Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 — Principles of Computer Systems

Handout 16a March 19, 1997

Correction to `Mutex` Proof

This handout repeats the second, direct proof that `SpinLock` implements `Mutex` from handout 16 correcting the error in the handout's treatment of the case that corresponds to `HAVOC` in the spec, where a thread releases a mutex that it did not hold. The changes are marked with bars in the margin, and we have omitted some repetitive details.

This handout also repeats the proofs of `Mutex2Impl` and `ClockImpl`, and the code for `MutexImpl` and `ConditionImpl`, all from handout 16, so that we can discuss them today.

Abstraction function

We have to introduce a history variable. We do this as follows:

- We augment the state of the code with two additional variables:

```
ms: (Thread + Null) := nil

hs: Bool
```

- We define the effect of each atomic action in the code on the history variable; written in Spec, this results in the following modified code:

```
PROC Acquire () = VAR t: AH |
  DO <<t := m; m := held>>; IF t # held => << ms := SELF >>; RET [*] SKIP FI OD;

PROC Release () = << m := available; hs := hs \/ (ms = SELF); ms := nil >>
```

You can easily check that these additions to the code satisfy the constraints required for adding history variables.

Now we can proceed to define the abstraction function. Here are the routines in `Mutex` annotated with the PC values.

`APROC Acquire() =` $[A_1]$ `<< m = nil => m := SELF >>` $[A_2]$

`APROC Release() =` $[R_1]$ `<< m # SELF => HAVOC [*] m := nil >>` $[R_2]$

Here are the routines in `SpinLock` annotated with the PC values. The transitions in `Acquire` may be a little confusing: there's a transition from a_4 to a_3, as well as transitions from a_4 to a_5.

```
PROC Acquire () = [a1] VAR t := AH |
     [a2] DO [a3] << t := m; m := held >>;
     [a4] IF t # held => [a5] << ms := SELF >>; [a6] RET [*] SKIP FI OD;

PROC Release () = [r1] << m := available; hs := hs \/ (ms = SELF); ms := nil >> [r2]
```

Now we can define the mappings on program counters:

- If a thread is not in `Acquire` or `Release` in the code, then it is not in either in the spec.
- $\{a_1, a_2, a_3, a_4, a_5\}$ maps to A_1, a_6 maps to A_2
- r_1 maps to R_1, r_2 maps to R_2

The part of the abstraction function dealing with the global variables of the module simply defines `m` in the spec to have the value of `ms` in the code, and `$havoc` in the spec to have the value of `hs` in the code. As in handout 8, we just throw away all but the spec part of the state.

Since there are no local variables in the spec, the mapping on program counters and the mapping on the global variables are enough to define how to construct a state of the spec from a state of the code.

Once again, the abstraction function on program counters determines how transitions in the code simulate sequences of transitions in the spec:

- If π is an external transition, π simulates the singleton sequence containing just π.
- If π takes a thread from a_5 to a_6, π simulates the singleton sequence containing just the transition from A_1 to A_2.
- If π takes a thread from r_1 to r_2, π simulates the singleton sequence containing just the transition from R_1 to R_2.
- All other transitions simulate the empty sequence.

Proof sketch

As in the previous example, we will need some invariants to do the proof. Rather than trying to write them down first, we will see what we need as we do the proof.

First, we show that initial states of the code map to initial states of the spec. This is easy; all the thread states correspond, and the initial state of `ms` in the code is `nil`.

Next, we show that transitions in the code and the spec correspond. All transitions from outside the module to just before a routine's body are straightforward, as are transitions from the end a routine's body to outside the module (i.e., when a routine returns). The transition in the body of `Release` is also straightforward. The hard cases are in the body of `Acquire`.

Consider all the transitions in `Acquire` before the one from a_5 to a_6. These all simulate the null transition, so they should leave the abstract state unchanged. And they do, because none of them changes `ms`.

The transition from a_5 to a_6 simulates the transition from A_1 to A_2. There are two cases: when `ms = nil`, and when `ms` $\neq$ `nil`.

