

## Chapter 2

# Lock Service Example

### 2.1 Introduction

This chapter introduces SESF with a simple example consisting of a lock service, a lock manager that offers the service, and collection of producers and consumers that uses the service. Figure 2.1 shows the layers of the example.

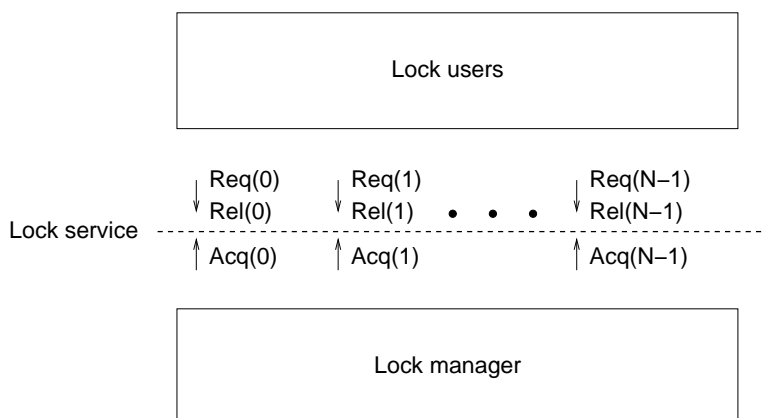


Figure 2.1: Lock service, users, and offerer.

The lock service is for  $N$  users associated with integer ids  $0, 1, \dots, N-1$ . The service defines three events for each id  $i$ :  $\text{Req}(i)$ , to request the lock,  $\text{Acq}(i)$ , to acquire the lock, and  $\text{Rel}(i)$ , to release the lock.  $\text{Req}(i)$  and  $\text{Rel}(i)$  are downward events of the service; they are called by user  $i$  and correspond to externally-controlled events of the lock manager.  $\text{Acq}(i)$  is an upward event of the service; it is called by the lock manager and corresponds to an externally-controlled event of user  $i$ . The service requires of each user that it have at most one request pending at any time, and that it does a release only if it has the lock. The service requires of the offerer, i.e., the lock manager, that at most one user acquires the lock at any time, and that every request is eventually satisfied provided a user does not hold the lock indefinitely.

We first present a service program defining the lock service. The program assumes that the joint status of all the users can be read and updated atomically. This “unreasonable” atomicity is what makes the program easy to understand. Next we present a lock manager program and establish that it offers the lock service, illustrating both operational and assertional proofs. The lock manager assumes a platform that provides weak fairness and reasonable atomicity, i.e., atomic reads and writes of boolean variables. Finally we present producer and consumer programs that use the lock service to achieve correct operation (i.e., no buffer underflow or overflow). Because the lock service isolates the users of the service from the details of the offerer (and vice versa), correct operation of the producers and consumers is assured given *any* system underneath that offers the lock service, including the particular lock manager above.

## 2.2 Lock service

The program `LockSrv(N)` in figure 2.2 defines the lock service (assignment is denoted by “:=” and equality by “=”). The initialization code, `init`, defines an array variable `z`, which indicates for each user `i`, whether the user is “thinking” (not interested in the lock), “hungry” (has requested the lock), or “eating” (has acquired the lock). The program has downward (dnw) events `Req(i)` and `Rel(i)`, and upward (upw) event `Acq(i)`. Each event has an “enabling condition” (ec), defining when the event can be executed, and an “action” (ac), defining the resulting update to `z`. The enabling condition and action are evaluated atomically. Note that `Acq(i)` reads all the entries of `z` and updates `z[i]` in one atomic step.

An execution, or evolution, of the program is a sequence of enabled event executions. If several events are enabled at the same time, any one of them can be chosen for execution. Thus the program defines a set of possible executions. In particular, `LockSrv` generates *all* executions such that (1) each user `i` cycles through request, acquire, and release, and (2) between any two acquires, there is a release corresponding to the first acquire.

