution of $check$, the value of $Flag$ does not change.

Assume without loss of generality that initially $Flag[0]{=}Flag[1]{=}0$.

Claim. $C(0,0){=}C(1,1){=}$FALSE *and* $C(0,1){=}C(1,0){=}$TRUE.

To show this claim, note that initially $check$ returns FALSE as no letters have been delivered. Moreover, initially $Flag[0]{=}Flag[1]{=}0$. Therefore $C(0,0) =$ FALSE.

From the initial system state, a solo execution of $deliver$ by the postman must set $Flag[1]$ to 1 (otherwise a subsequent execution of $check$ incorrectly returns $C(0,0) =$ FALSE) and we have $C(0,1) =$ TRUE.

After this solo execution of $deliver$, suppose there is a solo execution of $remove$ by the wife. This execution sets $Flag[0]$ to 1 (otherwise a subsequent execution of $check$ incorrectly returns $C(0,1) =$ TRUE) and we have $C(1,1) =$ FALSE.

After these solo executions of $deliver$ and $remove$, suppose there is a solo execution of $deliver$. Then, it sets $Flag[1]$ to 0 and we have $C(1,0) =$ TRUE. This shows the claim.

Let S be the system state after a solo execution of $deliver$ from the initial state. In state S, $Flag[0]{=}0$ and $Flag[1]{=}1$.

We now give an algorithm that we will show solves consensus for the two processes. Process i first writes its proposed value into a shared variable $V[i]$. Then, starting from state S, process 0 executes operation $remove$ of algorithm $\mathcal{A}$ and process 1 executes operation $deliver$ of $\mathcal{A}$. If process i ends up with a different value in $Flag[i]$ than when it started, then it decides on the value of $V[0]$; otherwise, it decides on the value of $V[1]$.

This algorithm solves consensus because (a) if process 0 executes by herself then $remove$ flips the value of $Flag[0]$ so the process decides on $V[0]$; (b) if process 1 executes by himself then $deliver$ leaves $Flag[1]$ unchanged so the process decides on $V[1]$; (c) if both processes execute then, after they finish, the values of $Flag[0]$ and $Flag[1]$ either both flip or both remain the same (it is not possible for only one of them to flip, because $C(0,0) = C(1,1) =$ FALSE and operation $check$ must return TRUE afterwards), and so both processes decide the same value.

This consensus algorithm uses only atomic registers and it is wait-free since $\mathcal{A}$ is non-blocking and each process invokes at most one operation of $\mathcal{A}$. This contradicts the consensus impossibility result [2, 9].

5 Bounded-signaling problems

The mailbox problem is an instance of a broader class of problems, called *bounded-signaling problems*, which we now define. In a bounded-signaling problem, each process $i = 0, 1$ has an input v_i that can vary. From time to time, a process wishes to know the value of a finite-range function $f(v_0, v_1)$ applied to the latest values of v_0 and v_1. Each input v_i could be unbounded and, when it varies, process i can access all of shared memory. However, when a process wishes to know the latest value of f, it is limited to accessing a small amount of state.

For example, in the mailbox problem, v_0 is the number of letters that the wife has removed, v_1 is the number of letters delivered by the postman, and $f(v_0, v_1)$ indicates whether $v_0 = v_1$ or $v_0 \neq v_1$. The mailbox problem places some problem-specific restrictions on how v_0 and v_1 can change. For instance, they are monotonically nondecreasing and $v_0 \leq v_1$ because if $check$ returns FALSE then the wife does not execute $remove$. Other bounded-signaling problems may not have restrictions of this type.

A precise statement of a bounded-signaling problem is the following. We are given a finite-range function $f(x, y)$, and we must implement two operations, $change(v)$ and $readf()$. If operations never execute concurrently, $readf$ must always return the value of $f(v_0, v_1)$ where v_i is the value in the last preceding invocation to $change(v)$ by process i or $v_i = \perp$ if process i never invoked $change(v)$. The concurrent specification is obtained in the usual way from this condition by requiring linearizability. Furthermore, the implementation of $readf$ must access a small amount of persistent state. We consider two alternative interpretations of this requirement:

- *(Weak access restriction)* Operation $readf$ accesses at most one shared variable, of finite range; and it accesses this variable only by reading.
- *(Strong access restriction)* Operation $readf$ accesses at most one shared variable, of finite range; it accesses this variable only by reading; and it returns a value that depends only on what it reads from the shared variable.

It turns out that the algorithm in Section 3.3 can be changed as follows to solve any bounded-signaling problem with the strong access restriction. We replace $deliver$ and $remove$ with a single procedure $change(v)$ that sets $counter$ to v, and we modify the end of procedure $compare$ to compute f with arguments c and $otherc$ (instead of just comparing c and $otherc$), and write the result and timestamp to $Flag$. The resulting algorithm is non-blocking. It is an open problem whether there exist wait-free algorithms for the general problem. Our wait-free algorithm in Section 3.4 does not solve the general problem since it relies on problem-specific restrictions on the inputs v_i.

6 Related work

The mailbox problem is a type of consumer-producer synchronization problem, with the unique feature that the consumer must determine if there are items to consume by looking only at a finite-range variable.

Work on bounded timestamping shows how to bound the timestamps used in certain algorithms (e.g., [5, 1]). That work considers a fixed-length array that holds some finite set of objects that must be ordered by timestamps. In our algorithms, it is not evident what this set should be. However, we believe some of the binary relations devised in that body of work could be used in our algorithms instead of the relation given by $Matrix$ in Section 3.3 (but this would result in much larger timestamps than the ones we use).

Acknowledgements. We are grateful to Ilya Mironov for pointing out to us that the relation of Section 3.3 should exist for sufficiently large sets, and to the anonymous reviewers for useful suggestions.

References

1. D. Dolev and N. Shavit. Bounded concurrent time-stamping. *SIAM Journal on Computing*, 26(2):418–455, Apr. 1997.
2. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985.
3. M. P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, Jan. 1991.
4. M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.
5. A. Israeli and M. Li. Bounded time-stamps. *Distributed Computing*, 6(4):205–209, July 1993.
6. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, Aug. 1974.
7. L. Lamport. The $^{+}$CAL algorithm language, July 2006. `http://research.microsoft.com/users/lamport/tla/pluscal.html`. The page can also be found by searching the Web for the 25-letter string obtained by removing the "`-`" from `uid-lamportpluscalhomepage`.
8. B. W. Lampson and D. D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 17(8):453–455, Aug. 1974.
9. M. C. Loui and H. H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.
10. J. H. Saltzer. Traffic control in a multiplexed computer system. Technical Report Project MAC Technical Report MAC-TR-30, M.I.T., June 1966.