PURDUE UNIVERSITY

Principles of Concurrency

Week 5

Sequential Consistency and Linearizablity

Material adapted from Herlihy and Shavit, Art of Multiprocessor Programming, Chapters 9 and 10

Consistency

How do we reason about the order in which concurrent operations are performed

- particularly relevant for shared-memory abstractions
- what view do threads have about memory? Memory model

Simplest model: strict consistency

- Maintain program order among operations from individual processors
- Enforce a global wall-clock time ordering on operations
- Every operation is sequentialized against this ordering
- But, what does "global time" even mean in a concurrent setting?

```
  T1        T2            T3
W(x,0)
          W(x,1)
                    R(x)   //L1
                    R(x)   //L2
```

Can L1 read 0, L2 read 1?
Can L1 and L2 both read 0?
Can L1 and L2 both read 1?

Sequential Consistency

Alternative model: sequential consistency

- Maintain program order among operations from individual processors
- Maintain a single interleaved execution order among operations from all processors that respects per-thread program order
  - memory operations execute atomically
  - no global clock

```
  T1        T2
W(x,1)
          R(x)   //L1
          R(x)   //L2
```

L1 can read 0 or 1
L2 can read 1 if L1 reads 0 or 1
L2 *cannot* read 0 if L1 reads 1

Key points:

- program order of individual threads must be maintained
- data coherence must be respected:
  any read must return most recent visible write

Architecturally

Processors must ensure its previous memory operation completes before starting the next one

- Memory must provide explicit acknowledgement previous write has completed
- Caches must invalidate or update all cached copies

Writes to the same location must be made visible in the same order to all processors

- serialized

Value of a write cannot be written until all invalidates or updates acknowledged

Architecturally

Executed code must contain enough synchronization to prevent incorrect reorderings

Initial: [x]=0 ∧ [y]=0	
proc 0	proc 1
MOV [x]←$1 MFENCE MOV EAX←[y]	MOV [y]←$1 MFENCE MOV EBX←[x]
Forbid: EAX=0 ∧ EBX=0	

Naive enforcement of SC is expensive (~40% overhead on x86 over well-studied benchmarks)

Initally, [100] = 0

At the end, [100] = 2

proc:0	proc:1
LOCK; INC [100]	LOCK; INC [100]

Transformations


```
T1          T2

W(x,1)      R(x) //r1
            R(x) //r2
            if (r1 == r2) {
              print 1
            } else { print 2 }
```

```
T1          T2

W(x,1)      R(x) //r1
            r2 = r1
            if (r1 == r2) {
              print 1
            } else { print 2 }
```

Is this transformation correct under SC?

```
W(x,1), R(x)1,  R(x)1,  Pr(1)
R(x)0,  W(x,1), R(x)1,  Pr(2)
R(x)0,  R(x)0,  W(x,1), Pr(1)
R(x)0,  R(x)0,  Pr(1),  W(x,1)
```

```
W(x,1), R(x)1,  Pr(1)
R(x)0,  W(x,1), Pr(1)
R(x)0,  Pr(1),  W(x,1)
```

Can transform P1 into P2, but not vise versa

Common Subexpression Elimination

- In general, not sound under SC

```
        T1                      T2

W(x,1)                    if (R(x) == 1) {
W(y,1)                       W(x,2)
if (R(y) == 2) {             W(y,2)
  print (R(x))             }
}
```

W(x,1), W(y,1), R(x)1, W(x,2), W(y,2), R(y)2, R(x)2, Pr(2)

Only one interleaving that results in "print" being called:

```
        T1                      T2

W(x,1)                    if (R(x) == 1) {
W(y,1)                       W(x,2)
if (R(y) == 2) {             W(y,2)
  print (1)              }
}
```

W(x,1), W(y,1), R(x)1, W(x,2), W(y,2), R(y)2, Pr(1)

Optimization yields different results

Another Example …


```
     T1              T2              T1(M)

R(x)*2 //r1     W(x,1)          R(x)*2 //r1
R(y)   //r2     W(y,1)          R(y)   //r2
R(x)*2 //r3                     r3 = r1
```

The transformed program (`T1(M) || T2`) can observe `r2 == 1` and `r3 == 0`, which is not possible under (`T1 || T2`)

Need to ensure that the value of x has not changed since its last read

```
      T1

L1: R(x)*2 //r1
    R(y)   //r2
    r3 = r1
    if (x modified since L1)
      R(x)*2   //r3
```

The modification check can be implemented by tracking cache coherence and invalidation messages

Transformations

- Transformations involving only thread-local variables and compiler-generated temporaries are always SC preserving.
- Some simple transformations on shared variables are also safe
  - Two consecutive loads of the same variable can be replaced by a single load: preserves SC since we only need to consider interleavings of the original program in which no other threads executes between these loads.

