Chapter 2
Simple Lock

2.1 Introduction

This chapter fleshes out a simplified version of the producer-consumer-lock example outlined in the previous chapter. Section 2.2 presents a "simple lock" program that implements a lock assuming a platform with minimal atomicity, namely, atomic reads and writes of memory words, and minimal progress, namely, weak fairness of threads. Section 2.3 presents a service program that defines the desired external behavior of the simple lock program. Section 2.4 states the conditions for the simple lock to implement the simple lock service. Section 2.5 proves that the conditions hold. This section also illustrates assertional proofs. Section 2.6 presents a program in which a producer and a consumer make use of the simple lock service.

To keep things simple, the lock users are restricted to have thread ids `0`, `1`, $\cdots$, `N-1`, where `N` is a parameter of the program. This allows the lock program to track its users with an array. (Tids of general structure complicates the handling of simultaneous lock requests.) Also for simplicity, the simple lock program has a local thread that continuously checks for user requests. (Implementations without local threads are presented in Chaps. 9 and 10.)

2.1.1 Conventions

Assignment is denoted by "$\leftarrow$" and equality by "$=$". Conjunction is denoted by "`and`", disjunction by "`or`", and implication by "$\Rightarrow$". These operators are "short-circuit" evaluated from left to right; e.g., when evaluating "`A and B`", first `A` is evaluated and then `B` is evaluated iff `A` is true. For an implication "`A` $\Rightarrow$ `B`", we refer to `A` as the "lhs" (left-hand side) and to `B` as the "rhs" (right-hand side). Likewise, a leads-to assertion "`A` *leads-to* `B`" has lhs `A` and rhs `B`.

A.U. Shankar, *Distributed Programming: Theory and Practice*,
DOI 10.1007/978-1-4614-4881-5_2,

The construct `j..k`, where `j` and `k` are integers, is the sequence of integers `j`, `j+1`, ⋯, `k`; it is empty if `j` exceeds `k`. The construct `mod(j,N)` stands for `j` modulo-`N`, the non-negative remainder of `j` divided by `N`.

The type `Sid` denotes sids (system ids); they are pointers to systems. A non-null sid `z` is said to be "alive" iff it points to a system; i.e., a system with sid `z` has been created and not yet terminated. The construct `sid()` returns a non-null sid that is not alive; a system can be created later with that sid. The construct `sid(z)`, where `z` is an integer or character string, returns a non-null non-alive sid with value `z` if sid `z` is not currently allocated; otherwise it returns `null`.

The type `Tid` denotes tids (thread ids); they are pointers to threads. A non-null tid `z` is said to be "alive" iff it points to a thread. The constructs `tid()` and `tid(z)` are analogous to the corresponding constructs for sids.

The following predicates are available for stating properties of programs. The predicate "`thread t in S`", where `S` is a statement or code chunk, is true iff thread (with tid) `t` is inside `S`. The predicate "`thread in S`" is true iff some thread is in `S`. The predicate "`not thread in S`" is true iff there is no thread in `S`. In these predicates, "`in`" can be replaced by "`at`" to mean that the thread is at the start of `S`, and by "`on`" to mean that the thread is at or in `S`.

For a non-null sid (or tid) `z`, the predicate "`z.alive`" is true iff `z` is alive. For notational convenience, we assume that the state of a system `z` is available for writing assertions even after `z` has been terminated; this allows us, for example, to shorten "`z.alive and (thread in z.f)`" to "`thread in z.f`".

Given an atomic step e and predicates P and Q, we say e **unconditionally establishes** Q **from** P to mean that Q holds after an execution of e assuming *only* that P and any input assumption of e hold prior to the execution. If P is vacuous (i.e., `true`), we say e **unconditionally establishes** Q. If P and Q are the same, we say e **unconditionally preserves** P.

End of conventions

2.2 Lock Program

The simple lock program is shown in Fig. 2.1 (ignore the "•"s for now). The program, called `SimpleLock`, has one parameter, `N`, indicating the number of possible users; their tids would be `0`, …, `N-1`. The program input assumption requires `N` to be at least `1`. The main code first defines some variables. Boolean array `xreq` has its `i`th entry true if user `i` has a request pending. Boolean `xacq` is true if a user has the lock. Integer `xp` is an index into `xreq`. The main code then starts a thread to execute function `serve` and stores its tid in variable `t`. The main code ends by returning its sid. (Tid `t` may coincidentally be in `0..N-1`, but it doesn't matter.)

In function `serve`, thread `t` repeatedly cycles through `xreq`. When it finds that the user corresponding to `xp` is waiting, it sets `xacq` to true, sets `xreq[xp]` to false (freeing thread `xp` to return from its `acq` call), and busy waits until `xacq` becomes false (which happens when the thread releases the lock).

```
program SimpleLock(int N)  {     // Implements lock for users 0, ... , N-1
   ia {N ≥ 1}                    // program input assump; for analysis only
   boolean[N] xreq ← false;      // xreq[i] true if user i has requested lock
   boolean xacq ← false;         // true if a user has acquired the lock
   int xp ← 0;                   // index into xreq
   Tid t ← startThread(serve()); // start thread t executing function serve()
   return mysid;
   // end main

   function void serve() {
     while (true) {
  a0:  if  • (xreq[xp]) {
  a1:   • xacq ← true;
  a2:   • xreq[xp] ← false;
  a3:     while  • (xacq) skip;
        }
  a4:  xp ← mod(xp+1,N);
     }
   }

   input void mysid.acq() {
      ia {mytid in 0..N-1}   // for analysis only
  a5: xreq[mytid] ← true;
  a6: while  • (xreq[mytid]) skip;
      return;
   }

   input void mysid.rel() {
      ia {mytid in 0..N-1}       // for analysis only
  a7: xacq ← false;
      return;
   }

   input void mysid.end() {
      ia {true}                   // for analysis only
      endSystem();
   }

   atomicity assumption {                            // for analysis only
      reads and writes of xacq, xreq[0], ..., xreq[N-1]
   }

   progress assumption {weak fairness for threads}  // for analysis only
}
```

