LOCKING PROTOCOLS & SOFTWARE TRANSACTIONAL MEMORY

Why Study (Software) Transactions

Why Study (Software) Transactions

1. Transactions are good abstractions

Why Study (Software) Transactions

1. Transactions are good abstractions
2. Design choices and issues can guide your concurrent data structures

Coarse/Fine Grained Locking

- Locking discipline enforce protection
  - Each object is protected by some lock
  - One lock protects a set of objects
- Lock leveling for deadlock avoidance
  - Assign a partial ordering to locks, Level(X)
  - After acquiring a lock X, the transaction is only allowed to acquire locks Y with Level(Y) > Level(X)
- Enforce locking discipline and lock leveling using static analysis tools

Two Phase Locking (Revisited)

1. Transactions acquire a lock on an object before accessing the object
2. Transactions cannot acquire a lock once the transaction has released some lock

- Works with shared/exclusive locks
- Deadlock avoidance

Use Hardware Primitives

```
Lock (L);
x = x + 1;
Unlock(L);
```

```
InterlockedIncrement(&x);
```

- Interlocked operations are implemented by low-level locking by the hardware

```
asm{
  lock inc [x];
}
```

Using Compare And Exchange

- If you need to protect more than one logical variables
- But the variables can be compressed into a 32/64 bits

```
Lock(L); // protects x and y

x = f(x, y);
y = g(x, y);

Unlock(L);
```

Using Compare and Exchange

- Compress x and y into a single 'state' variable

```
retry:
s = state; // compress state into 64 bits

x = s.x; y = s.y;
x = f(x,y); y = g(x,y)
t.x = x; t.y = y;

if(! CompareAndExchange(&state, t, s))
  goto retry;
```

Progress Guarantees

- Wait freedom
  - Every thread makes progress in the presence of conflicts
- Lock freedom
  - Some thread makes progress in the presence of conflicts
- Obstruction freedom
  - A thread makes progress when all other threads are suspended

What is the progress guarantee?

```
Lock(L);

x = f(x, y);
y = g(x, y);

Unlock(L);
```

```
Lock(L);

x = f(x, y);
y = g(x, y);

Unlock(L);
```

What is the progress guarantee?

```
retry:
s = state;
x = s.x; y = s.y;
x = f(x,y);
y = g(x,y)
t.x = x; t.y = y;

if(!CAS(&state,t,s))
  goto retry;
```

```
retry:
s = state;
x = s.x; y = s.y;
x = f(x,y);
y = g(x,y)
t.x = x; t.y = y;

if(!CAS(&state,t,s))
  goto retry;
```

Sophisticated Fine Grained Locking Protocols

- Locking protocols for guaranteeing linearizability
  - Even when not following the 2-phase locking discipline
- Allows early release of locks for more concurrency
- Proof of correctness is complicated
  - Never roll out your own

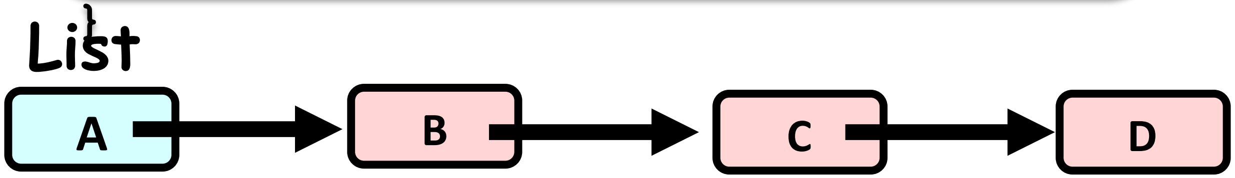
Tree Locking

- The objects form a tree
- Transactions acquire a lock on an object before accessing the object
  - Henceforth not state this constraint explicitly

1. A transaction may begin executing by locking any object
2. Can acquire a lock on X only if already holding the lock on parent of X
3. Can acquire a lock on X at most once
4. (Can otherwise release locks in arbitrary order)

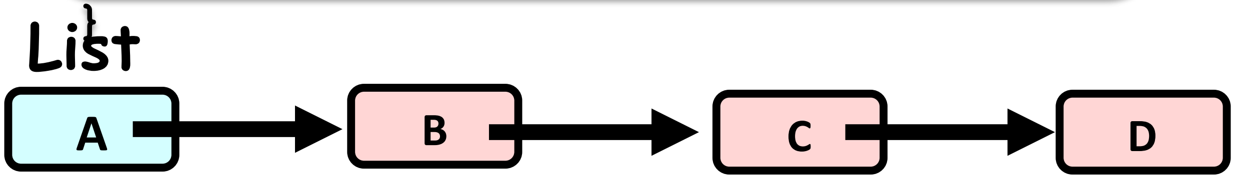
"Hand Over Hand" Locking

```
void IncrementAll(List list, int k)
{
  Node c = list;

  while( c != null ) {
    c.data += k;


    c = c.next ;

  }
}
```



"Hand Over Hand" Locking

```
void IncrementAll(List list, int k)
{
  Node c = list;
  Lock(c);
  while( c != null ) {
    c.data += k;
    if(c.next != null)
       Lock(c.next);
    c = c.next ;
    Unlock(c);
  }
}
```



Proof of Tree Locking

- If two transactions conflict, one of the two acquires all shared locks before the other
- This ordering enforces the serialization order
  - => Serializability
- Is this protocol linearizable?