The progress obligations, expressed with “leads-to” constructs, impose an additional constraint on the evolutions of the program. An evolution satisfies  $A \rightsquigarrow B$  if, whenever `A` holds at some point in the evolution then `B` holds at that point or at a later point in the evolution. Thus the progress obligations state that if every eating user eventually stops eating then every hungry user eventually eats.

```

service-program LockSrv(int N) { // lock service for N users
  init() {
    int T := 0, H := 1, E := 2; // T=thinking, H=hungry, E=eating
    int[0..N-1] z := T; // z[i] is T, H, or E; initially T.
  }

  dnw-event Req(int i) {
    ec 0 ≤ i < N and z[i]=T
    ac z[i] := H
  }

  upw-event Acq(int i) {
    ec 0 ≤ i < N and z[i]=H and [forall j, 0 ≤ j < N: z[j] ≠ E]
    ac z[i] := E
  }

  dnw-event Rel(int i) {
    ec 0 ≤ i < N and z[i]=E
    ac z[i] := T
  }

  progress-obligations {
    [forall i, 0 ≤ i < N :: z[i]=E  $\rightsquigarrow$  z[i]=T] // if every eating session eventually ends
    ⇒ [forall i, 0 ≤ i < N :: z[i]=H  $\rightsquigarrow$  z[i]=E] // then every request is eventually met
  }
} // LockSrv

```

Figure 2.2: A lock service program.

## 2.3 Lock manager

The program `LockMgr(N)` in Figure 2.3 is intended to offer `LockSrv(N)`. The initialization code defines variables `xreq` and `xacq`, and starts a process executing the function `main()`. Variable `xreq` is a boolean array

```

system-program LockMgr(int N) { // offers lock service to users 0,1,..., N-1

  init() {
    boolean[0..N-1] xreq:= false; // xreq[i] true if user i has requested the lock
    boolean xacq := false;      // true if a user has acquired the lock
    int xp := 0;                // index into xreq[0..N-1]
    StartProcess(main());
  }

  xc-event Req(int i) {
    maps LockSrv(N).Req(i)
    ec 0≤i<N
    ac xreq[i] := false;
  }

  xc-event Rel(int i) {
    maps LockSrv(N).Rel(i)
    ec 0≤i<N
    ac xacq := false;
  }

  function main() { // ⟨ ... ⟩ denotes atomic operation
    forever do {
      a1:  if ⟨xreq[xp]⟩ then {
      a2:  ⟨xreq[xp] := false⟩;
      a3:  ⟨xacq := true⟩;
      a4:  LockSrv(N).Acq(xp);
      a5:  while ⟨xacq⟩ do no-op;
      } ;
      a6:  if xp=N-1 then xp := 0 else xp := p+1
    }
  }

  progress-assumptions { weak fairness } // main() is executed with non-zero speed
}

```

Figure 2.3: A lock manager program (statement labels a1, . . . , a6 are used in analysis).

whose  $i$ th entry is true if user  $i$  has a request pending. Variable  $xacq$  is a boolean that is true if a user has the lock.

The program has two externally-controlled (xc) events,  $\text{Req}(i)$  and  $\text{Rel}(i)$ , corresponding to the  $\text{Req}(i)$  and  $\text{Rel}(i)$  events of  $\text{LockSrv}(N)$ . Each xc event has three attributes: maps, ec (enabling condition), and ac (action). The maps attribute identifies the corresponding service event. As we shall see, this is more convenient than the traditional way of having the environment directly call  $\text{LockMgr}(N).\text{Req}(i)$ . The action attribute of an xc event is executed whenever the environment calls the event. The enabling condition attribute of an xc event is *not* evaluated when the xc event is called. It is *only* for analysis; from any state satisfying the enabling condition, the action should execute atomically (without any undefined operation or infinite loop).

Function  $\text{main}()$  defines the locally-controlled activity of the program. The process executing  $\text{main}()$  cycles through  $xreq$ . When it finds that user  $xp$  is waiting, it sets  $xacq$  to true, calls  $\text{Acq}(xp)$  to grant the lock to user  $xp$ , and busy waits until  $xacq$  becomes false (which happens when the user releases the lock).