Fig. 2.1 Simple lock program; statement labels `a0`, ..., `a7` are used in analysis; "•"s are atomicity breakpoints

There are three input functions: acq and rel to acquire and release the lock; and end to initiate termination of the lock system. In acq and rel, the input assumption requires the calling thread's tid, mytid, to be in 0..N–1. When thread i calls acq, it sets xreq[i] to true, busy waits until xreq[i] becomes false (due to thread t executing a2 with xp equal to i), and returns. When thread i calls rel, it sets xacq to false (which frees thread t from waiting on a3) and returns. Function end can be called by any thread (its tid need not be in 0..N–1). It executes endSystem, initiating termination of the lock system. Termination occurs if and when the system has no guest threads. So it may may never occur, or it may occur when the lock is with a user.

The atomicity assumption requires the underlying platform to atomically execute reads and writes of the boolean variables xacq, xreq[0], ···, xreq[N–1]. These variables are subject to simultaneous conflicting accesses by the local thread t and the guest threads 0, ···, N–1 (executing input functions). Specifically, variable xacq is accessed by thread t and any thread calling rel, and variable xreq[i] is accessed by threads i and t.

The progress assumption requires that thread t and the guest threads are run with weak fairness by the underlying platform, i.e., no thread enters a state where it gets no further processor cycles. The atomicity and progress assumptions are very conservative, and would be provided by practically any bare hardware platform.

Note that mysid.acq's input assumption allows the calling thread to already have the lock. Similarly, mysid.rel's input assumption allows the calling thread to not have the lock. These requirements, which will be imposed later in the lock service, are omitted here *purely to illustrate* that an implementation can assume that users obey the service. While these particular requirements can be easily checked in the input function bodies, in general, the check may be expensive. They would also be redundant if the callers can be trusted. We could have even set the input assumptions to simply true, but then the program would not satisfy the fault-freedom and effective-atomicity proved below.

Each evolution of SimpleLock is a sequence of transitions, each transition being an execution of a statement in the main code or in a function of the program. Each transition takes the program from one state to another, starting from the initial state defined by a value for N. Many evolutions are possible because requests and releases can occur at arbitrary times.

2.2.1 *Fault-Freedom and Effective Atomicity*

We now prove two properties of program SimpleLock, that is, properties satisfied by every *allowed* evolution (i.e., assuming inputs satisfy their input assumptions). The first property is that program SimpleLock is fault-free. There only two ways that the program can become faulty. The first way is if xreq is defined with negative N; this is ruled out by the program's input assumption. The second way is an out-of-bound reference to xreq in functions mysid.acq or serve. The former is not possible because

of mysid.acq's input assumption. The latter is not possible because xp is initially in 0..N−1 and is changed only by a4, which keeps xp in this range. Thus the following holds.

A_0 : SimpleLock is fault-free

It's worth noting that SimpleLock has allowed evolutions that are undesirable. For example, if a user calls rel when it does not have the lock, two users may end up having the lock at the same time (how?).

The second property is a partition of its code into effectively-atomic steps. We denote the partition by selecting a subset of control points. The selected control points, which we refer to as **atomicity breakpoints**, are indicated in Fig. 2.1 by the "•"s. *The sequence of instructions that a thread executes from one atomicity breakpoint until the next atomicity breakpoint is effectively atomic*. In particular, each such sequence has at most one atomic instruction that conflicts with code executed by other threads. Details follow:

- Initial step: The main code (which has no internal atomicity breakpoints) together with function serve upto its first atomicity breakpoint (at a0) is effectively atomic. It conflicts only in return mysid (after which input functions can be called).
- Function serve: Each of a0, a1, a2, and a3's iteration is atomic (from the program's atomicity assumption). Each conflicts with code of other threads, so no two of them can be grouped together. Statement a4 does not conflict, so it can be grouped with the preceding atomic step; i.e., a4 can be executed atomically with a0 when xreq[xp] is false and with a3 when xacq is false. (Thread t may be terminated, along with the system, after partially executing a4, but this is equivalent to not having started executing a4.)
- Function mysid.acq: From the start of the function to the atomicity breakpoint at a6 is effectively atomic; its only conflict is in the atomic write in a5. From the atomicity breakpoint at a6 to the next atomicity breakpoint at a6 (or to the return) is effectively atomic; its only conflict is the atomic read in a6.
- Function mysid.rel: The entire body is effectively atomic; its only conflict is the atomic write in a7.
- Function mysid.end: The entire body is effectively atomic.

