1 Introduction

![Diagram of distributed locking protocol](image)

Figure 1: The Architecture of a Distributed Locking Protocol

![Diagram of initial and transient po cycles](image)

Figure 2: The po DAG, and also transient cycles that can form

A distributed locking protocol ("locking protocol") is our subject of study in Assignment 3. This protocol is an abstraction of an actual protocol employed in the Quarks software distributed shared memory system developed at Utah by Khandekar and Carter around 1995. We could imagine the lock being a comfortable chair on which only one person can sit at a time, and the participating processes are a collection of people who want to serially reuse the chair. Initially, process 0 possesses the chair and may, at will, sit on it zero or more times in succession. When a process sits on the chair, it sets a variable `locked` to 1; when it is not, it sets the variable to 0. Initially every other process knows who the current owner of the chair is: namely process 0. They indicate this through a variable called "probable owner" (po). Figure 1 shows the architecture of the system while Figure 2 shows the kinds of po cycles that can develop during this protocol. The initial situation with po chains is shown left-most in Figure 2, where every process has po = 0, and in this case the "probable owner" is also the actual owner. Later on, as the protocol executes, the probable owner is not going to be the actual owner. In quiescent states, the po chains form a directed acyclic graph (DAG) with the sink node being the actual lock owner, as in the middle of Figure 2. However, as shown in the right-hand side of Figure 2, we may even have transient cycles develop, and these may persist for an arbitrary number of steps, as will be very soon illustrated when we discuss the details of this protocol.

---

To model reality a bit more, each process is split into an acquire thread and a handle thread. The acquire thread is the main thread. In a perpetual loop, it does the following things:

- If the chair isn’t available, request it, and wait; when woken up, the chair is available;
- Now that the chair is available, make sure that locked is set to 1, sit on the chair and relax for a while;
- Now see if there are a queue of waiters for the chair; if not, just set locked to 0, and repeat this loop; else, let the queue of waiters be h::t where h is the head waiter and t the tail of the queue. Send the chair to h, passing along t also to it.

The handler thread is a helper thread. In a perpetual loop, it does the following things:

- Wait for an incoming message to arrive at node i’s handler.
- If po is pointing to another node j, pass the message along to j.
- If po is pointing to i, and the lock is currently free (it is 0 currently), it is an invariant that the queue of waiters at this node is empty (we will later see how this is true). Then, we simply send the lock (“the chair”) to the requesting process.
- Finally, if po is pointing to i, and the lock is not currently free (it is 1 currently), then we simply queue up the requester into the queue of waiters. It will be ensured that this queue of waiters will be later processed by proctype acquire when it eventually frees up the lock.

```
NC po==me /
locked==1
recv req(me,x)
append x to queue

W

C

NC

queue = {}
locked <= 0

po==me
send req(po,me)
recv granted(Q)
queue <= Q
locked <= 1

recv granted(Q)
queue <= h::t
send granted(t) to h
locked <= 0

po <- h

locked <= 0

queue ={}

Proctype Acquire

Proctype Handle

po==me /
locked==1
recv req(me,x)
append x to queue
send granted({}) to x

po==me /
locked==0
po <- x
recv req(me,x)
send granted({}) to x

recv req(me,x)
po <- me
send req(po,me)

po<>me
```

Figure 3: The State Machines

The finite state machines in Figure 3 describe these steps more concretely.

A few protocol scenarios will help solidify the understanding of this protocol in the reader’s minds. Consider how the cycle in Figure 2 middle can develop starting from the initial po graph shown on the left. Consider the sequence of lock acquisitions: P1 gets the lock, then P2 gets the lock, then P1 gets the lock, and finally P0 gets the lock.

- When P1 requests P0 and obtains the lock, P0’s po pointer is adjusted to point to P1, and P1 would point to itself.
- P2’s po variable would still be pointing to P0. Therefore, when P2 requests the lock, the request would initially go to P0 which would pass along the request to P1. P1 would grant the lock to P2, while updating its P0 variable to P2. When P2 receives the lock, it sets its po variable to point to itself.

- Now P1 requests the lock and obtains it back; the end result would be one where P2 points to P1 and P1 points to itself, while P0 points to P1. In this hand-over, there would be a transient graph cycle as shown in the right-hand side of Figure 2. The end result would of course be the DAG shown in the middle of Figure 2.