The atomicity assumptions of the program are expressed by the xc events and the angled brackets in  $\text{main}()$ . Specifically, the program assumes that the underlying platform atomically executes the xc event

actions and the statements enclosed in angled brackets. Note that the rest of the program, including `init()`, involves only variables accessed by one process. Thus it suffices if the underlying platform ensures atomic reads and writes of the shared boolean variables, i.e., `xacq`, `xreq[0]`,  $\dots$ , `xreq[N - 1]`. In future, we may just have a statement to this effect at the start of the program, rather than inserting angled brackets in the code.

Each execution, or evolution, of `LockMgr` consists of initialization followed by a sequence of event executions, of which some are `Req` and `Rel` executions and the others are steps of `main()`. Many evolutions are possible because the `Req` and `Rel` events can occur at arbitrary times. For `LockMgr(N)` to offer `LockSrv(N)`, every evolution must be consistent with the service. We make this precise next.

## 2.4 Establishing service satisfaction

We say an evolution of `LockMgr(N)` is safe wrt `LockSrv(N)` if the sequence of `Req`, `Rel` and `Acq` events in the execution can be generated by `LockSrv(N)`. We say an event call is safe if extending the evolution thus far by the event call results in an evolution that is safe wrt `LockSrv(N)`.

`LockMgr(N)` offers `LockSrv(N)` if it satisfies the following:

- **Fault freedom:** `LockMgr(N)` is fault-free *in isolation* (i.e., does no undefined operation assuming that its xc events are called only when enabled).
- **Input safety:** Whenever `Req` or `Rel` can be called safely (wrt the service), the enabling condition (in `LockMgr(N)`) holds.
- **Output safety:** `LockMgr(N)` makes only safe calls of `Acq` assuming that the environment's calls of `Req` and `Rel` are safe.
- **Progress:** `LockMgr(N)` extends every safe evolution to one that satisfies `LockSrv(N)`'s progress obligations provided `main()` is executed with non-zero speed.

The first two conditions above are a bit stronger than needed. It suffices to require that `LockMgr(N)` be fault-free assuming that the environment's calls of `Req(i)` and `Rel(i)` are safe. But the formulation given above is better because it breaks up the analysis into two parts, one dealing with the lock manager in isolation and another dealing with its obligations concerning inputs.

We next establish that the above four conditions hold.

**Proof of fault freedom.** The only undefined operations possible in `LockMgr(N)` are out-of-bound references to `xreq`, either in `Req(i)`'s action or in `main()`. The former is not possible because of `Req(i)`'s enabling condition. The latter is not possible because `xp` is initially in  $0..N - 1$  and is changed only by `a6`, which keeps `xp` in this range.

**Proof of input safety.** `LockSrv(N)` requires that a safe `Req(i)` call must have `i` in  $0..N - 1$ , which implies `Req(i)`'s enabling condition. The same holds for `Rel(i)`.

**Proof of output safety.** Let `x` denote *any* finite evolution of `LockMgr` such that `x` is safe wrt `LockSrv(N)` and the process executing `main()` is at `a4` at the end of `x`. We need to show that at the end of `x` the following hold: (1) user `xp` is hungry, and (2) no process has the lock.

The first property holds because the process comes to `a4` only after executing `a1` and finding `xreq[xp]` true; user `xp` was hungry at that time and remains so because steps `a2` and `a3` do not make any user eating.

Regarding the second property, we establish the following stronger property: *no user has the lock when control is not at a5*. This holds initially. We proceed by induction on the number of executions of `a5` in `x`. Let `x` have an `Acq(xp)` call execution at time  $t_0$ . Because the induction hypothesis holds prior to  $t_0$ , just after the call, user `p` is the only user to have the lock. Also, `xacq` is true just after the call because `a3` was executed at some time  $t_1$  ( $< t_0$ ) and during  $t_1$  to  $t_0$  no user has the lock and so could not have set `xacq` to false (without doing an unsafe `Rel`). So after  $t_0$ , control is stuck on `a5` until user `p` sets `xacq` to false (again, no other user can do this without doing an unsafe `Rel`). So until user `p` releases the lock, control is stuck on `a5` and no other user can get the lock. So when user `p` releases the lock, no user has the lock. So when control gets past `a5`, no user has the lock and the induction step is sustained.