2.3 Lock Service Program

The lock service program, called SimpleLockService, is shown in Fig. 2.2. The program parameters and program input condition are as in SimpleLock. It has three input functions, acq, rel and end, and no output functions. The main code defines an array acqd that indicates the user (if any) that has acquired the lock, and a boolean ending that indicates whether end has been called. The program is fault-free. Faulty initialization of acqd is precluded by the program input assumption. Out-of-

```
service SimpleLockService(int N)  { // lock service for users 0, ..., N-1
   ic {N ≥ 1}
   boolean[N] acqd ← false;     // acqd[i] is true iff i has lock
   ending ← false;              // termination initiated
   return mysid;
   // end main
   }

   input void mysid.acq() {
      ic {not ending and (mytid in 0..N-1)
          and not acqd[mytid]}
      oc {forall(j in 0..N-1: not acqd[j])}
      acqd[mytid] ← true;
      return;
   }

   input void mysid.rel() {
      ic {not ending and (mytid in 0..N-1)
          and acqd[mytid]}
      acqd[mytid] ← false;
      oc {true}
      return;
   }

   input void mysid.end() {
      ic {not ending}
      ending ← true;
      oc {true}
      return;
   }

   atomicity assumption {input parts and output parts}

   progress assumption {
     // rel call returns
     forall(Tid i: (thread i in mysid.rel) leads-to (not i in mysid.rel));

     // if no one holds the lock forever then acq call returns
     forall(i in 0..N-1: acqd[i] leads-to not acqd[i])
       ⇒ forall(Tid i: (thread i in mysid.acq) leads-to (not i in mysid.rel));

     // end call returns
     forall(Tid i: (thread i in mysid.end) leads-to (not i in mysid.end));
   }
}
```

Fig. 2.2 Simple lock service program

bound reference to acqd is precluded by the conjunct "mysid in 0..N–1" in the input conditions of acq and rel. (A faulty service program is useless as a standard for implementation.)

A_1 : SimpleLockService is fault-free

The input condition of function acq is more constraining than in the simple lock. In addition to requiring the calling thread's tid to be in 0..N–1, it requires that end has not been called and that the calling thread does not have the lock. The output condition of acq requires that the return be executed only if no user has the lock. The input condition of function rel is more constraining than in the simple lock, requiring also that end has not been called and that the calling thread have the lock. Its output condition is vacuous, so it is non-blocking. The input condition of function end requires that end has not been called. Its output condition is vacuous, so it is non-blocking.

A program that implements this service can assume that its input functions will be called only when their service input conditions hold; it need not check for them. As with any service program, the input and output parts are to be executed atomically by the platform. The powerful (read-modify-write) atomicity of acq's return is not a problem because the lock service program is intended for analysis and not for execution. The progress assumption is expressed with "leads-to" assertions rather than fairness assertions.

Each allowed evolution of the service program consists of an atomic execution of the main code followed by a sequence of atomic executions of input parts and output parts. At any time, any input (or output) part can be executed as long as the input (or output) condition holds and a thread exists or can arrive from the environment. Thus the program defines *all* evolutions such that: (1) each user i cycles through acq call, acq return, rel call, and rel return; (2) between any two successive acq returns, there is a rel call by the user of the first acq return; (3) no calls can be made after an end call; (4) every rel call eventually returns; (5) every end call eventually returns; and (6) every acq call eventually returns provided no user holds the lock indefinitely. The last three conditions come from the progress assumption.

This lock service is more general than the simple lock because it allows requests to be served in any order, only requiring that no request is indefinitely delayed. Thus it represents many possible implementations, including the simple lock given earlier.

2.4 Implements Conditions

We now state the conditions that must hold for SimpleLock(N) (or any other program) to implement SimpleLockService(N). The conditions are proved in the next section. There is a safety condition and a progress condition. Roughly speaking, the safety condition is that whenever the lock and the service have undergone the same input-output history, the lock can accept any input that the service can accept, and any

output the lock does is an output that the service can do. The progress condition is that the lock satisfies the service's progress assumption. These conditions can be formalized in terms of the evolutions of the lock and the service (as shown in Sect. 1.7). But it is better formalized in terms of the lock and service programs, and we proceed with that now.

The first step is to "invert" the service program, i.e., interchange its inputs and outputs, resulting in a program that provides the most general environment that the lock can expect. This program, called `SimpleLockServiceInverse`, is shown in Figs. 2.3 and 2.4. It has an additional parameter, `lck`, matching the return value of the service program's main code; it will be set to the sid of the lock implementation to be tested. Input parts become output parts, output parts become input parts, and `mysid` becomes `lck`. For example, input function `mysid.acq` becomes an output function that does an output call `lck.acq`. The progress assumption is now a condition to be satisfied. For example, the first progress assertion, where `mysid.rel` has become `lck.rel`, requires system `lck` to eventually return `lck.rel` calls.

The next step is to define a program, say `Z`, that executes the lock and service inverse concurrently. The program is shown in Fig. 2.5. It has the same parameter and input condition as the service program. It starts a lock system, `lck`, and a service inverse system, `lsi`, which interact solely with each other (because aggregate system `Z` is "closed"). It needs no atomicity assumptions because its basic system has only thread.

The final step is to write down the assertions that aggregate system `Z` must satisfy in order for lock to implement service. They are shown in Fig. 2.6. B_0 is a safety assertion stating that `lck` returns an `acq` call only if the corresponding input conditions in `lsi` holds. No assertion is needed for `lck`'s return of a `rel` call because `lsi.doRel`'s input condition is vacuous. The same is true for `lck`'s return of an `end` call. $B_1 - B_3$ are simply the assertions in `lsi`'s progress condition. (We can write "thread `i` in `lck.rel`" without preceding it by "`lck.alive`" because of our convention that `lck`'s state is available for writing assertions even after its termination.)

