Concurrency and asynchronous computing

- How do we deal with evaluation when we have a bunch of processors involved?

Concurrency and asynchronous computing

- Object oriented approaches lose "referential transparency"
  - **Referential transparency** means equal expressions can be substituted for one another without changing the value of the expression

Example of referential transparency

```
(define (make-adder n)
   (lambda (x) (+ x n))

(define D1 (make-adder 4))

(define D2 (make-adder 4))
```

Are D1 and D2 the same?

- Different procedural objects
- But can replace any expression with D1 by same expression with D2 and get same value – so **YES**

Example of loss of transparency

```
(define (make-account balance)
   (define (withdraw amount)
      (if (>= balance amount)
          (begin (set! Balance (- balance amount))
                 balance)
          "Insufficient funds"))
   (define (deposit amount)
      (set! Balance (+ balance amount)))
   (define (dispatch m)
      (cond ((eq? M 'withdraw) withdraw)
            ((eq? M 'deposit) deposit)
            ((eq? M 'balance) balance)
            (else (error "unknown request" m))))
   dispatch)

(define peter (make-account 100))
(define paul (make-account 100))
```

} **Are these the same?**

The role of time in evaluation

```
(d1 5)
;Value: 9

(d1 5)
;Value: 9
```

Order of evaluation doesn't matter

```
((peter 'deposit) 5)
;Value: 105

((peter 'deposit) 5)
;Value: 110
```

Order of evaluation does matter

Role of concurrency and time

- Behavior of objects with state depends on sequence of events that precede it.
- Objects don't change one at a time; they act concurrently.
- Computation could take advantage of this by letting processes run at the same time
- But this raises issues of controlling interactions

Why is time an issue?

```
(define peter (make-account 100))
(define paul peter)

((peter 'withdraw) 10)
((paul 'withdraw) 25)
```

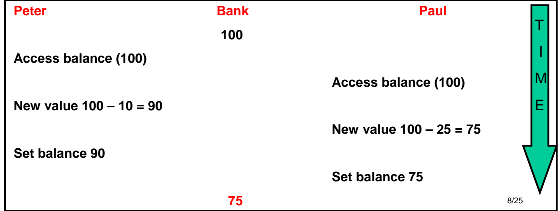
Peter	Bank	Paul
	100	
-10	90	
	65	-25

Peter	Bank	Paul
	100	
	75	-25
-10	65	

Why is time an issue?

```
(define (withdraw amount)
      (if (>= balance amount)
          (begin (set! Balance (- balance amount))
                 balance)
          "Insufficient funds"))

((peter 'withdraw) 10)
((paul 'withdraw) 25)
```



Correct behavior of concurrent programs

- REQUIRE
  - That no two operations that change any shared state variables can occur at the same time
  - That a concurrent system produces the same result as if the processes had run sequentially in some order
    - Does not require the processes to run sequentially, only to produce results **as if** they had run sequentially
    - There may be more than one "correct" result as a consequence!

Parallel execution

```
(define x 10)
(define p3 (lambda () (set! X (* x x))))
(define p4 (lambda () (set! X (+ x 1))))

(parallel-execute p3 p4)

P1:   a: lookup first x in p3
      b: lookup second x in p3
      c: assign product of a and b to x
P2:   d: lookup x in p4
      e: assign sum of d and 1 to x
```

a b c d e	a d b e c
a b d c e	d a b e c
a d b c e	a d e b c
d a b c e	d a e b c
a b d e c	d e a b c

Parallel execution

```
(define x 10)
(define p3 (lambda () (set! X (* x x))))
(define p4 (lambda () (set! X (+ x 1))))

(parallel-execute p3 p4)

P1:   a: lookup first x in p3
      b: lookup second x in p3
      c: assign product of a and b to x
P2:   d: lookup x in p4
      e: assign sum of d and 1 to x
```

a b c d e	10 10 100 100 101	a d b e c	10 10 10 11 100
a b d c e	10 10 10 100 11	d a b e c	10 10 10 11 100
a d b c e	10 10 10 100 11	a d e b c	10 10 11 11 110
d a b c e	10 10 10 100 11	d a e b c	10 10 11 11 110
a b d e c	10 10 10 11 100	d e a b c	10 11 11 11 121

Parallel execution

```
(define x 10)
(define p3 (lambda () (set! X (* x x))))
(define p4 (lambda () (set! X (+ x 1))))

(parallel-execute p3 p4)

P1:   a: lookup first x in p3
      b: lookup second x in p3
      c: assign product of a and b to x
P2:   d: lookup x in p4
      e: assign sum of d and 1 to x
```

a b c d e	10 10 100 100 101	a d b e c	10 10 10 11 100
a b d c e	10 10 10 100 11	d a b e c	10 10 10 11 100
a d b c e	10 10 10 100 11	a d e b c	10 10 11 11 110
d a b c e	10 10 10 100 11	d a e b c	10 10 11 11 110
a b d e c	10 10 10 11 100	d e a b c	10 11 11 11 121

Serializing access to shared state

- Serialization:
  - Processes will execute concurrently, but there will be distinguished sets of procedures such that only one execution of a procedure in each serialized set is permitted to happen at a time.
  - If some procedure in the set is being executed, then a process that attempts to execute any procedure in the set will be forced to wait until the first execution has finished.
  - Use serialization to control access to shared variables.