1. In the first case, the transition from A_1 to A_2 is enabled and, when taken, changes the state so that `m = SELF`. This is just what the transition from a_5 to a_6 does.

2. Now consider the case when `ms` $\neq$ `nil`. We claim this case is possible only if a thread that didn't hold the mutex has done a `Release`. Then `hs = true`, the spec has done `HAVOC`, and anything can happen. In the absence of havoc, if a thread is at a_5, then `ms = nil`. But even though this invariant is what we want, it's too weak to prove itself inductively; for that, we need the following, stronger invariant:

   > Either
   >
   > > If `m = available` then `ms = nil`, and
   > >
   > > If a thread is at a_5, or at a_4 with `t = available`, then `ms = nil`, `m = held`, there are no other threads at a_5, and for all other threads at a_4, `t = held`
   >
   > or `hs` is true.

Given this invariant, we are done: we have shown the appropriate correspondence for all the transitions in the code. So we must prove the invariant. We do this by induction. It's vacuously true in the initial state, since no thread could be at a_4 or a_5 in the initial state. Now, for each transition, we assume that the invariant is true before the transition and prove that it still holds afterwards.

The external transitions preserve the invariant, since they change nothing relevant to it.

The transition in `Release` preserves the first conjunct of the invariant because afterwards both `m = available` and `ms = nil`. To prove that the transition in `Release` preserves the second conjunct of the invariant, there are two cases, depending on whether the spec allows `HAVOC`.

1. If it does, then the code sets `hs` true; this corresponds to the `HAVOC` transition in the spec, and thereafter anything can happen in the spec, so any transition of the code simulates the spec. The reason for explicitly simulating `HAVOC` is that the rest of the invariant may not hold after a rogue thread does `Release`. Because the rogue thread resets `m` to `available`, it's possible for several threads to get to a_5, or to a_4 with `t = available`.

2. In the normal case `ms` $\neq$ `nil`, and since we're assuming the invariant is true before the transition, this implies that no thread is at a_4 with `t = available` or at a_5. After the transition to r_2 it's still the case that no thread is at a_4 with `t = available` or at a_5, so the invariant is still true.

Now we consider the transitions in `Acquire`. The transitions from a_1 to a_2 and from a_2 to a_3 obviously preserve the invariant. The transition from a_4 to a_5 puts a thread at a_5, but `t = available` in this case so the invariant is true after the transition by induction. The transition from a_4 to a_3 also clearly preserves the invariant.

The transition from a_3 to a_4 is the first interesting one. We need only consider the case `hs = false`, since otherwise the spec allows anything. This transition certainly preserves the first conjunct of the invariant, since it doesn't change `ms` and only changes `m` to `held`. Now we assume the second conjunct of the invariant true before the transition. There are two cases:

1. Before the transition, there is a thread at a_5, or at a_4 with `t = available`. Then we have `m = held` by induction, so after the transition both `t = held` and `m = held`. This preserves the invariant.

2. Before the transition, there are no threads at a_5 or at a_4 with `t = available`. Then after the transition, there is still no thread at a_5, but there is a new thread at a_4. (Any others must have `t = held`.) Now, if this thread has `t = held`, the second part of the invariant is true vacuously; but if `t = available`, then we have:

   `ms = nil` (since when the thread was at a_3 `m` must have been `available`, hence the first part of the invariant applies);

   `m = held` (as a direct result of the transition);

   there are no threads at a_5 (by assumption); and

   there are no other threads at a_4 with `t = available` (by assumption). So the invariant is still true after the transition.

Finally, assume a thread `h` is at a_5, about to transition to a_6. If the invariant is true here, then `h` is the only thread at a_5, and all threads at a_4 have `t = held`. So after it makes the transition, the invariant is vacuously true, because there is no other thread at a_5 and the threads at a_4 haven't changed their state.

We have proved the invariant. The invariant implies that if a thread is at a_5, `ms = nil`, which is what we wanted to show.