2.5 Proving the Implements Conditions

The goal is to prove that aggregate system `Z` satisfies assertions $B_0 - B_3$. For ease of reference, the simple lock and service inverse programs are shown together in Fig. 2.7. The service inverse has an atomicity breakpoint (marked by "•") at each output condition; this is the case for any service or service inverse program. Together with the atomicity breakpoints of `SimpleLock`, they define the following steps to be effectively atomic:

- `Z`'s initial step: consisting of `Z`'s main, `lck`'s initial step, and `lsi`'s main.
- Step `lsi.doAcq` call: from `lsi.doAcq`'s output condition to the atomicity breakpoint in `lck.a6`.

```
service SimpleLockService(int N) {
program SimpleLockServiceInverse(int N, Sid lck) {
   // lck: lock system being tested
   ic {N ≥ 1}
   boolean[N] acqd ← false;
   ending ← false;
   return mysid;
   // end main

   input void mysid.acq() {
   output doAcq() {
      ic oc {not ending and (mytid in 0..N-1)
             and not acqd[mytid]}
      lck.acq();                                    // added call
      oc ic {forall(j in 0..N-1: not acqd[j])}
      acqd[mytid] ← true;
      return;
   }

   input void mysid.rel() {
   output doRel() {
      ic oc {not ending and (mytid in 0..N-1)
             and acqd[mytid]}
      acqd[mytid] ← false;
      lck.rel();                                    // added call
      oc ic {true}
      return;
   }

   input void mysid.end() {
   output doEnd() {
      ic oc {not ending}
      ending ← true;
      lck.end();                                    // added call
      oc ic {true}
      return;
   }

// continued
```

Fig. 2.3 Part 1: `SimpleLockServiceInverse`, obtained from `SimpleLockService` after deletions and additions (*underlined*)

```
// SimpleLockServiceInverse continued

   atomicity assumption {input parts and output parts}

   progress ~~assumption~~ condition {
     forall(Tid i: (thread i in ~~mysid~~ lck.rel) leads-to (not i in ~~mysid~~ lck.rel));

     forall(i in 0..N-1: acqd[i] leads-to not acqd[i])
       ⇒ forall(Tid i: (thread i in ~~mysid~~ lck.acq)
                         leads-to (not i in ~~mysid~~ lck.acq));

     forall(Tid i: (thread i in ~~mysid~~ lck.end) leads-to (not i in ~~mysid~~ lck.end));
   }
}
```

Fig. 2.4 Part 2: SimpleLockServiceInverse, obtained from SimpleLockService after deletions and additions (*underlined*)

```
program Z(int N) {
  ic {N ≥ 1}
  inputs(); outputs();                          // aggregate system Z is closed
  Sid lck ← startSystem(SimpleLock(N));         // lock
  Sid lsi ← startSystem(SimpleLockServiceInverse(N, lck)); // service inverse
  return mysid;

  atomicity assumption {}
  progress assumption {weak fairness}
}
```

Fig. 2.5 Program of lock and inverse lock service; main body is effectively atomic

```
B0 : Inv (thread i at lsi.doAcq.ic) ⇒ forall(j in 0..N-1: not acqd[j])
B1 : forall(Tid i: (thread i in lck.rel) leads-to (not i in lck.rel))
B2 : forall(i in 0..N-1: acqd[i] leads-to not acqd[i])
       ⇒ forall(Tid i: (thread i in lck.acq) leads-to (not i in lck.acq))
B3 : forall(Tid i: (thread i in lck.end) leads-to (not i in lck.end))
```

Fig. 2.6 Assertions to be satisfied by program Z in order for SimpleLock to implement SimpleLockService

- Step lck.acq return: from the atomicity breakpoint in lck.a6 to the end of lsi.doAcq.
- Step lsi.doRel: from lsi.doRel's output condition to the end of lsi.doRel, including lck.rel's body.

```
program SimpleLock(int N) {
   ia {N ≥ 1}
   boolean[N] xreq← false;
   boolean xacq ← false;
   int xp ← 0;
   Tid t ← startThread(serve());
   return mysid;

   function void serve() {
     while (true) {
  a0:  if ● (xreq[xp]) {
  a1:   ● xacq ← true;
  a2:   ● xreq[xp] ← false;
  a3:     while ● (xacq) skip;
       }
  a4:  xp ← mod(xp+1,N);
     }
   }

   input void mysid.acq() {
      ia {mytid in 0..N-1}
  a5: xreq[mytid] ← true;
  a6: while ● (xreq[mytid]) skip;
      return;
   }

   input void mysid.rel() {
      ia {mytid in 0..N-1}
  a7: xacq ← false;
      return;
   }

   input void mysid.end() {
      ia {true}
      endSystem();
   }

   atomicity assumption {...}
   progress assumption {...}
}
```

```
program SimpleLockServiceInverse
                  (int N, Sid lck) {
   ic {N ≥ 1}
   boolean[N] acqd ← false;
   ending ← false;
   return mysid;

   output doAcq() {
   ● oc {not ending
         and (mytid in 0..N-1)
         and not acqd[mytid]}
      lck.acq();
      ic {forall(j in 0..N-1: not acqd[j])}
      acqd[mytid] ← true;
      return;
   }

   output doRel() {
   ● oc {not ending
         and (mytid in 0..N-1)
         and acqd[mytid]}
      acqd[mytid] ← false;
      lck.rel();
      ic {true}
      return;
   }

   output doEnd() {
   ● oc {not ending}
      lck.end();
      ic {true}
      return;
   }

   atomicity assumption {...}

   progress condition {...}
}
```

Fig. 2.7 Component programs of `Z(N)`; "●"s are atomicity breakpoints

- Step lsi.doEnd: from lsi.doEnd's output condition to the end of lsi.doEnd, including lck.end's body.
- The steps inside lck defined by the internal atomicity breakpoints at a0, a1, a2 and a3. (The input condition of lsi and the output conditions of lsi.doAcq, lsi.doRel and lsi.doEnd ensure that every input to lck is allowed. Hence the atomicity breakpoints established for SimpleLock also hold for instantiation lck.)

