Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 – Principles of Computer Systems

Handout 55 December 8, 1993

Solutions to Problem Set #6

The maximum score for this problem set was 45 points.

Please refer to Handout 30, Problem Set # 6, for the specification of `MODULE Wr`, multiple mutexes and conditions.

Problem A (35 points)

Implement the procedures in the `Wr` *module except for* `Flush`*. Treat* `Flush` *as a black box provided by someone else (but your program must provide mutual exclusion for the flush action). Your solution should satisfy the following requirements:*

- *Different threads can call* `PutChar` *concurrently on different writers and not delay each other.*
- *No character should remain unflushed in a buffer for more than 50 ms or so. To meet this requirement, invent a suitable* `Delay` *primitive. You may assume that everything except Delay executes in an amount of time that is small compared to 50 ms.*

It should be obvious how to translate your implementation into a reasonable program in C, Clu, or whatever: there should be no spin locks or repeated examination of writers that don't need to be flushed, writers should require only bounded buffers, it should be clear how to implement any sets as linked lists, etc.

Explain informally why your program works; be sure to give the invariants it maintains.

Hint: The crucial issue is what mutexes to have and in what order to acquire them. The program should not be more than a couple of pages.

In order to allow concurrent access to different writers while providing mutual exclusion to each individual writer, we introduce a mutex into each writer:

```
VAR w : SEQ [sink, buf:SEQ Char, m] := {}
% w(wr).buf and w(wr).sink are protected by w(wr).m
```

For practical reasons, we maintain an invariant on the size of each buffer:

```
% INVAR 1: (ALL wr | w(wr).buf.size <= MAXBUFSIZE)
VAR MAXBUFSIZE = ...  % some constant > 0
```

In order to allow the `Worker` thread to find the writers that need to be flushed, we introduce a set `p` (for pending) of writers containing all indices to writers with non-empty buffers:

```
% INVAR 2: (ALL wr | w(wr).buf.size # 0 ==> wr IN p)
VAR p : SET Wr := {}
```

We could awaken the `Worker` periodically even when `p` is empty, but that would not be desirable, so we introduce:

```
VAR c := C.New()
% c is signalled whenever p becomes non-empty
```

The set `p` is accessed concurrently by the `Worker` thread and by threads calling `PutChar`, so it must be protected by a mutex:

```
VAR mp := M.New()
% p is protected by mp
```

Also, for the ease of implementing the `New` procedure in SPEC, an extra mutex is added to protect `w`:[1]

```
VAR mw := M.New()
% w is protected by mw
```

Then the `New` procedure can simply be:

```
PROC New(sink) -> Wr =
    VAR wr |
        mw.acq;
        w := w ++ {sink:=sink, buf:={}, m:=M.New()};
        wr := w.size - 1;
        mw.rel;
        RET wr
```

Now comes the first real design decision: the locking order. Neither order makes everything smooth. Let's try:

[1]Since `w` is copied in the expression `w ++ entry` in `New` procedure below written in SPEC, we should acquire all the buffer locks to ensure that an alteration to one of the buffers will not be lost during the copy. However, since the end result is intended for translation into a reasonable language, and since any reasonable language has pointers and allocated spaces, so this is not necessary. For example, if while we are executing `w ++ entry`, `PutChar` to one of the already existing `wr` is called concurrently and alters `w(wr)`, the resulting `w` after its assignment `w ++ entry` includes the new character put to `w(wr)`. This is usually the case in languages like C, and this is more efficient.

```
% (ALL wr | w(wr).m < mp)
```

Then `PutChar` goes smoothly:

```
PROC PutChar(wr,char) =
    w(wr).m.acq;
    BEGIN w(wr).buf.size = MAXBUFSIZE => Flush(wr) [*] SKIP END;
    w(wr).buf := w(wr).buf ++ char; % INVAR 1: w(wr).buf.size < MAXBUFSIZE
    w(wr).buf.size = 1 =>           % just changed from empty to non-empty
        BEGIN mp.acq;
              p := p ++ wr; % INVAR 2:ALL wr | wr.buf.size # 0 ==> wr IN p
              Signal(c);
              mp.rel
        END;
    [*] SKIP;
    w(wr).m.rel
```