This proof is a good example of how to use invariants and of the subtleties associated with preconditions. It's possible to give a considerably simpler proof, however, by handling the history variable `ms` in a less natural way. This verrsion is closer to the two-stage proof we saw earlier. In particular, it uses the transition from a_3 to a_4 to simulate the body of `Mutex.Acquire`. We omit the `hs` history variable and augment the code as follows:

```
PROC Acquire () = [a1] VAR t := AH |
    | [a2] DO [a3] << t := m; m := held; IF t # held => ms := SELF [*] SKIP FI >>;
    | [a4] IF t # held => [a6] RET [a7] [*] SKIP FI OD;

PROC Release () = [r1] << m := available; ms := nil >> [r2]
```

The abstraction function maps `ms` to `Mutex.m` as before, and it maps PC's a_1- a_3 to A_1 and a_6-a_7 to A_2. It maps a_4 to A_1 if `t = held`, and to A_2 if `t = available`; thus a_3 to a_4 simulates `Mutex.Acquire` only if `m` was `available`, as we should expect. There is no need for an invariant; we only used it at a_5 to a_6, which no longer exists.

The simulation argument is the same as before except for a_3 to a_4, which is the only place where we changed the code. If `m = held`, then `m` and `ms` don't change; hence `Mutex.m` doesn't change, and neither does the abstract PC; in this case the transition simulates the empty trace. If `m = available`, then `m` becomes `held`, `ms` becomes `SELF`, and the abstract PC becomes A_2; in this case the transition simulates A_1 to A_2, as promised.

The moral of this story is that it can make a big difference how you choose the abstraction function. The crucial decision is the choice of the critical transition that models the body of `Mutex.Acquire`. It seems very natural to change `ms` in the code after the test of `t # held` that is already there, but this forces the critical transition to be after the test. Then there has to be an invariant to carry forward the relationship between the local variable `t` and the global variable `m`, which complicates things, and the `HAVOC` case complicates them further by falsifying the natural statement of the invariant and requiring the additional `hs` variable to patch things up. The uglier code with a second test of `t # held` inside the atomic test-and-set command makes it possible to use that action, which does the real work, to simulate the body of `Mutex.Acquire`, and then everything falls out nicely.

More complicated code requires invariants even when we choose the best abstraction function, as we see in the next two examples.

`Mutex2Impl` implements `Mutex`

First we show that the simple, deadlocking version `Acquire0` maintains mutual exclusion. Recall that we write `h*` for the thread that is the partner of thread `h`. Here is the code for `Acquire0`.

```
VAR req         :  Thread -> Bool := {* -> false}

PROC Acquire0() =
    [a1] req(SELF) := true;
    DO [a2] req(SELF*) => SKIP OD [a3]
```

Intuitively, we get mutual exclusion because once `req(h)` is true, `h*` can't get from a_2 to a_3. It's convenient to define

```
FUNC Holds0(h: Thread) = RET req(h) /\ h.$pc # a2
```

Abstractly, `h` has the mutex if `Holds0(h)`, and the transition from a_2 to a_3 simulates the body of `Mutex.Acquire`. Precisely, the abstraction function is

```
    Mutex.ms = (Holds0.set = {} => nil [*] Holds0.set.choose)
```

Recall that if `P` is a predicate, `P.set` is the set of arguments for which it is true.

To make precise the idea that `req(h)` stops `h*` from getting to a_3, the invariant we need is

```
    Holds0.set.size <= 1 /\ (h.$pc = a2 ==> req(h))
```

The first conjunct is the mutual exclusion. It holds because, given the first conjunct, only (a_2, a_3) can increase the size of `Holds0.set`, and `h` can take that step only if `req(h*) = false`, so `Holds0.set` goes from `{}` to `{h}`.The second conjunct holds because it can never be

`true ==> false`, since only the step (a_1, `req(h) := true`, a_2) can make the antecedent true, this step also makes the consequent true, and no step away from a_2 makes the consequent false.

This argument applies to `Acquire0` as written, but you might think that it's unrealistic to fetch the shared variable `req(SELF*)` and test it in a single atomic action; certainly this will take more than one machine instruction. We can appeal to big atomic actions, since the whole sequence from a_2 to a_3 has only one action that touches a shared variable (the fetch of `req(SELF*)`) and therefore is atomic.