Serializers to "mark" critical regions

- We can mark regions of code that cannot overlap execution in time. This adds an additional constraint to the partial ordering imposed by the separate processes.
- Assume `make-serializer` takes a procedure as input and returns a serialized procedure that behaves like the original procedure, expect that if some other procedure in the same serialized set is underway, this procedure must wait for that process' completion before beginning.

Serialized execution

```
(define x 10)
(define mark-red (make-serializer))
(define p5 (mark-red (lambda () (set! X (* x x)))))
(define p6 (mark-red (lambda () (set! X (+ x 1)))))

(parallel-execute p5 p6)

P1:   a: lookup first x in p5
      b: lookup second x in p5
      c: assign product of a and b to x
P2:   d: lookup x in p6
      e: assign sum of d and 1 to x
```

a b c d e 10 10 100 100 101

d e a b c 10 11 11 11 121

Serializing access to a shared state variable

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! Balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! Balance (+ balance amount)))
  (let ((protected (make-serializer)))
   (define (dispatch m)
     (cond ((eq? M 'withdraw) (protected withdraw))
           ((eq? M 'deposit) (protected deposit))
           ((eq? M 'balance) balance)
           (else (error "unknown request" m))))
   dispatch))

(define peter (make-account 100))
(define paul peter)
```


Multiple shared resources

- Swapping money between accounts

```
(define (exchange account1 account2)
  (let ((difference (- (account1 'balance)
                       (account2 'balance))))
    ((account1 'withdraw) difference)
    ((account2 'deposit) difference)))
```

A1 = 300 A2 = 100 A3 = 200

(exchange a1 a2) & (exchange a2 a3)

1. Difference a1 & a2 = d1
2. Withdraw d1 from a1
3. Deposit d1 to a2
4. Difference a1 & a3 = d2
5. Withdraw d2 from a1
6. Deposit d2 to a3

1 → d1 = 200
4 → d2 = 100
5 → a1 = 200
6 → a3 = 300
2 → a1 = 0
3 → a2 = 300

Locking out access to shared state variables

```
(define (make-account-with-serializer balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! Balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! Balance (+ balance amount)))
  (let ((balance-serializer (make-serializer)))
   (define (dispatch m)
     (cond ((eq? M 'withdraw) withdraw)
           ((eq? M 'deposit) deposit)
           ((eq? M 'balance) balance)
           ((eq? M 'serializer) balance-serializer)
           (else (error "unknown request" m))))
   dispatch))
```

Serialized access to shared variables

```
(define (deposit account amount)
  (let ((s (account 'serializer))
        (d (account 'deposit)))
    ((s d) amount)))

(define (serialized-exchange acct1 acct2)
   (let ((serializer1 (acct1 'serializer))
         (serializer2 (acct2 'serializer)))
     ((serializer1 (serializer2 exchange))
       acct1
       acct2)))
```


Deadlocks

- Suppose Peter attempts to exchange a1 with a2
- And Paul attempts to exchange a2 with a1

- Imagine that Peter gets the serializer for a1 at the same time that Paul gets the serializer for a2.
- Now Peter is stalled waiting for the serializer from a2, but Paul is holding it.
- And Paul is similarly waiting for the serializer from a1, but Peter is holding it.

- This “deadly embrace” is called a **deadlock.**

Implementing serializers

- We can implement serializers using a primitive synchronization method, called a **mutex.**
- A mutex acts like a semaphore flag:
  - Once one process has acquired the mutex (or run the flag up the flagpole), no other process can acquire the mutex until it has been released (or the flag has been run down the flagpole).
  - Thus only one of the procedures produced by the serializer can be running at any given time. All others have to wait for the mutex to be released so that they can acquire it and block out competing processes.

A simple Scheme Mutex

```
(define (make-serializer)
  (let ((mutex (make-mutex)))
    (lambda (p)
       (define (serialized-p . Args)
          (mutex 'acquire)
          (let ((val (apply p args)))
             (mutex 'release)
             val))
       serialized-p)))
```


A simple Scheme Mutex

```
(define (make-mutex)
   (let ((cell (list #f)))
      (define (the-mutex m)
         (cond ((eq? M 'acquire)
                (if (test-and-set! Cell)
                    (the-mutex 'acquire)))
               ((eq? M 'release)
                (clear! Cell))))
      the-mutex))
(define (clear! Cell)
  (set-car! Cell #f))
```


Implementing test-and-set!

```
(define (test-and-set! Cell)
    (if (car cell)
        #t
        (begin (set-car! Cell #t)
               #f)))
```

This operation must be performed atomically, e.g. directly in the hardware!!

Concurrency and time in large systems

- Can enable parallel processes by judiciously controlling access to shared variables
- In essence this defines a notion of atomic actions, which must be initiated and completed before other actions may proceed.
- Careful programming leads to inefficient processing, while ensuring correct behavior
- Ultimately concurrent processing inherently requires careful attention to communication between processes.

25/25