2.5.1 *Proving the Safety Condition:* B_0

We now prove that Z satisfies B_0. Given Z's effective atomicity, thread i comes to lsi.doAcq.ic iff it was on statement lck.a6 and lck.xreq[i] was false. Thus B_0 is equivalent to $Inv\,C_0$, where C_0 is as follows. (Note that C_0 is the predicate of invariant assertion $Inv\,C_0$, whereas B_0 is an invariant assertion.)

C_0 : ((thread i on lck.a6) and not lck.xreq[i])
⇒ forall(j in 0..N−1: not lsi.acqd[j])

It suffices to prove that Z satisfies $Inv\,C_0$. Henceforth we omit the lck and lsi prefixes when there is no ambiguity, and we sometimes shorten "thread t on S" to "t on S". Let's take a closer look at the evolution of Z assuming that lck remains alive. After Z's initial step, xacq and the entries of xreq and acqd are false, and thread t starts repeatedly executing a0 and a4. When a thread i executes step doAcq call, it sets xreq[i] to true and waits on a6. When thread t finds xreq[xp] true (in a0), it executes a1 and a2 and waits on a3. Until t executes a2, every acqd entry is false. Thread xp, which would be waiting on a6, finds xreq[xp] false and executes step lck.acq return, which sets acqd[xp] to true (in doAcq's input part). Thread t remains on a3 until thread xp executes step doRel, setting xacq and acqd[xp] to false. At this point, t executes a3 and a4 and returns to a0.

So when thread t is not on a3 (and lck is alive), nobody has the lock. When thread t is on a3 (which also implies that lck is alive), thread xp has the lock or is about to get it, and nobody else has the lock. In terms of assertions, we have established $Inv\,C_1$ and $Inv\,C_2$, where:

C_1 : (lck.alive and not (thread t on a3))
⇒ forall(j in 0..N−1: not acqd[j])
C_2 : (thread t on a3)
⇒ ((acqd[xp]
or (not acqd[xp] and (thread xp on a6) and not xreq[xp]))
and forall(j in 0..N−1, j ≠ xp: not acqd[j]))

Given that $Inv\,C_1$ and $Inv\,C_2$ hold, it's easy to prove $Inv\,C_0$. Assume that lck is alive, otherwise C_0 holds vacuously. Initially C_0 holds (vacuously, because thread i is not on a6). Thread i comes to a6 by executing step doAcq call; this sets xreq[i] to true and hence preserves C_0 vacuously. Variable xreq[i] becomes false when t executes a2 with xp equal to i; at this point no one has the lock (because C_1 holds

non-vacuously before this step) and hence C_0's rhs holds. While thread i remains on a6, no other thread can set its acqd entry to true (because that would violate $Inv\,C_2$). When thread i leaves a6, C_0 holds vacuously. So $Inv\,C_0$ holds and we are done.

2.5.2 *Proving the Progress Condition:* $B_1 - B_3$

We now prove that Z satisfies assertions $B_1 - B_3$. B_1 says that every lck.rel call returns. It holds because the body of step lck.rel has no loops and is executed with weak fairness (from lck's progress assumption). B_3 says that every lck.end call eventually returns. It holds in the same way as B_1. B_2 says that every lck.acq call returns if the lock is not acquired indefinitely. It is has the form $D_0 \Rightarrow D_1$, where

D_0 : `forall(i in 0..N-1: acqd[i]` *leads-to* `not acqd[i])`
D_1 : `forall(i in 0..N-1: (thread i in lck.acq)` *leads-to* `(not i in lck.acq))`

What remains is to establish that Z satisfies D_1 assuming D_0. Because step doAcq call is atomic, "thread j in lck.acq" is equivalent to "thread j at a6". Suppose thread t is at a0 with xp = j and xreq[j] true. At this point, thread j is waiting on a6 and xreq[j] stays true (because only t can make it false). Thus t eventually executes statements a0..a2 (because of weak fairness), setting xreq[j] to false and xacq to true, and starts waiting on a3. Hence thread j, which is still on a6, eventually executes step lck.acq return, at which point acqd[j] becomes true. Thus the following holds:

D_2 : `((thread t on a0) and xp = j and xreq[j])`
leads-to `((t on a3) and xp = j and acqd[j])`

If D_2's rhs holds, then thread j eventually executes doRel (because of assumption D_0), making xacq false. It stays false (because only t can make it true), and so t eventually leaves a3. Thus the following holds. (Recall that a4's execution is combined with a3's.)

D_3 : `((thread t on a3) and xp = j and acqd[j])`
leads-to `((t on a0) and xp = mod(j+1,N))`

D_2 and D_3 imply the following:

D_4 : `((thread t on a0) and xp = j)` *leads-to* `((t on a0) and xp = mod(j+1,N))`

Thus xp keeps increasing modulo-N. Hence for every thread j such that j on a6, thread lck eventually comes to a0 with xp = j, where it finds xreq[j] to be true, and so eventually acqd[j] becomes true (from D_2). This establishes D_1, and we are done.

2.5.3 *Assertional Proof of Inv* C_1

The previous analysis illustrated assertional reasoning with operational proofs. We now illustrate an assertional proof for $InvC_1$, that is, a proof consisting of applications of proof rules. We use only one proof rule.

> **Invariance induction rule**: Z satisfies $InvC_1$ if there is a predicate D satisfying the following:
>
> 1. Z's initial atomic step unconditionally establishes D.
> 2. Every non-initial atomic step of Z unconditionally preserves D.
> 3. $D \Rightarrow C_1$ holds.