This is the right thing to do in practice, but it's instructive to see how to do it by hand. We break the last line down into two atomic actions:

```
VAR t | DO [a2] << t := req(SELF*) >>; [a21] << t => SKIP >> OD [a3]
```

We examine several ways to show the correctness of this; they all have the same idea, but the details differ. The most obvious one is to add the conjunct `h.$pc # a21` to `Holds0`, and extend the mutual exclusion conjunct of the invariant so that it covers a thread that has reached a_{21} with `t = false`:

```
(Holds0.set + {h | h.$pc = a21 /\ h.t = false}).size <= 1
```

Or we could get the same effect by saying that a thread acquires the lock by reaching a_{21} with `t = false`, so that it's the transition (a_2, a_{21}) with `t = false` that simulates the body of `Mutex.Acquire`, rather than the transition to a_3 as before. This means changing the definition of `Holds0` to

```
FUNC Holds0(h: Thread) =
    RET req(h) /\ h.$pc # a2 /\ (h.$pc = a21 ==> h.t = false)
```

Yet another approach is to make explicit in the invariant what `h` knows about the global state. One purpose of an invariant is to remember things about the global state that a thread has discovered in the past; the fact that it's an invariant means that those things stay true, even though other threads are taking steps. In this case, `t = false` in `h` means that either `req(h*) = false` or `h*` is at a_2 or a_{21}, in other words, `Holds(h*) = false`. We can put this into the invariant with the conjunct

```
h.$pc = a21 /\ h.t = false ==> Holds(h*) = false
```

and this is enough to ensure that the transition (a_{21}, a_3) maintains the invariant.

We return from this digression on proof methodology to study the non-deadlocking `Acquire`:

```
VAR req         :  Thread -> Bool := {* -> false}
    lastReq     :  Int

PROC Acquire() =
    [a11] req(SELF) := true;
    [a12] lastReq := self;
    DO [a2] (req(SELF*) /\ lastReq = SELF) => SKIP OD [a3]
```

We discussed liveness informally earlier, and we don't attempt to prove it.. To prove mutual exclusion, we need to extend `Holds0` in the obvious way:

```
FUNC Holds(h: Thread) = req(h) /\ h.$pc # a12 /\ h.$pc # a2
```

and add `\/ h.$pc = a12` to the antecedent of the invariant. We also need to add something to the invariant to express the fact that `h` won't find `lastReq = h*` as long as `h*` holds the lock. This is

```
   Holds0.set.size <= 1
/\ (h.$pc = a2  \/ h.$pc = a12  ==> req(h))
/\ (Holds(h*) /\ h.$pc = a2    ==> lastReq = h)
```

The last conjunct holds because (a_{12}, a_2) makes it true, and the only way to make it false is for `h*` to do `lastReg := SELF`, which it can only do from a_{12}, so that `Holds(h*)` is false. With this invariant it's obvious that (a_2, a_3) maintains the invariant.

`ClockImpl` implements `Clock`

Here are the modules.

```
MODULE Clock EXPORT Read =

VAR  t           :  Int := 0                              % the current time

THREAD Tick() = DO << t := t + 1 >> OD                    % demon thread advances t

PROC Read() -> Int = VAR t1: Int |
     [R1] << t1 := t >>; [R2] << VAR t2 | t1 <= t2 /\ t2 <= t => RET t2 >> [R3]

END Clock
```

```
MODULE ClockImpl EXPORT Read =

VAR  base        := 2**32                                 % constant
TYPE Word        =  Int SUCHTHAT (\ i: Int | i IN 0 .. base-1)

VAR  lo          :  Word := 0
     hi1         :  Word := 0
     hi2         :  Word := 0

THREAD Tick() = DO VAR newLo: Word, newHi: Word |
     << newLo := lo + 1 // base; newHi := hi1 + 1 >>;
     IF   << newLo # 0  => lo := newLo >>
     [*]  << hi2 := newHi >>; << lo := newLo >>; << hi1 := newHi >>
     FI

PROC Read() -> Int = VAR tLo: Word, tH1: Word, tH2: Word, t1Hist: Int |
     << tH1 := h1; t1Hist := T(lo, hi1, hi2) >>;
     << tLo := lo >>;
     << tH2 := h2; RET T(tLo, tH1, tH2) >>

FUNC T(l: Int, h1: Int, h2: Int) -> Int = h2 * base + (h1 = h2 => l [*] 0)

END ClockImpl
```