Using a model-checker, one can actually find the sequence of steps necessary to attain various states. These steps are called justifying steps and the sequence of states traversed during justification is called the justification sequence. Basically, to compute the justification sequence for state $S$, state $\neg S$ as a safety property, and collect the error trace. If $S$ is indeed reachable, the error trace would spell out exactly how $S$ can be reached. In another sense, we are checking whether the initial state of the system falls within the pre-image of $S$.

2 Data Structures

Figure 3 shows the acquire and handle state machines. We will now describe how these state machines act on the data structures shown in Figure 1. There is one bit locked held at every acquire process. A mutex mutex protects shared accesses between acquire and handle. Variable po maintains the probable owner information. A queue of length 1 called q.len.ch is maintained. This queue is used to convey the number of waiters that are going to be forwarded to this node. The actual waiters themselves are going to be sent separately, and they arrive into the waiters queue which is of size Nprocs - 1. This is a commonly used protocol design approach because sending advanced notification of how many are to arrive often helps prepare the receiving process for the burst of arrivals. Variable q.len records the number of waiters in the queue waiters. Finally, there is also a queue request also of size Nprocs into which incoming requests arrive.

It is sufficient for the size of the waiters and request queues to be one less than the number of processes (Nprocs), because of two (related) reasons: (i) the number of waiters at any time cannot be more than this number, and (ii) the case of a process that owns the lock and sends a request to itself does not arise (and so, allocating Nprocs locations is wasteful). Notice that this allocation is still excessive in two respects:

- Since there can be at-most Nprocs requests in the system at any time, most of the buffer locations will be unoccupied.
- Since the buffering needs grow quadratically, the overall design can be very expensive if Nprocs is large.

We shall later study changes to the protocol to arrive at a more economical implementation in terms of buffering needs. For the present protocol, however, we shall have to be content with allocating $O(Nproc^2)$ buffer locations.

3 Promela Coding

Salient aspects of the Promela code will now be discussed.
First, we begin with a few constants and type synonyms:

```c
#define Nprocs 4
#define Nprocs_1 3
#define PID byte
```

The data structures described in Section 2 are declared below:

```c
bit mutex[Nprocs];
PID po[Nprocs];
chan request[Nprocs] = [Nprocs_1] of {PID};
chan q_len_ch[Nprocs] = [1] of {byte};
chan waiters[Nprocs] = [Nprocs_1] of {PID};
byte qlen[Nprocs]; /* How many are waiting? */
bit locked[Nprocs];
```

A variable is declared to be able to state properties:

```c
byte pid_acq[Nprocs]; /* PID of acquire process */
```

The actions of acquire are now described beginning with lock acquisition. • First we take the mutex that controls accesses between acquire and handle

```c
atomic {
    mutex[me] == 0 ->
    mutex[me] = 1;
}
```

• Next, if po[me]==me, process ‘me’ can acquire the lock. Else, through request[po[me]]!me, we send a request to who ‘me’ thinks to be the probable owner. The request itself consists of me, i.e., the sending processes’s ID.
• We then release the mutex and wait for the condition

```c
q_len_ch[me] ? [count] && mutex[me]==0
```

This says that something has arrived in the q_len_ch channel. This condition is tested as the guard of an atomic and hence we are able to take mutex also in this case.

• The next section of code is very interesting:

```c
    po[me] = me;
    mutex[me] = 1;
    q_len_ch[me] ? count;
    assert(qlen[me]==0);
    qlen[me] = qlen[me]+count;
    count = 0;
};
( len(waiters[me]) == qlen[me] );
locked[me] = 1;
mutex[me]=0
```

We first set me to be the probable ownewr, thus owning the lock. Note that this is another way of recording that a lock is acquired (other than explicitly setting locked, which happens towards the end of this code fragment). In many protocols, we will find such variables that approximately track each other, and yet may subtly differ in semantics. For instance, in the present situation, we set
po[me] = me as soon as we know the count of waiters being forwarded. We will set locked[me] = 1 only after the waiters have arrived. It is possible to ‘enjoy’ the lock after the first event itself. However, in some protocols, it may make an observable difference as to when we start enjoying the lock.