**Proof of progress condition.** We first show that control cannot get stuck on `a5`. Let control enter `a5` at some point in `x`. From the output safety argument above, we have that user `p` has the lock. Because every eating session eventually ends, it eventually gives up the lock and falsifies `xacq`. Once this happens, `xacq` stays false (because only `main()` can set `xacq` to true), and hence control eventually exits `a5` (from non-zero process speed).

Because `a5` is the only potentially unbounded statement in `main()`, we have that `p` keeps increasing modulo-`N`. Hence for every waiting user `j`, control eventually comes to `a1` with `xp = j`. From there, it enters `a2` and user `j` eventually gets the lock. This establishes the progress condition, thus completing the proof that `LockMgr(N)` offers `LockSrv(N)`.

## 2.5 Program-based conditions

In the preceding analysis, the safety and progress conditions were expressed in terms of the set of evolutions of the lock manager and the set of evolutions of the lock service, and the conditions were proved by operational arguments. We now express the safety and progress conditions in terms of the programs of the lock manager and lock service, specifically, as assertions to be satisfied by the composite system of `LockMgr(N)` and a modified `LockSrv(N)`. The conditions can be established by operational or assertional proofs, and we illustrate both.

```

system-program LockSrv'(int N) { // system version of LockSrv(N)

  init() {
    int T := 0, H := 1, E := 2; // T=thinking, H=hungry, E=eating
    int[0..N-1] z := T; // z[i] is T, H, or E; initially T.
  }

  lc-event Req(int i) {
    ec (0 ≤ i < N) and (z[i]=T)
    ac z[i] := H; LockMgr(N).Req(i)
  }

  xc-event Acq(int i) {
    ec 0 ≤ i < N and z[i]=H and [forall j, 0 ≤ j < N: z[j] ≠ E]
    ac z[i] := E
  }

  xc-event Rel(int i) {
    ec (0 ≤ i < N) and (z[i]=E)
    ac z[i] := T; LockMgr(N).Rel(i)
  }

  no progress obligations
} // LockSrv'(N)

```

Figure 2.4: `LockSrv(N)` modified (changes in boxes) to interact with `LockMgr(N)`.

The first step is to change `LockSrv(N)` to a system program that models the environment that `LockMgr(N)` expects. The resulting program, called `LockSrv'(N)`, is shown in figure 2.4. Each upw event (i.e., `Acq(i)`) has been changed to an externally-controlled event (which is executed when `LockMgr(N)` calls it). Each dnw event (i.e., `Req(i)` and `Rel(i)`) has been changed to a locally-controlled event and its action augmented with a call to the corresponding event of `LockMgr`; the event's action is executed only if the enabling condition holds.

Let  $X$  denote the composite system of  $\text{LockMgr}(N)$  and  $\text{LockSrv}'(N)$ . System  $X$  has two interacting processes, one executing  $\text{LockMgr}(N)$ 's  $\text{main}()$  and another executing the  $\text{Req}(i)$  and  $\text{Rel}(i)$  events of  $\text{LockSrv}'(N)$ . Furthermore, let  $X$  be closed, that is,  $X$ 's environment does not call any  $\text{xc}$  events within  $X$ . Intuitively,  $X$  executes  $\text{LockMgr}(N)$  in an environment of  $N$  correctly functioning users. Given this, it should be reasonably obvious that the conditions for  $\text{LockMgr}$  to offer  $\text{LockSrv}$  can be rephrased as follows:

- Fault freedom:  $\text{LockMgr}(N)$  is fault-free in isolation.
- Input safety:  $X$  satisfies the following:
  - $\square \text{LockSrv}'(N).\text{Req}(i).\text{ec} \Rightarrow \text{LockMgr}(N).\text{Req}(i).\text{ec}$
  - $\square \text{LockSrv}'(N).\text{Rel}(i).\text{ec} \Rightarrow \text{LockMgr}(N).\text{Rel}(i).\text{ec}$

(That is, in every execution of  $X$ , the  $\text{Req}(i)$  and  $\text{Rel}(i)$  of  $\text{LockMgr}(N)$  are called only when enabled. Note that the predicates relate variables of  $\text{LockMgr}(N)$  and  $\text{LockSrv}'(N)$ .)