The invariant that makes this work is based on the idea that `Read` might complete before the next `Tick`, and therefore the value `Read` would return by reading the rest of the shared variables must be between `t1Hist` and `Clock.t`. We can write this most clearly by annotating the labels in `Read` with assertions that are true when the PC is there.

```
% ABSTRACTION FUNCTION Clock.t = T(lo, hi1, hi2), Clock.Read.t1 = Read.t1Hist
% The PC correspondence is R1 <-> r1, R2 <-> r2, r3, R3 <-> r4

PROC Read() -> Int = VAR tLo: Word, tH1: Word, tH2: Word, t1Hist: Int |
     [r1] << tH1 := h1; t1Hist := T(lo, hi1, hi2) >>;
     [r2] % I2: T(lo , tH1, hi2) IN t1Hist .. T(lo, hi1, hi2)
          << tLo := lo; >>
     [r3] % I3: T(tLo, tH1, hi2) IN t1Hist .. T(lo, hi1, hi2)
          << tH2 := h2; RET T(tLo, tH1, tH2) >>
     [r4] % I4: $a IN t1Hist .. T(lo, hi1, hi2)
```

The whole invariant is thus

```
h.$pc = r2 ==> I2 /\ h.$pc = r3 ==> I3 /\ h.$pc = r4 ==> I4
```

The steps of `Read` clearly maintain this invariant, since they don't change the value before `IN`. The steps of `Tick` maintain it by case analysis.

Implementing mutexes and condition variables in an operating system

This section presents code for mutexes and condition variables based on the Taos operating system from DEC SRC. Instead of spinning like `SpinLock`, it explicitly queues threads waiting for locks or conditions. The code for mutexes has a fast path that stays out of the kernel in `Acquire` when the mutex is available, and in `Release` when no other thread is waiting for the mutex. There is also a fast path for `Signal`, for the common case that there's nobody waiting on the condition. There's no fast path for `Wait`, since that always requires the kernel to run in order to reschedule the processor (unless a `Signal` sneaks in before the kernel gets around to the rescheduling).

Notes on the code for mutexes:

1. `MutexImpl` maintains a queue of waiting threads, blocks a waiting thread using `Deschedule`, and uses `Schedule` to hand a ready thread over to the scheduler to run.

2. `SpinLock` and `ReleaseSpinLock` acquire and release a global lock used in the kernel to protect thread queues.

3. The loop in `Acquire` serves much the same purpose as a loop that waits on a condition variable. If the mutex is already held, the loop calls `KernelQueue` to wait until it becomes available, and then tries again. `Release` calls `KernelRelease` if there's anyone waiting, and `KernelRelease` allows just one thread to run. That thread returns from its call of `KernelQueue`, and it will acquire the mutex unless another thread has called `Acquire` and slipped in since the mutex was released (roughly).

4. There is clumsy code in `KernelQueue` that puts the thread on the queue and then takes it off if the mutex turns out to be available. This is not a mistake; it avoids a race with `Release`, which calls `KernelRelease` to take a thread off the queue only if it sees that the queue is not empty. `KernelQueue` changes `q` and looks at `s`; `Release` uses the opposite order to change `s` and look at `q`.

This opposite-order access pattern often works in hard concurrency, that is, when there's not enough locking to do the job in a straightforward way. We saw another version of it in `Mutex2Impl`, which sets `req(h)` before reading `req(h*)`. In this case `req(h)` acts like a lock to keep `h*.$pc` = a_2 from changing from true to false.