- We obtain the count of the number of messages yet to arrive on the waiters queue via q_len_ch[me] ? count. We also sanity-check our code through the assert statement assert(qlen[me]==0). This means that there ought to be nothing accumulated in the waiters queue locally. This is of course correct, since proctype me just now acquired the lock, and so there was no possibility of it accumulating arriving requests in the waiters queue. In particular, the code of handler shows that if the lock is not held in the local site, then arriving requests are forwarded along the po chain. Thus, we do assert(qlen[me]==0) and then safely accumulate count via qlen[me] = qlen[me]+count. We perform count=0 as a manual ‘dead variable’ elimination step. Then, (len(waiters[me]) == qlen[me] ) is a statement that blocks till the expected number of waiters have all arrived. Finally we do locked[me]=1 and mutex[me]=0; the latter step now allows the handler thread to run.

- Releasing the lock is interesting. We acquire the mutex, and in the same step do locked[me]=0.

```
  release:
    atomic {
      mutex[me] == 0 ->
      locked[me] = 0;
      mutex[me] = 1
    };
```

- We now iterate till qlen[me] drops to 0. Notice how we set po[me] directly from the head of waiters[me]. Notice how we also send the one-diminished number of waiters in q_len_ch. The last assert uses >=; it could truly be =. Even though further requests might arrive after mutex[me] = 0, they will promptly be forwarded and not stored, because we've already updated our po[me].

```
  if
    :: qlen[me] ->
      waiters[me] ? po[me];
      qlen[me] = qlen[me]-1;
      q_len_ch[po[me]] ! qlen[me];
    do
      :: qlen[me] -> atomic {
        waiters[me] ? thread;
        waiters[po[me]] ! thread;
        thread = 0;
        qlen[me] = qlen[me]-1
      }
    :: !qlen[me] -> break
    od
  :: !qlen[me] -> skip
fi;
mutex[me] = 0;
assert (qlen[me] >= 0);
goto loop
```

- proctype handle employs the guard mutex[me] == 0 && request[me]?[requester] and acquires the mutex, and also obtains the requester via request[me]?requester. The rest of its actions are self-explanatory.
proctype handle(int me)
{
    PID requester;
    do
    :: atomic {
        mutex[me] == 0 && request[me] ? [requester]
        ->
        mutex[me] = 1;
        request[me] ? requester
    };
    if
    :: po[me] != me -> request[po[me]] ! requester
    :: po[me] == me && locked[me] ->
        waiters[me] ! requester;
    qlen[me] = qlen[me] + 1;
    assert (qlen[me] < Nprocs)
    :: po[me] == me && !locked[me] ->
        q_len_ch[requester] ! 0;
    assert (qlen[me] == 0);
    po[me] = requester
    fi;
    mutex[me] = 0
    od
}

We spawn the right number of processes:

init {
    byte cnt;

    atomic {
        do
        :: cnt == Nprocs-> break
        :: cnt != Nprocs->
            pid_acq[cnt] = run acquire(cnt);
            run handle(cnt);
            cnt = cnt+1
        od
    }
}

3.1 Verification

As far as the properties of interest go, one can state the obvious safety property that the lock can be held in exactly one place at any time. There are many more safety and liveness properties that we don’t discuss, but encourage the reader to think about. In particular, think about how transient cycles might affect the correctness of things you state. A natural fairness condition to employ would be that eventually the transient cycles will go away (a function of scheduling).

We tried verifying the following liveness property:

#define at_release_0 (acquire[pid_acq[0]]@release)
#define at_release_1 (acquire[pid_acq[1]]@release)
#define at_release_2 (acquire[pid_acq[2]]@release)
The LTL to Büchi automaton translator obtains a rather impressive looking automaton, and the property was found violated.

### 3.2 Fixing the Protocol as well as Improving it

This is what Assignment 3 consists of. It also asks you to reduce the state space without losing any information (e.g., legal executions and legal states must still be enumerated).

A few improvements and other questions will be requested in a following assignment:

- How to compute justification sequences
- How to improve the buffering needs of the protocol using *hot potato routing*
- How to improve the buffering needs using a fixed buffer allocation scheme