Given a predicate D, it is easy to check whether the conditions of the rule hold. Finding a suitable D may not be easy. The natural way to do this is to consider atomic steps that do not preserve C_1, and then identify what more needs to hold so that they preserve C_1.

Note that if we don't want `acqd[j]` to hold in a situation, then we also don't want thread `j` to be about to get the lock, i.e., "thread `j` on `a6` and not `xreq[j]`" to hold. Otherwise thread `j` can execute the `lck.acq` return, making `acqd[j]` true. Let `z(j)` denote that `j` has the lock or is about to get it. Formally,

```
z(j): acqd[j] or ((thread j on a6) and not xreq[j])
```

We will establish the invariance of the following stronger version of C_1:

E_0 : `(lck.alive and not (thread t on a3)) ⇒ forall(j in 0..N-1: not z(j))`

In order for E_0 to hold when `t` leaves `a3`, the following should be invariant:

E_1 : `((t on a3) and not xacq) ⇒ forall(j in 0..N-1: not z[j])`

When `t` is on `a3` and `xacq` is true, we expect that thread `xp`, and no one else, has the lock or is about to get it.

E_2 : `((t on a3) and xacq) ⇒ (z(xp) and forall(j in 0..N-1, j ≠ xp: not z(j)))`

Conversely, when thread `i` has the lock or is about to get it, we expect thread `t` to be waiting on `a3` for `i`:

E_3 : `z(i) ⇒ ((t on a3) and xp = i and xacq)`

The following are also needed (e.g., E_4 is needed to preserve E_3 after `t` does `a2`):

E_4 : `(t on a2) ⇒ xacq`
E_5 : `(t on a1..a2) ⇒ (xreq[xp] and (xp on a6))`
E_6 : `forall(j in 0..N-1: xreq[j] ⇒ (j on a6))`

It's easy to check that the conjunction of $E_0 - E_6$ is a suitable predicate D for the invariance induction rule. Consider the initial step: it establishes the rhs of E_0 and E_1, and falsifies the lhs of the others. Consider step `a0` by `t`: it preserves E_0 (i.e., E_0 before the step ensures that E_0 holds after the step); it preserves E_3 and E_6; it establishes E_1, E_2, E_4 vacuously; it establishes E_5 (from E_6 holding prior to the step). Each of the remaining atomic steps can be similarly shown to preserve $E_0 - E_6$. Note that one can *check* this without having any global understanding of program `Z`. All that is needed is to understand the individual constructs of program `Z` and predicate D.

2.5.4 *Assertional Proof of* D_2

We now illustrate an assertional proof for the progress assertion D_2. The following proof rules are used:

> **Leads-to via thread** `t` **rule**: `Z` satisfies *P leads-to Q* if the following hold:
>
> 1. Every non-initial atomic step of `Z` unconditionally establishes *P* `or` *Q* from *P* `and not` *Q*.
> 2. Every non-initial atomic step of thread `t` in `Z` unconditionally establishes *Q* from *P* `and not` *Q*.

> **Leads-to closure rules**:
>
> - *P leads-to Q* holds if *P leads-to R* and *R leads-to Q* hold.
> - *P leads-to Q* holds if $Inv\,(P \Rightarrow Q)$ holds.

The following lines establish that `Z` satisfies D_2. Each line below states an assertion (at the left) and a proof rule application (at right) that establishes the assertion. D_2 follows from the closure of F_0 through F_5.

F_0 : `((t on a0) and xp = j and xreq[j])`
leads-to `((t on a1) and xp = j and xreq[j])` [via `t`]

F_1 : `((t on a1) and xp = j and xreq[j])`
leads-to `((t on a2) and xp = j and xreq[j])` [via `t`]

F_2 : `((t on a2) and xp = j and xreq[j])`
leads-to `((t on a2) and xp = j and xreq[j]`
`and xacq and (j on a6))` [F_1, $Inv\,E_4$, $Inv\,E_6$, closure]

F_3 : ((t on a2) and xp = j and xreq[j]
and xacq and (j on a6))
leads-to ((t on a3) and xp = j and not xreq[j]
and xacq and (j on a6)) [via t]

F_4 : ((t on a3) and xp = j and not xreq[j]
and xacq and (j on a6))
leads-to ((t on a3) and xp = j and not xreq[j]
and xacq and (j on a6)
and forall(i in 0..N-1, i ≠ j: not acqd[j])) [$Inv E_2$]

F_5 : ((t on a3) and xp = j and not xreq[j]
and xacq and (j on a6)
and forall(i in 0..N-1, i ≠ j: not acqd[j]))
leads-to ((t on a3) and xp = j and acqd[j]) [via j]

2.6 Producer and Consumer Using Lock Service

So far we have seen one role of the lock service program, that is, as the standard that any lock implementation must satisfy. We now illustrate the other role of the lock service, that is, to serve as a lock in the design of a larger program that makes use of locks. Figure 2.8 shows a program ProdCons1 in which a producer and consumer make use of the lock service. (This corresponds to the program of the same name in Chap. 1.) Using the service instead of a lock implementation makes

```
program ProdCons1(int J) {
   ia {J ≥ 0}    // J: number of items produced in total
   inputs();     // no external inputs; hides lck.end
   Tid tP ← tid(0);
   Tid tC ← tid(1);
   if (tP = null or tC = null)
      return [-1, mysid];

   // tids available; start lock, consumer, producer
   Sid lck ← startSystem(SimpleLockService(2));
   Sid cons ← startSystem(Consumer(lck, tC, J));
   Sid prod ← startSystem(Producer(lck, cons, tP, J));
   return [0, mysid];

   atomicity assumption {}  // none
   progress assumption {weak fairness}
}
```

Fig. 2.8 Producer-consumer program

ProdCons1 easier to analyze because the service program is simpler. Any correctness property of ProdCons1 is preserved when the service program is replaced by any valid implementation, including the simple lock program.