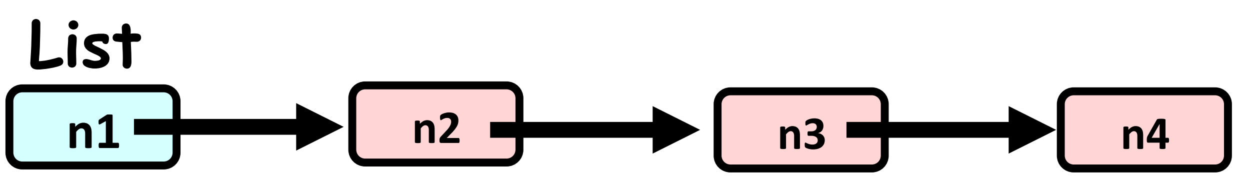
DAG Locking

- The objects form a DAG

1. A transaction may begin executing by locking any object
2. Can acquire a lock on X only if
    1. already holding the lock on some parent of X
    2. Has previously acquired locks on all parents of X
3. Can acquire a lock on X at most once
4. (Can otherwise release locks in arbitrary order)

Domination Locking

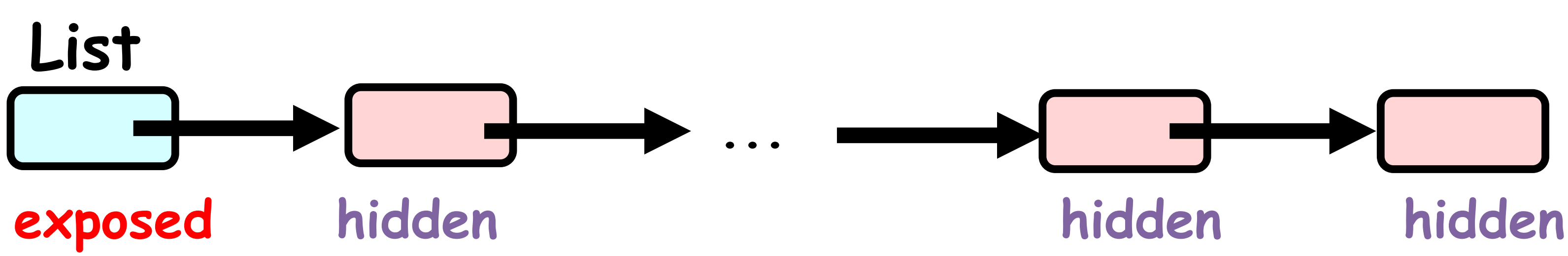
- Leverages the restricted semantics of software modules
  - thread can access n3 only after n1 & n2
- Allows early release
- Allows for dynamic modification to the underlying struture



Two types of objects

- Distinguishes between exposed and hidden objects
- Exposed objects
  - “roots” of data structures
  - may be pointed by transaction arguments
- Hidden objects
  - may not be pointed by transaction arguments
  - may be reachable via exposed objects

void insert(List l, int k) {…}



Domination Protocol Rules

1. A hidden object u can be acquired by t only if every path between an exposed object to u includes an object which is locked by t
2. Perform leveled two phase locking on exposed objects

Software Transactional Memory

- Different tradeoffs than database transactions
- Disk access ~10ms vs Memory access ~10ns
- Durability (the D in ACID is not necessary)
- Databases can control all access to the data, while STMs should deal with non-transactional accesses

Pessimism vs Optimism

- Pessimistic concurrency control
  - Expect conflict
  - Prevent conflict (by holding locks)
  - Need to avoid deadlocks
  - Useful when conflicts are frequent
- Optimistic concurrency control
  - Expect no conflicts
  - Rollback and retry
  - Need to avoid livelocks, wasted work on retry
  - Useful when conflicts are rare
- Combination possible

What is the concurrency control in TL2?

Version Management : How to perform writes

- "Eager" writes
  - Perform writes in place
  - Maintain undo-logs to revert writes on abort
- "Lazy" writes
  - Perform writes in a local log
  - Write to memory at commit time
  - Need to guarantee read-your-own-writes

Dealing with inconsistent reads

```
atomic {
  X = 1;



  Y = 1;
}
```

```
atomic {
  if(X != Y)
    while(1);
}
```

Obstruction Freedom: Is it acceptable to hold locks?

- Long running transactions can block progress ?
- A switched out thread can block progress ?
- A failed thread can block progress ?
- Priority inversion?

Optimisitic Concurrency For Reads

- Work loads tend to be read mostly
- Should allow efficient concurrent reads
- Look out for writes introduced in a read-only transaction

TL2 Algorithm

```
read_ver = global_clock;
on_read(x):
  (x_ver, x_locked) = version(x);
  validate_read(x):
     if(x_locked || x_ver > read_ver) abort;
on_write(x):
   write (x,v) to log

forall w in write_set lock(w);
atomic{ write_ver = ++global_clock; }
forall r in read_set valid_read(r);
forall w in write_set{
      version(w) = write_ver;
      unlock(w);
}
```