- Output safety:  $X$  satisfies the following:
  - $\square \text{LockMgr}(N) \text{ on } a4 \Rightarrow \text{LockSrv}'(N).\text{Acq}(\text{LockMgr}(N).\text{xp}).\text{ec}$
- Progress:  $X$  satisfies  $\text{LockSrv}(N)$ 's progress obligations.

As in the previous formulation, the first three conditions are a bit stronger than needed. They could be replaced by the condition that  $X$  be fault-free. We prefer to keep the three conditions for the same reason as before.

## 2.6 Establishing the program-based conditions

The next step is to establish that  $X$  satisfies the above assertions. We start by restating the assertions in more detail. For any statement  $a_j$  in  $\text{main}()$ , we say “on  $a_j$ ” to mean that the process executing  $\text{main}()$  is at or within statement  $a_j$ ; note that it can be within only if statement  $a_j$  is not atomic. For brevity, and without incurring ambiguity, we omit the prefix  $\text{LockMgr}(N)$  or  $\text{LockSrv}'(N)$  when referring to variables of  $X$ .

The safety condition is that  $X$  should satisfy  $\square S_1 \wedge S_2 \wedge S_3$ , where

$$S_1: (\text{on } a4) \Rightarrow ((0 \leq \text{xp} < N) \text{ and } (z[\text{p}]=\text{H}) \text{ and } [\text{forall } j, 0 \leq j < N :: z[j] \neq E]) \quad (\text{for } \text{Acq}(i))$$

$$S_2: ((0 \leq i < N) \text{ and } z[i] = \text{T}) \Rightarrow (0 \leq i < N) \quad (\text{for } \text{Req}(i))$$

$$S_3: ((0 \leq i < N) \text{ and } z[i] = \text{E}) \Rightarrow (0 \leq i < N) \quad (\text{for } \text{Rel}(i))$$

The progress condition is that  $X$  should satisfy

$$P_1: [\text{forall } i, 0 \leq i < N :: z[i]=\text{E} \rightsquigarrow z[i]=\text{T}] \Rightarrow [\text{forall } i, 0 \leq i < N :: z[i]=\text{H} \rightsquigarrow z[i]=\text{E}]$$

We can establish that  $X$  satisfies these assertions by an operational proof or by an assertional proof. The former would essentially be a repetition of the proof given in the previous section. Here, we give an assertional proof. We come up with a sequence of assertions leading up to the above assertions, such that each assertion follows from previous assertions and the program  $X$  by applications of proof rules.

Here are the proof rules we will be using.

- Invariance rule:  $X$  satisfies  $\square B$  if:
  - (1)  $B$  holds initially (i.e., after executing  $\text{LockMgr}(N).\text{init}()$  and  $\text{LockSrv}'(N).\text{init}()$ ).
  - (2) For every (lc or xc) event  $e$  of  $X$ , executing  $e.\text{ac}$  in any state satisfying  $B$  and  $e.\text{ec}$  results in a state that satisfies  $B$ .
- Leads-to via  $\text{LockMgr}$  rule:  $X$  satisfies  $P \rightsquigarrow Q$  if:

- (1) Starting from any state satisfying  $P$ , the execution of  $\text{main}()$  establishes  $Q$  assuming nothing else executes (i.e., no  $\text{Req}$  or  $\text{Rel}$  executions).
  - (2) The execution of any other event in  $X$  preserves  $P$  or establishes  $Q$ .
- Closure rules, for establishing assertions from other assertions. Examples include:
    - $\Box P$  holds if  $\Box Q$  and  $Q \Rightarrow P$  hold.
    - $P \rightsquigarrow Q$  holds if  $P \rightsquigarrow (Q \text{ or } R)$  and  $R \rightsquigarrow Q$  hold.

**Proof of  $\Box S_2$  and  $\Box S_3$**  Trivial. In each of  $S_2$  and  $S_3$ , the predicate's antecedent implies its consequent.

