Synchronization Primitives

Locks

```
synchronized (lock) { balance += amt; }
```

Messages

```
 (thread server)
 ... (channel-put deposit-ch amt) ...
```

Transactions

```
     atomic { balance += amt; }
```

Transactions

`atomic` marks a set of actions to appear to happen instantaneously to all other processes

Instead of stopping other processes, let everyone run until non-instantaneous state is detected

This potential problem is called a ***conflict***

Hide the problem by discarding/rewinding changes and trying again later

This is called an ***abort***

If there was no problem, then make the changes permanent

This is called a ***commit***

Transactions

Process 1

```
atomic {
  a++;
  b++;
  c++;
}
```

Process 2

```
atomic {
  d++;
  e++;
  f++;
}
```

No conflict: processes 1 and 2 run completely in parallel

Transactions

Process 1

```
atomic {
  a++;
  b++;
  c++;
}
```

Process 2

```
atomic {
  d++;
  b++;
  f++;
}
```

One process may have to retry its transaction

Transactions

Process 1

```
atomic {
  a++;
  b++;
  c++;
}
```

Process 2

```
atomic {
  d++;
  e = b;
  f++;
}
```

Depends on transaction implementation

Multiple Data

Locks (and deadlock)

```
synchronized (lockA) {
  synchronized (lockB) {
    a.op(b);
    b.op(a);
  }
}

synchronized (lockB) {
  synchronized (lockA) {
    ...
  }
}
```

Multiple Data

Messages (and multiple managers)

```
(define (a-server ...)
  (sync
   (handle-evt a-request-ch
               ...)))

(define (b-server ...)
  (sync
   (handle-evt b-request-ch
               ...)
   (handle-evt a+b-request-ch
               ... a-request-ch ...)))
```

Multiple Data

Transactions (no problem)

```
atomic {
  a.op(b);
  b.op(a);
}
```

Transactions can fix deadlock and priority inversion

Waiting

Locks

```
lock.lock();

while (q.isEmpty())
  nowFull.await();
result = q.dequeue();

lock.unlock();
```

Waiting

Messages

```
...
(sync
 (if (empty? queue)
     never-evt
     (channel-put-ev dequeue-ch
                     (first queue))))

... (channel-get dequeue-ch) ...
```

Waiting

Transactions

```
atomic {
  if (q.isEmpty())
    retry;
  result = q.dequeue();
}
```

`retry` means “try again when something changes”

Implementing Transactions

Eager implementation:

- Perform a write immediately, but remember old value
- On abort, rewind changes (block other processes)
- On commit, discard old values

⇒ transaction commits quickly

Lazy implementation:

- Remember pending writes, and use them for re-reads within the transaction
- On abort, discard changes (other processes continue)
- On commit, perform pending writes

⇒ transaction aborts quickly

Implementing Transactions

Pessimistic implementation:

- Watch for conflicts during transaction

⇒ abort early to avoid wasted work

Optimistic implementation:

- Check for conflicts just before commit

⇒ lower overall overhead

Issues with Transactions

Transactions only work with actions that are undoable or immediate — which does not include I/O

If a transaction is too long:

- Read/write logs grow large
- The transaction may be constantly interrupted

Tracking reads and writes to detect conflicts can incur significant overhead