Because the lock service restricts user tids to 0 and 1, program ProdCons1 first attempts to obtain these tids (via tid(0) and tid(1)). If the tids are not available, ProdCons1 gives up and returns the tuple [-1, mysid], where -1 indicates failure. Otherwise it instantiates the lock service (Fig. 2.9), consumer (Fig. 2.11), and producer (Fig. 2.10), passes tids 0 and 1 to the producer and consumer, and returns the tuple [0, mysid], where 0 indicates success.

The producer repeatedly produces an item and gives it to the consumer by calling the consumer's input function put. (This function should have an item parameter, but it's omitted for simplicity.) After producing J items, the producer terminates itself. The consumer repeatedly empties its cache of items and consumes them. The producer and consumer use the lock service to ensure that an item is not put in the cache while the cache is being consumed. After consuming J items, the consumer terminates the lock system and then itself.

We start our analysis of aggregate system ProdCons1 by showing that it is fault-free. System lck is instantiated first; it has no local thread so it makes no calls. System cons is instantiated next; it has a local thread whose only calls are to lck, which already exists. System prod is instantiated next; it has a local thread whose only calls are to lck and cons, both of which already exist. Systems prod and cons use the lock service correctly, i.e., lck is called only by a thread with tid 0 or 1, and that each thread cycles through lck.acq and lck.rel. (If this did not hold, the lock implementation, which would replace the lock service at run time, can do anything, and nothing could be claimed about aggregate ProdCons1.) Thus aggregate system ProdCons1 is fault-free.

Next we partition the code of aggregate system ProdCons1 into effectively atomic steps. It turns out that a single atomicity breakpoint, at lck.acq.oc, is all that is needed. Thus the following steps are effectively atomic:

- ProdCons1 initial step: consisting of ProdCons1 main, lck main, cons main, prod main, consume initial step (start to atomicity breakpoint), and produce initial step (start to atomicity breakpoint).

  Except for the consume and produce initial steps, this step is executed by a single thread. System lck's main has no conflict with lck's input functions because those functions are called only after prod and cons become alive, which happens only after lck's main returns. Similarly, cons's main has no conflict with cons's input function because the latter is called only after prod becomes alive, which happens only after cons main returns. The initial steps of produce and consume, executed by threads tP and tC, have no conflict because np and nc are accessed by only one thread.
- Step produce iteration: thread tP executing from atomicity breakpoint back to the atomicity breakpoint or to producer termination (if np equals J). In particular, the calls to cons.put, lck.rel and endSystem are nonblocking.

```
service SimpleLockService(int N) {
   ic {N ≥ 1}    // N: # lock users
   ...
   return mysid;

   input void mysid.acq() {
      ic {...}
    • oc {...}
   }

   input void mysid.rel() {...}

   input void mysid.end() {...}

   atomicity assumption {...}
   progress assumption {...}
}
```

Fig. 2.9 Lock service; "•" is atomicity breakpoint

```
program Producer(Sid lck, Sid cons,
                 Tid tP, int J) {
  ia {true}
  int np ← 0;  // # items produced
  startThread(tP, produce());
  return mysid;

  function void produce() {
    while (np < J) {
       np ← np+1; // produce item
       lck.acq();  // acquire lock
       cons.put(); // item to cons
       lck.rel();  // release lock
    }
    endSystem();
  }

  // none: only one thread
  atomicity assumption { }

  progress assumption {weak fairness}
}
```

Fig. 2.10 Producer program

```
program Consumer
           (Sid lck,Tid tC, int J) {
  ia {true}
  int nc ← 0;    // # items consumed
  int cache ← 0; // # items in cache
  startThread(tC, consume());
  return mysid;

  function void consume() {
    while (nc < J) {
       lck.acq();  // acquire lock
       // consume cache
       nc ← nc+cache;
       cache ← 0;
       lck.rel();  // release lock
    }
    lck.end();
    endSystem();
  }

  input void mysid.put() {
    ia {true}
    cache ← cache+1;
    return;
  }

  // none: uses lck instead
  atomicity assumption { }
  progress assumption {weak fairness}
}
```

Fig. 2.11 Consumer program

- Step consume iteration: thread tC executing from atomicity breakpoint back to the atomicity breakpoint or to consumer termination (if nc equals J).

  In particular, consume's accesses to cache do not conflict with put's access to cache because the lock service prevents them from being executed at the same time. (If this observation is not obvious, one can prove it by treating the statements involving cache as null statements and proving that the resulting program allows at most one thread at any of the null statements at any time.)

Any further analysis can now be conveniently done. Here are some desired properties, whose proof is left as an exercise. G_0 says that only produced items are consumed. G_1 says that J items are consumed eventually. G_2 says that the producer, consumer, and lock systems are eventually terminated.

G_0 : *Inv* `cons.nc` $\leq$ `prod.np`
G_1 : `true` *leads-to* `cons.nc` $=$ `J`
G_2 : `true` *leads-to* `not (lck.alive or prod.alive or cons.alive)`

2.7 Concluding Remarks

This chapter has illustrated all the main pieces of SESF. We presented a simple lock that assumes a platform with only read-write atomicity and weak fairness. We defined its intended lock service. We defined the conditions for the simple lock to implement the lock service, and showed that they hold. We then used the lock service in a producer-consumer program.