Redundant load: `R(x) //r1; R(x) //r2 ==> R(x) //r1; r2 = r1`
Forwarded load: `W(x,r1); R(x) //r2 ==> W(x,r1); r2 = r1`
Dead store: `W(x,r1); W(x,r2) ==> W(x,r2)`
Redundant store: `R(x) //r1; W(x,r1) ==> R(x) //r1`

Non-SC preserving transformations are those that change the order of memory accesses performed by a thread in a way that becomes visible to other threads

Constant-Folding and Copy Propagation


```
W(x,1)                    W(x,1)
R(z) //r1      ==>        R(z) //r1
W(y, r1)                  W(y, z)
R(x) //r2                 r2 = 1
```

Is there a concurrent context in which transformed program exhibits a behavior not possible in the original?

Consider:

```
if (R(x) == 1) {
    W(x,3);
    W(z,2)
} else {
    W(z,4)
}
```

Original:

```
r1 = 0, r2 = 1
r1 = 4, r2 = 1
r1 = 0, r2 = 3
r1 = 2, r2 = 3
```

Transformed:

```
r1 = 0, r2 = 1
r1 = 4, r2 = 1
r1 = 0, r2 = 1
r1 = 2, r2 = 1
```

Basic Idea

- Sequential consistency reasons about concurrent actions by enforcing a global ordering over thread operations
- Linearizability is a property of a concurrent program expressed in terms of time, not order
- Both definitions reduce concurrent actions to some form of sequential execution

Linearizability

- The state of an object can be manipulated by its methods concurrently
- Two methods that do not overlap in a history are always ordered with respect to global time (happens-before relation)
- Two methods that do overlap in a history are ordered in a way that preserves the illusion that the object is sequential (i.e., not manipulable by its methods concurrently)
  - Reading an object should reflect the effect of the most recent write, regardless of the thread performing the read

Complications

- Definition appeals to notion of global time
- Concurrently executing methods can overlap

Linearizability

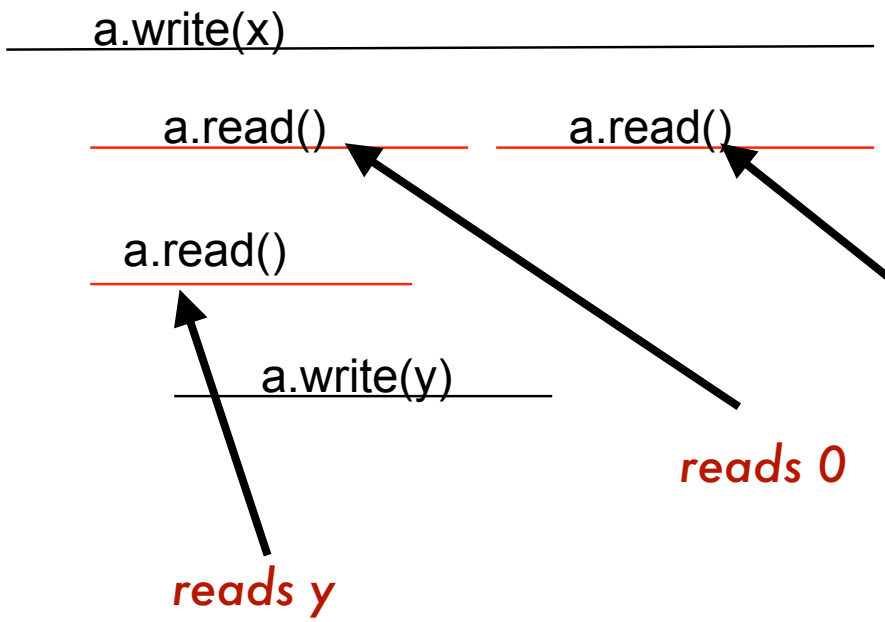
Non-overlapping actions

a.write(x)

a.read()

a.read()

Overlapping actions



Need to enforce:

- read returns result of most recent write
- once effect of a write becomes visible, all subsequent reads must return the value, until a new write completes

what value can be read here if the execution was linearizable?