But the `Worker` is awkward:

```
PROC Worker() =
    DO
      mp.acq;
      DO p.isEmpty => Wait(c,mp) OD
      mp.rel;
      Delay(50);  % delay 50 ms after Worker checks that p is non-empty
      VAR  wr := Choose() |
          DO wr # -1 =>
              BEGIN w(wr).m.acq;
                    Flush(wr);
                    w(wr).m.rel END;
              wr := Choose ();
          OD;
    OD

PROC Choose() -> Wr =  % return any wr in p
    VAR wr |
        mp.acq;
        VAR wr' | wr' IN p => BEGIN wr := wr'; p := p - wr' END;
        [*] wr = -1;   % unable to find an wr in p
        mp.rel;
        RET wr
```

Then, we give the abstraction function of this implementation to the specification of `MODULE Wr`:

```
MODULE Wr

TYPE
    Sink = Int
    Wr   = Int                         % Index to w

VAR
    contents : Sink -> SEQ Char := {* -> {}}
    w        : SEQ [sink:Sink, buf:SEQ Char] := {}

PROC New(sink) -> Wr  = $N1$ << ... >> $N2$
PROC PutChar(wr,char) = $P1$ << ... >> $P2$
THREAD Worker() = DO $W1$ << ... >> $W2$ OD

END Wr
```

Putting all the implementation parts together and adding program counters, we get:

```
MODULE WrImpl

TYPE
    Sink = Int
    Wr   = Int                         % Index to w

VAR
    contents : Sink -> SEQ Char := {* -> {}}
    w        : SEQ [sink:Sink, buf:SEQ Char, m] := {}
    mw       : M := M.New()
    p        : SET wr := {}
    mp       : M := M.New()
    c        := C.New()

VAR MAXBUFSIZE : Int := ...           % some constant > 0

PROC New(sink) -> Wr =
    VAR wr |
   $n1$ mw.acq; $n2$
        w := w ++ {sink:=sink, buf:={}, m:=M.New()};
        wr := w.size - 1;
   $n3$ mw.rel; $n4$
```

```
        RET wr  $n5$

PROC PutChar(wr,char) =
$p1$ w(wr).m.acq; $p2$
    BEGIN w(wr).buf.size = MAXBUFSIZE => Flush(wr) [*] SKIP END;
    w(wr).buf := w(wr).buf ++ char; % INVAR 1: w(wr).buf.size < MAXBUFSIZE
    w(wr).buf.size = 1 =>           % just changed from empty to non-empty
        BEGIN mp.acq;
              p := p ++ wr; % INVAR 2:ALL wr | wr.buf.size # 0 ==> wr IN p
              Signal(c);
              mp.rel
        END;
    [*] SKIP;
$p3$ w(wr).m.rel  $p4$

PROC Worker() =
    DO
 $w1$ mp.acq; $w2$
      DO p.isEmpty => $w3$ Wait(c,mp) $w4$ OD;   % at w3, mp.rel and wait
                                            % at w4, mp.acq again after c is signal
      mp.rel;
 $w5$ Delay(50);  % delay 50 ms after Worker checks that p is non-empty
      VAR $w6$ wr := Choose() $w7$ $w8$ | % at w7, RHS is evaluated
                                          % at w8, RHS is binded to LHS
          DO $w9$ wr # -1 =>
              BEGIN $w10$ w(wr).m.acq; $w11$
                    Flush(wr);
                    $w12$ w(wr).m.rel END;
        $w13$ wr := Choose(); $w14$ $w15$ % at w14, RHS is evaluated
                                          % at w15, RHS is binded to LHS
          OD; $w16$
    OD $w17$

PROC Choose() -> Wr =  % return any wr in p
    VAR wr |
   $c1$ mp.acq; $c2$
        VAR wr' | wr' IN p => BEGIN wr := wr'; p := p - wr' END;
        [*] wr = -1;   % unable to find an wr in p
   $c3$ mp.rel; $c4$
        RET wr  $c5$

END WrImpl
```

Abstraction function:

1. global variables: `Wr.w` = `WrImpl.w` with the extra state discarded and `Wr.contents` = `WrImpl.contents`.
2. local variables: with INVAR 2 and INVAR 3 `wr` in `WrImpl.Worker` is the same as `wr` in both `Wr.Worker`.

```
% INVAR 3:
Choose() returns a wr in p if p is non-empty and returns -1 otherwise.
```

   In brief, the INVAR 3 is maintained through the critical section protected by `mp` in the `Choose` procedure.