**Proof of  $\Box S_1$**  Denote the conjunction of the predicates  $B_1, \dots, B_5$  below by  $B_{1..5}$ .  $B_{1..5}$  satisfies the invariance rule (check the details), and so  $\Box B_{1..5}$  holds.  $B_{1..5}$  implies  $S_1$ , and so  $\Box S_1$  holds.

$$B_1 : (\text{not on a5}) \Rightarrow [\text{forall } j, 0 \leq j < N :: z[j] \neq E]$$

$$B_2 : ((\text{on a5}) \text{ and } \text{xacq}) \Rightarrow (z[\text{xp}] = E \text{ and } [\text{forall } j, 0 \leq j < N, j \neq \text{xp} :: z[j] \neq E])$$

$$B_3 : ((\text{on a5}) \text{ and } (\text{not xacq})) \Rightarrow [\text{forall } j, 0 \leq j < N :: z[j] \neq E]$$

$$B_4 : (\text{on a4}) \Rightarrow \text{xacq}$$

$$B_5 : z[i] = H \Leftrightarrow (\text{xreq}[i] \text{ or } (\text{on a3..a4}))$$

Note that one can *check* each proof rule application, and hence the entire proof, without having any global understanding of how the system works. All that is needed is to understand the individual constructs of the program and assertions used in the proof rule application. Of course, coming up with the assertions requires global understanding.

**Proof of  $P_1$**  The first line below is an assumption. Each remaining line below states an assertion (at the left) and a proof rule application (at right) that establishes the assertion.  $P_1$  follows from  $C_1$  and  $C_8$  below.

$$C_1 : z[k] = E \rightsquigarrow z[k] = T \quad [\text{assumption (lhs of } P_1)]$$

$$C_2 : (\text{on a1 and xp} = j \text{ and xreq}[j]) \rightsquigarrow z[j] = E \quad [\text{via LockMgr, closure}]$$

$$C_3 : (\text{on a5 and xp} = k \text{ and xacq}) \rightsquigarrow (\text{on a5 and xp} = k \text{ and } \neg \text{xacq}) \quad [\text{finite eating, invariance } B_2, \text{ closure}]$$

$$C_4 : (\text{on a5 and xp} = k \text{ and } \neg \text{xacq}) \text{ leadsto } (\text{on a6 and xp} = k) \quad [\text{via LockMgr, invariance } B_2, \text{ closure}]$$

$$C_5 : (\text{on a1 and xp} = j) \rightsquigarrow ((\text{on a1 and xp} = j \text{ and xreq}[j]) \text{ or } (\text{on a6 and xp} = k)) \quad [\text{via LockMgr, closure}]$$

$$C_6 : (\text{on a1 and xp} = j) \rightsquigarrow (\text{on a1 and } (\text{xp} = (j+1) \bmod N)) \quad [C_2, \dots, C_5, \text{ closure}]$$

$$C_7 : z[k] = H \rightsquigarrow (z[k] = H \text{ and } \text{on a1 and xp} = k) \quad [C_6, \text{ closure}]$$

$$C_8 : z[k] = H \rightsquigarrow z[k] = E \quad [C_7, C_6, \text{ closure}]$$

## 2.7 Producer-consumer users

So far we have specified a lock service and a lock manager system that offers the lock service. Figure 2.5 shows a producer-consumer system that uses the lock service for two users, i.e.,  $\text{LockSrv}(2)$ , to achieve correct operation. The system consists of a producer process and a consumer process that share an integer variable *balance*. The producer repeatedly adds to it. The consumer repeatedly empties it out. The producer-consumer system assumes that the underlying platform provides weak fairness for the consumer (but not necessarily the producer), and atomicity of boolean reads and writes. So the producer and consumer use the lock service to ensure that their operations on *balance* are atomic.