Example


```
q = new ConcurrentQueue()

T1:                          T2:
q.add(10)                    t = q.remove()
```

After execution, can assert that either

```
t = fail  && q.size = 1 or
t = 10) && q.size = 0)
```

```
q = new ConcurrentQueue()

T1:                          T2:
q.add(10)                    q.add(20)
t = q.rem()                  t = q.remove()
```

After execution, can assert that:

```
q.size = 0 &&
(t = 10  && u = 20) ||
(t = 20  && u = 10)
```

Specifications

Sequential:

- If the state of an object is S before method M is called
- Then, its state becomes S′ after M is called

Example:

- If the queue Q = x.xs before a dequeue operation is invoked
- Then, the queue after the operation completes is Q′ = xs

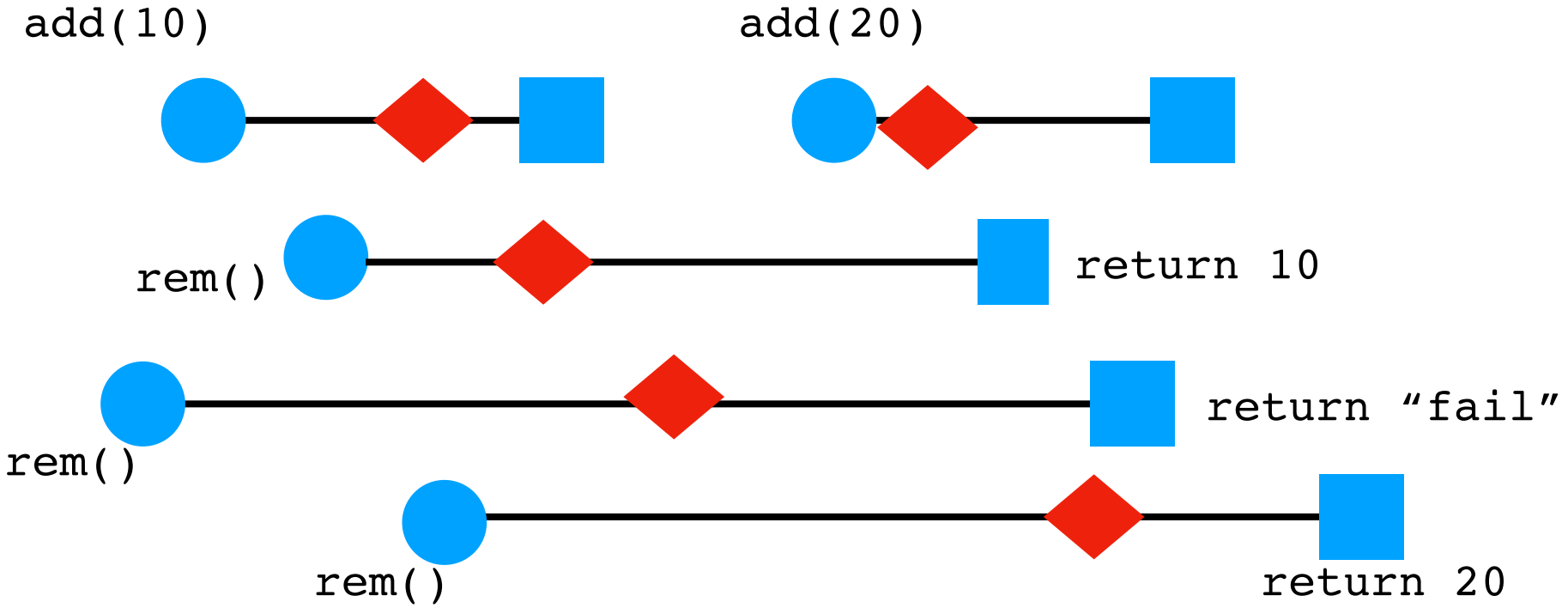
Specifications describe each method in isolation