The boxes show how `Acquire` and `Release` differ from the versions in `SpinLock`.

```
MODULE MutexImpl EXPORT M, New =                    % implements ForgetfulMutex

TYPE AH          =  Mutex.AH
     M           =  Int WITH {acq:=Acquire, rel:=Release}
VAR  s           :  M -> AH          := {}
     q           :  M -> SEQ Thread  := {}

APROC New() -> M = << VAR m | ~ s!m => s(m) := available, q(m) := {}; RET m >>

PROC Acquire(m) = VAR t: AH |
     DO <<t := s(m); s(m) := held>>; IF t#held => RET [*] SKIP FI; KernelQueue() OD

PROC Release(m) = s(m) := available; IF q(m) # {} => KernelRelease(m) [*] SKIP FI

% KernelQueue and KernelRelease run in the kernel.

PROC KernelQueue(m) =
% This is just a delay until there's a chance to acquire the lock.
% Queuing SELF before testing s(m) ensures that Release doesn't miss us.
% The spin lock keeps KernelRelease from getting ahead of us.
     SpinLock();
          q(m):= q(m) + {SELF};
          IF   s(m) = available =>
               q(m) := q(m).reml                      % try again at Acquire
          [*]  Deschedule(SELF)                       % wait, then try again
          FI;
     ReleaseSpinLock()

PROC KernelRelease(m) =
     SpinLock();
          q(m) # {} => Schedule(q(m).head); q(m):= q(m).tail;
     ReleaseSpinLock()
     % The newly scheduled thread competes with others to acquire the mutex.

END MutexImpl
```

Notes on the code for conditions:

1. In the code for `Condition`, the 'event count' `ecSig` deals with the standard 'wakeup-waiting' race condition: the `Signal` arrives after the `m.rel` but before the thread is queued. Note the use of the global spin lock as part of this. It looks as though `Signal` always schedules exactly one thread if the queue is not empty, but other threads that are in `Wait` but have not yet acquired the spin lock may keep running; in terms of the spec they are awakened by `Signal` as well.

2. `Signal` and `Broadcast` test for any waiting threads without holding any locks, in order to avoid calling the kernel in this common case. The other event count `ecWait` ensures that this test doesn't miss a thread that is in `KernelWait` but hasn't yet blocked.

```
MODULE ConditionImpl EXPORT C, New =              % implements Condition

TYPE C           =  Int WITH {wait:=Wait, signal:=Signal, broadcast:=Broadcast}
     M           =  Mutex.M

VAR  ecSig       :  C -> Int
     ecWait      :  C -> Int
     q           :  C -> SEQ Thread := {}

APROC New() -> C =
     << VAR c | ~ ecSig!c => ecSig(c) := 0; ecWait(c) := 0; q(c) := {}; RET c >>

PROC Wait(c, m) = VAR i := ecSig(c) | m.rel; KernelWait(c, i); m.acq

PROC Signal(c) = VAR i := ecWait(c) |
     ecSig(c) := ecSig(c) + 1; IF q(c) # 0 \/ i # ecWait(c) => KernelSig(c)

PROC Broadcast(c) = VAR i := ecWait(c) |
     ecSig(c) := ecSig(c) + 1; IF q(c) # 0 \/ i # ecWait(c) => KernelCast(c)

PROC KernelWait(c, i: Int) =                      % internal kernel procedure
     SpinLock();
          ecWait(c) := ecWait(c) + 1;
          % if ecSig changed, there must have been a Signal, so return, else queue
          IF i = ecSig(c) => q(c) := q(c) + {SELF}; Deschedule(SELF) [*] SKIP FI;
     ReleaseSpinLock()

PROC KernelSig(c) =                               % internal kernel procedure
     SpinLock();
          IF q(c) # {} => Schedule(q(c).head); q(c) := q(c).tail [*] SKIP FI;
     ReleaseSpinLock()

PROC KernelCast(c) =
     SpinLock();
          DO q(c) # {} => Schedule(q(c).head); q(c):= q(c).tail OD;
     ReleaseSpinLock()

END ConditionImpl
```

The implementations of mutexes and conditions are quite similar; in fact, both are cases of a general semaphore.