Just as we established that `LockMgr(N)` offers `LockSrv(N)`, we have to establish that `ProducerConsumer()` correctly uses `LockSrv(2)`, i.e., it does not make unsafe calls of `Req(i)` or `Rel(i)` and it is fault-free assuming `LockSrv(2)` is correctly offered. We formalize this just like before. Let `LockSrv'(2)` be `LockSrv(2)` modified to represent a correct service offerer; change the upw event `Acq(i)` to an lc event and the dnw events `Req(i)` and `Rel(i)` to xc events. Let `Y` denote the composite system of `ProducerConsumer` and `LockSrv'(2)`. `ProducerConsumer` uses `LockSrv(2)` if the following hold:

- Fault freedom: `ProducerConsumer()` is fault-free in isolation.
- Input safety: `Y` satisfies the following:
  - $\square \text{LockSrv}'(2).\text{Acq}(i).\text{ec} \Rightarrow \text{ProducerConsumer}.\text{Acq}(i).\text{ec}$
- Output safety: `Y` satisfies the following:
  - $\square \text{ on LockSrv}(2).\text{Req}(i) \Rightarrow \text{LockSrv}'(2).\text{Req}(i).\text{ec}$
  - $\square \text{ on LockSrv}(2).\text{Rel}(i) \Rightarrow \text{LockSrv}'(2).\text{Rel}(i).\text{ec}$
- Progress: none.

It is trivial to see that the above conditions hold. Given that, we can now analyze the producer-consumer system using the lock service, simply by analyzing `Y`. For example, we can show that `Y` satisfies the following assertions:

- $\square \text{cn} \leq \text{pn}$  (no buffer underflow)
- $\text{pn} \geq k \rightsquigarrow \text{cn} \geq k$  (every item produced is eventually consumed)

Note that neither the producer-consumer system nor our analysis is concerned with how `LockSrv(2)` is implemented, i.e., with the internals of the system offering `LockSrv(2)`. That system could be the `LockMgr(2)` described earlier, or some other system. All that is needed is that it correctly offers `LockSrv(2)`.

## 2.8 Concluding remarks

TBD

## 2.9 Exercises

- 2.1 Specify a lock service in which the offerer can take away the lock from a user. In particular, introduce the “forced release” upw event `FRel(i)` in addition to `Req(i)`, `Rel(i)`, and `Acq(i)`, and make all the modifications you think necessary. [Such a lock service would be relevant for database transactions: a transaction updates a local copy of the data only after it gets the lock; at some point, the transaction either commits (updates the master copy and releases (`Rel`) the lock) or aborts (is forced (`FRel`) to release the lock and the master copy is not updated).]
- 2.2 Specify a lock service in which the user that has acquired the lock keeps the lock until another user requests it. In particular, introduce the “request indication” upw event `ReqInd(i)`, informing user `i` that another user wants the lock, and make all the modifications you think necessary. Note this is not a forced release: if `ReqInd(i)` is called when user `i` needs the lock, the user releases the lock only after it no longer needs it.
- 2.3 Prove that the `ProducerConsumer()` system using `LockSrv(2)` satisfies the properties given above.
- 2.4 Obtain a lock manager that offers the lock service in problem 2.1.
- 2.5 Obtain a lock manager that offers the lock service in problem 2.2.

```

system-program ProducerConsumer() { // uses LockSrv(2)
  atomicity-assumption{ reads and writes of shared boolean variables}

  init() {
    integer balance := 0; // balance shared between producer and consumer
    boolean wait[0..1]; // wait[i] true if user i waiting for the lock
    StartProcess(Producer()); // user 0 of lock service
    StartProcess(Consumer()); // user 1 of lock service
  }

  xc-event Acq(int i) {
    maps LockSrv(2).Acq(i)
    ec  $0 \leq i < 2$ 
    ac wait[i] := false;
  }

  function Producer() {
    integer pn := 0; // local variable
    forever do {
      pn := pn + 1;
      wait[0] := true;
      LockSrv(2).Req(0); // request lock
      while wait[0] do no-op;
      balance := balance + 1 ;
      LockSrv(2).Rel(0); // release lock
    }
  }

  function Consumer() { // executed by user 1
    integer cn := 0; // local variable
    forever do {
      wait[1] := true;
      LockSrv(2).Req(1); // request lock
      while wait[1] do no-op;
      if balance > 0 {
        cn := cn + balance;
        balance := 0 ;
      };
      LockSrv(2).Rel(1); // release lock
    }
  }

  progress-assumptions { weak fairness of Consumer()}
}

```

Figure 2.5: A lock user program.