To keep the example simple, our lock service constrains user ids to be from `0..N-1` where `N` is fixed at lock creation. Whereas the lock service usually provided in a programming language does not have this constraint. Also for simplicity, the lock implementation here has a local thread that constantly checks for a request, which is wasteful. Of course, there are algorithms that assume the same platform and avoid local threads, e.g., [1–4]. We will see such lock implementations in Chaps. 9 and 10.

The implementation here also makes users do busy waiting; i.e., when a user thread requests the lock, it busy waits until it acquires the lock. This is unavoidable given an underlying platform without any built-in synchronization constructs (i.e., a bare hardware platform). Most programming languages that provide locks execute on platforms that provide non-busy waiting (i.e., without consuming processor cycles), and their lock implementation would, of course, exploit that. These languages also usually provide synchronization constructs that allow threads to sleep and be awakened. We will see such constructs in Chap. 3.

Exercises

2.1. Change the order of a1 and a2 in SimpleLock, that is, put "xacq ← true" after "xreq[xp] ← false". Does this revised program implement the simple lock service. If you answer no, give a counter-example evolution.

2.2. Show that aggregate program ProdCons1 (in Fig. 2.8) satisfies the assertions G_0–G_2 (in Sect. 2.6). Here are some intermediate assertions you may want to prove:

- (prod.np ≥ k and k < J) *leads-to* prod.np ≥ k+1
- prod.np ≥ k *leads-to* cons.nc ≥ k
- true *leads-to* lck.endable
- true *leads-to* cons.endable
- true *leads-to* prod.endable

2.3. Write a service program for program Producer (in Fig. 2.10). The program would have no input functions and three output functions (calling lck.acq, lck.rel and cons.put).

2.4. Write a service program for program Consumer (in Fig. 2.11). The program would have one input function (mysid.put) and two output functions (calling lck.acq and lck.rel).

2.5. Program ProdCons2 is like ProdCons1 except that system lck can be instantiated to handle more than two users, and threads in the environment of aggregate system ProdCons2 can call lck.acq and lck.rel.

```
program ProdCons2(int J, int N) {
   ia {J ≥ 0 and N > 2}
   // lck.acq() and lck.rel() callable by threads outside aggregate ProdCons2
   inputs(lck.acq(), lck.rel());

   Tid tP ← tid(0);
   Tid tC ← tid(1);
   if (tP = null or tC = null)          // if tid 0 or 1 not available, exit
      return [-1, mysid];

   Sid lck ← startSystem(SimpleLockService(N));  // lock service for N users
   Sid cons ← startSystem(Consumer(lck, tC));       // local thread tC
   Sid prod ← startSystem(Producer(lck, cons, tP)); // local thread tP
   return [0, mysid];

   atomicity assumption {}  // none
   progress assumption {weak fairness}
}
```

Thus threads with tids from 2..N-1 in the environment of aggregate system ProdCons2 can acquire and release the lock.

(a) Under what conditions will any safety property of aggregate `ProdCons1` also hold in aggregate `ProdCons2`?
(b) Under what conditions will any progress property of aggregate `ProdCons1` also hold in aggregate `ProdCons2`?

2.6. Modify the simple lock service so as to allow a user to request the lock even if it already has it and to release the lock even if it does not have it. The lock service should ignore invalid requests and otherwise provide the usual lock service.

2.7. Modify the simple lock service so that the service can inform a user to release the lock. Specifically, add a new "release indication" input function, say `relInd()`, which a user can call when it has the lock. The service returns the call either to tell the user to release the lock or after the user releases the lock.

(Such a lock service would be relevant in situations where conflicts between users are settled by aborting one or more users. Consider a database of objects, each with its own lock. A transaction updates a sequence of objects, obtaining their locks as needed. If two transactions conflict, i.e., both are half-way and each has done updates inconsistent with the other's, then at least one has to be aborted, in which case it has to be told to give up the locks it currently owns.)

2.8. Exercise 2.7 defined a lock service in which the service can tell a user to give up the lock. Another way to achieve such as service is as follows: (1) add a parameter, say `sys`, to the lock request input by which the user provides a sid when it requests the lock; (2) add an output function that does a "release request" output `s.relReq(lck)`, where `s` is the `sys` provided by the user with the lock; and (3) make all the modifications you think necessary. Compare this service with that obtained in Exercise 2.7. Which do you prefer and why?

2.9. Obtain the inverse program of the lock service from Exercise 2.7.

2.10. Obtain a lock that implements the lock service in Exercise 2.7. Your lock program should assume the same platform as program `SimpleLock`.

2.11. Obtain the inverse program of the lock service from Exercise 2.8.

2.12. Obtain a lock that implements the lock service in Exercise 2.8. Your lock program should assume the same platform as program `SimpleLock`.

References

1. E.W. Dijkstra, Solution of a problem in concurrent programming control. Commun. ACM **8**(9), 569– (1965). doi:10.1145/365559.365617. http://doi.acm.org/10.1145/365559.365617
2. L. Lamport, A new solution of dijkstra's concurrent programming problem. Commun. ACM **17**(8), 453–455 (1974). doi:10.1145/361082.361093. http://doi.acm.org/10.1145/361082.361093
3. G.L. Peterson, Myths about the mutual exclusion problem. Inf. Process. Lett. **12**(3), 115–116 (1981)
4. G. Taubenfeld, The black-white bakery algorithm and related bounded-space, adaptive, local-spinning and fifo algorithms. In: *DISC* ACM, New York, (2004), pp. 56–70

http://www.springer.com/978-1-4614-4881-5

Distributed Programming
Theory and Practice
Shankar, A.U.
2013, XVIII, 386 p.,
ISBN: 978-1-4614-4881-5