- Interactions among methods captured by changes to state
- Obvious point where side-effects become apparent:
  it is only when a method returns that other calls to the object can witness its effects
  single linear order of actions applied to an object

A linearizable concurrent data structure should exhibit behavio consistent with its sequential specification

Linearizability

An object is linearizable if in all its executions, the effects of its methods appear to take place atomically at a single temporal point between the method's call and return



Linearizability Manifesto

- Each method call should appear to take place instantaneously at some moment between its invocation and response
  - This is the method's linearization point
- Avoids the need to describe all concurrent interactions
- Can still use pre- and post- conditions to describe behavior
- Example: Enqueuing two objects (say x and y) concurrently onto an empty queue will result in a dequeue reading one of x or y, but not some arbitrary element z
- Properties:
  - Locality: a system is linearizable if each individual object is linearizable
  - Non-blocking: linearizable methods never need to block

Concurrent Histories

Sequence of events comprising invocations and returns

```
<T1, q, Invk, add, 10>
<T2, q, Invk, rem>
<T1, q, Ret, add, void>
<T2, q, Ret, rem, 10>
```

Assume histories are complete - every invocation has a return

Two histories are equivalent if

- threads perform operations in the same order
- the return values they observe are the same

Two operations in a history are concurrent if their intervals overlap:

(op1.Invk < op2.Ret) && (op2.Invk < op1.Ret)

Sequential Specification

A history is serial if every invocation is immediately followed by a return

The set of all serial histories defines the sequential behavior of the object

```
<T1, q, Invk, add, 10>
<T1, q, Ret, add, void>
<T2, q, Invk, rem>
<T2, q, Ret, rem, 10>
```

Linearizability

A concurrent history is linearizable if it is equivalent to a serial history in the sequential specification

This means that all operations that take place "before" in the concurrent history also take place "before" in the serial one

Checking Linearizability

- Testing/Model Checking:
  - Generate random concurrent scenarios
  - Verify that these executions match some sequential interleaving:
    * explore all possible (upto a bound) such interleavings
- Verification
  - Identify linearization points in an implementation
  - Prove that the effects performed at these points match the effects of the sequential specification
  - This can be complicated:
    * They may depend on complex control-flow
    * They may even be found in another method (e.g., consider helping mechanisms)

Checking Linearizability

– Distinguish between pure and effectful executions of abstract operations:

- A pure operation does not modify the abstract state
- An effectful operation does

- Sequential Specification

```
enqueue (item):
    Q = Q ++ [item]
```

```
dequeue():
   if isEmpty(Q)
       return EMPTY
   else {  result = Q.hd
           Q = Q.tl
           return result
        }
```

Checking Linearizability


```
typedef struct Node_s *Node;

struct Node_s {
  int val;
  Node tl;
}

struct Queue {
  Node head, tail;
} *Q;

void enqueue(int value) {
  Node node, next, tail;
  node = new_node();
  node->val = value;
  node->tl = NULL;
  while(true) {
    tail = Q->tail;
    next = tail->tl;
    if (Q->tail != tail) continue;
    if (next == NULL) {
      if (CAS(&tail->tl,next,node))
        break;
    } else {
      CAS(&Q->tail,tail,next);
    }
  }
  CAS(&Q->tail,tail,node);
}
```

Not conditional

```
void init(void) {
  Node node = new_node();
  node->tl = NULL;
  Q = new_queue();
  Q->head = node;
  Q->tail = node;
}

int tryDequeue(void) {
  Node next, head, tail;
  int pval;
  while(true) {
    head = Q->head;
    tail = Q->tail;
    next = head->tl;
    if (Q->head != head) continue;
    if (head == tail) {
      if (next == NULL)
        return EMPTY;
      CAS(&Q->tail,tail,next);
    } else {
      pval = next->val;
      if (CAS(&Q->head,head,next))
        return pval;
    }
  }
}
```

Conditional - it is a linearization point only if `Q->head != head` *is false; it is a linearization point when the queue is empty*

But, when the condition fails, the logical state is not modified

Concrete Implementation