Reed and Kanodia: Eventcounts and Sequencers

- New synchronization mechanism
- Allow control of direct ordering of events
- **Eventcounts:** communication path for signaling and observing the progress of concurrent computations
- **Sequencers:** assigns an order in the events occurring in the system

- **Primitives**
  - advance(E)
    (signals an event)
  - read(E)
    (reads the value directly)
  - await (E,V)
    (blocks until E$\geq$V)

Reed and Kanodia: Eventcounts and Sequencers

- **Eventcounts**
  - alternatives to P and V
  - (e.g., one classic P/V problem: can't read the semaphore value)
  - provide a *broadcast* mechanism
- **Definition**
  - equivalent to a history variable (but these are *real* values)
  - hold to memory, not like history variables

Reed and Kanodia: Eventcounts and Sequencers

- Big deal:
  - deals with time ordering
    ensure execution of critical regions to be a total ordering (i.e., one path)
  - semaphores deal more with mutual exclusion
  - alternative synchronization mechanism
    based on observing the *sequence* of significant events during the course of an asynchronous computation

- Proof techniques
  - Shared memory: definitions of **advance, read, await** are adequate
  - Distributed memory: not adequate (due to communication delays)
  - Solution: **partial ordering** (Lamport'78)
  - e.g., proof statement: $P_i \to C_i$
    produce i before consume i
    don't consume an empty
    underflow
  - $C_i \to P_{i+N}$
    don't produce until empty
    overflow
  - Statement of no deadlock is hard

```
procedure producer()
  begin
    integer i;
    for i:=1 to infinity do
      begin
        await(OUT, i-N);
         // i-N is calculated once and
         // OUT checks that value
         buffer[i mod N] := produce( );
         advance(IN);
      end
end
procedure consumer()
  begin
   integer i;
   for i:=1 to infinity do
       begin
          await(IN, i);
          consume(buffer[i mod N]);
          advance(OUT);
       end
end
```

Example: Single Producer, Single Consumer

- N-cell ring buffer
- Two eventcounts:
  - IN and OUT
  - initially zero
- consumer does not read the ith value from the buffer until it is stored by the producer
- the producer does not store the $(i + N)$th value into the buffer until the ith value is read by the consumer

Reed and Kanodia: Eventcounts and Sequencers

- Claim: Easier proofs for time ordered properties
- Advantage:
  - Orderings are explicit
  - Separates reading of variable from waiting on it (in contrast to P and V operations)
  - Good, when only one guys awaits on an eventcounter
  - Good for "one to one" things

Reed and Kanodia: Eventcounts and Sequencers

- **Sequencers**
  Useful when arbitration is necessary
  Give total order of events
- **Definition**
  - non-decreasing function call
  - ticket dispenser
  - X ← ticket(s)
  - ticket(s) is the sequencer, function call that returns the next highest number

Building Semaphores

Eventcounts and Sequencers viewed at a lower level than semaphores

```
procedure P(S)
begin integer t;
  t := ticket(S.T);
  await(S.E, t-S.I);
end

procedure V(S)
begin
  advance(S.E)
end
```

semaphore S, eventcount: $S.E$, and the sequencer component: $S.T$, initial value of S is $S.I$.

Sequencers Example

- Multiple producers (in producer-consumer example)
- No a priori sequence constraint
- Solution: use a sequencer

```
t := ticket(T)
await(IN,t)
//wait till your turn comes
//ie, only 1 producer in CS
await(OUT, t+1-N)
// synchronize with consumer
//wait until buffer is free
buffer[(t+1) mod N] :=produce();
advance(IN);
```

- "broadcast" nature of advance
  enables a producer and a consumer

Deadlock-Free Simultaneous-P

- How to lock various resources simultaneously
- Dijkstra's Dining Philosophers
- One semaphore per fork
- Each philosopher uses a simultaneous-P

Dijkstra's Dining Philosophers

```
procedure Pboth(R,S)
begin integer g, r, s;
//first lock, coordinated ticket
//generator, global lock on getting
//tickets from sequencers R & S
g:=ticket(G.T);
await(G.E,g);
   //get a coordinated set of tickets
   r:= ticket(R.T);
   s:= ticket(S.T);
advance(G.E);
//now wait for each semaphore in turn
await(R.E, r-R.I);
await(S.E, s-S.I);
end
```

Secure Readers-Writers

```
   Procedure reader();
        begin integer w;
abort:     w:=read(S);
           await(C,w);
           "read"
           if read(S) =/= w
             then abort;
   end
   Procedure writer()
     begin integer t;
        advance(S);
        t:=ticket(T);
        await(C,t);
        "read and update"
        advance(C);
   end
```

Secure Readers-Writers

- Database accesses
- Self-consistency
- Writers *both* read and update the database
- Readers should not exclude writers
- Readers must perform all the accesses, then abort of a writer modified the database in the meanwhile

"An Optimal Algorithm for Mutual Exclusion in Computer Networks"

Ricard, Agrawala, CACM 1981

- Mutual exclusion for cooperating processes on separate machines
- Uses message passing
- Alternative to shared memory mutual exclusion
- Assumes fault-free communication
- Big Deal
  - optimality
  - 2*(N-1) messages (lower bound)

Main IDEA

- Get *every other* process to consent when you want the critical section
- Send a REQUEST composed of
  - sequence# that is higher than any sequence# you know about AND mynode#
  - if you don't want the critical section, consent to everybody else's request (REPLY)
  - if you want the critical section and you receive a request, consent to the lowest sequence#
  - if a tie, consent to the lowest mynodes#

Issues to be proved: all by contradiction

- Mutual exclusion
  - look at cases...
- Deadlock is impossible
  - no circular waiting
  - sequence# help...
- No starvation
  - No node in critical section more than twice if the local one is kept waiting

Each node has three processes to implement mutual exclusion

1. Awakened when mutual exclusion is invoked on behalf of this node
2. Receives and processes REQUEST messages
3. Receives and processes REPLY messages
   - Is the receiving node also requiring the resource?
   - Compare local sequence# to the requesting process's sequence#
   - Compare local mynode# the requesting process's node#

Other issues

- Number of messages per critical section access is minimal
  - symmetrical, fully distributed
  - 2 separate messages per node
- Sequence# can be recycled to remove overflow
  - at each time there are only a number of outstanding messages
  - can do mod M, $M > 2*(N-1)$

Modifications: reduce the number of messages

- Implicit reply
  - no response $\Rightarrow$ okay, give consent
  - else send "deferred" message
  - good for *low* contention
  - then, # messages is N-1
- Broadcast the request
  - # messages is 1 + (N-1)$\rightarrow$replies
- Combo of implicit and broadcast: 1 message
- Ring structure: N messages