3. PCs:
   - {n1,n2} $\mapsto$ N1
   - {n3,n4,n5} $\mapsto$ N2
   - {p1,p2} $\mapsto$ P1
   - {p3,p4} $\mapsto$ P2
   - {w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w13,w14,w15,c1,c2,c3,c4,c5} $\mapsto$ W1
   - {w12,w16,w17} $\mapsto$ W2
4. Transitions:
   - All external transitions in the implementation map to the corresponding external transitions in the specification (a truism).
   - (n2,n3) $\mapsto$ (N1,N2)
   - (p2,p3) $\mapsto$ (P1,P2)
   - (w11,w12) $\mapsto$ (W1,W2) % `Flush` occurs
   - (w9,w16) $\mapsto$ (W1,W2) % `[*] SKIP` occurs
   - (w12,w13) $\mapsto$ (W2,W1) % `DO` loop back, notes w15 is followed by w9
   - AF(tr) $\mapsto$ *null* for all other transitions

Problem B (5 points)

How would your program differ if it acquired the mutexes in the opposite order?

If, on the other hand, we had chosen the opposite locking order, namely

```
(ALL wr | mp < w(wr).m)
```

then the `Worker` would be easier:

```
PROC Worker() =
    DO
      mp.acq;
      DO p.isEmpty => Wait(c,mp) OD;
      mp.rel;
      Delay(50);
      mp.acq;
      DO VAR wr |
             wr IN p =>
                 BEGIN w(wr).m.acq;
                       Flush(wr);
                       w(wr).m.rel END;
             p := p - wr
      OD;
    OD;
```

while `PutChar` would become tricky:

```
PROC PutChar(wr,char) =
    VAR b |
        w(wr).m.acq;
        BEGIN w(wr).buf.size = MAXBUFSIZE => Flush(wr) [*] SKIP END;
        wr.buf := wr.buf ++ char;
        b := (w(wr).buf.size = 1) /\ ~(wr IN p);
        w(wr).m.rel;
        BEGIN ~b => SKIP
              [*] mp.acq;
                  p: = p ++ wr;
                  Signal(c)
                  mp.rel END
```

The reason this is tricky is that between the two critical sections of the `PutChar` procedure, another thread could add `wr` to `p`. Therefore, inside the second critical section, `(wr IN p)` is possibly true. If `p` is represented as a lined list, for example, "adding" something in the list to the list could produce a circular list. Therefore, with this locking order, it is probably better to add a boolean to the writers recording they are in `p` or not.

Another disadvantage of this second approach is that the `Worker` holds the global lock for a potentially long time.

Comments: Almost all students tried to explain how his or her program would behave incorrectly if its mutexes are acquired in the opposite order, instead of trying to describe how he or she would change the program so that it would work for the opposite order of mutex acquisition. But that is okay.

Comments: The answer to this question is implementation-dependent.

Problem C (5 points)

Discuss what the impact would be on your program if a sink might contain a writer inside it, and flushing to the sink could cause PutChar to that writer.

If a sink might contain a write, the program would deadlock. If a flush tries a `PutChar` to the current buffer, a deadlock will occur immediately, when `PutChar` tries to lock the buffer. We could fix this with a version of lock that let owner of a lock request the lock again with no hanging.

In part (A), the sink can be thought of as an unbounded buffer. However, in this part, the sink now contains a bounded buffer. If the buffer is full, however, we are not going to be able to call `Flush` to clear it and we will be in an infinite loop.

For a purely performance viewpoint, we would now have to be aware of the fact that a character can take about 100 ms before reaching a second level sink. The `Delay` procedure would have to be modified if this is unacceptable.

Comments: The answer to this question is implementation-dependent.