SICP 3.4

CONCURRENCY: TIME IS OF THE ESSENCE

Withdrawing from a bank account

```
(withdraw 25)
75

(withdraw 25)
50
```

Seems easy, right?

For any events A and B

- either A occurs before B
- A and B are simultaneous
- or A occurs after B

EXAMPLE

Peter withdraws $10 and Paul withdraws $25 from a joint account that initially contains $100, leaving $65 in the account. Depending on the order of the two withdrawals, the sequence of balances in the account is either:

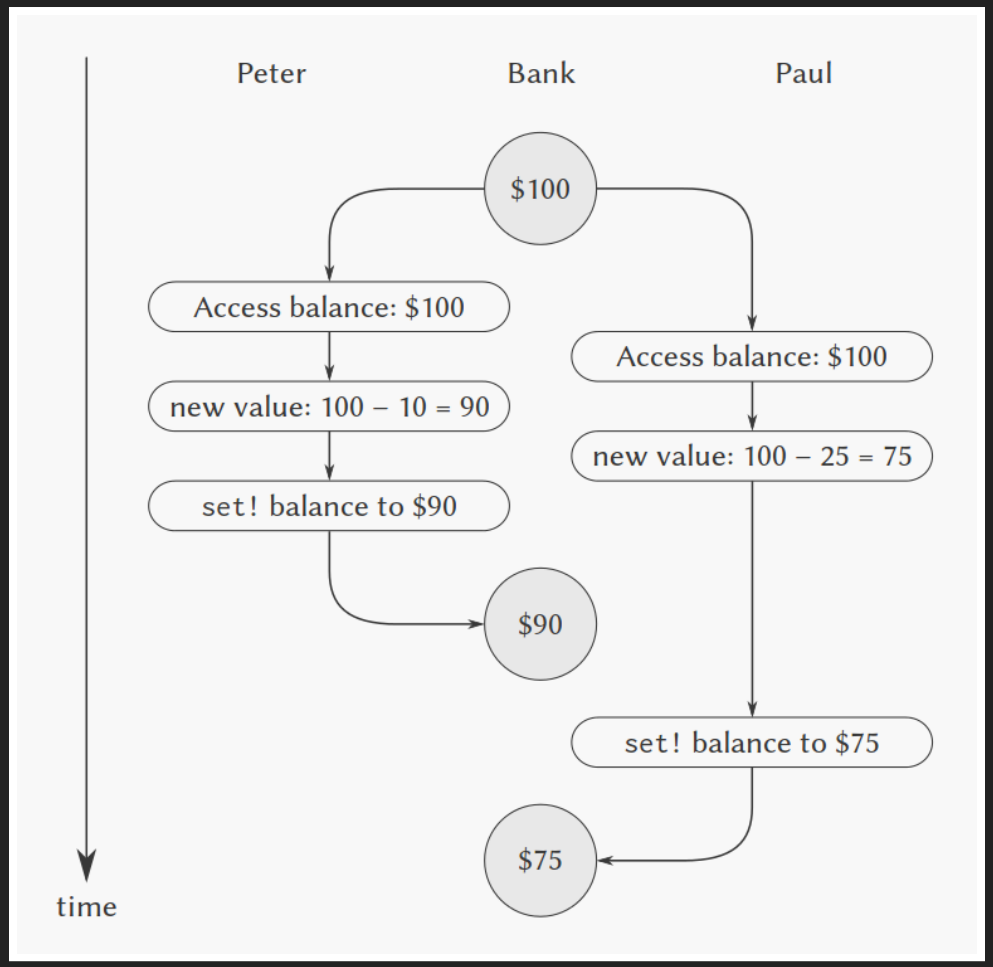
- $100 → $90 → $65
- $100 → $75 → $65

  This is the best case senerio then the two events are not simultaneous.

What happens when events are simultaneous?

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

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


Peter
Bank
Paul
$100
Access balance: $100
new value: 100 – 10 = 90
set! balance to $90
$90
Access balance: $100
new value: 100 – 25 = 75
set! balance to $75
$75
time

What went wrong?

```
(set! balance (- balance amount))
```

3.4.2 Mechanisms for Controlling Concurrency

Two events:

- 1: (a,b,c)
- 2: (x,y,z)

How many possible event combinations are there?

20 possible event combinations!

```
(a,b,c,x,y,z)  (a,x,b,y,c,z)  (x,a,b,c,y,z)
(x,a,y,z,b,c)  (a,b,x,c,y,z)  (a,x,b,y,z,c)
(x,a,b,y,c,z)  (x,y,a,b,c,z)  (a,b,x,y,c,z)
(a,x,y,b,c,z)  (x,a,b,y,z,c)  (x,y,a,b,z,c)
(a,b,x,y,z,c)  (a,x,y,b,z,c)  (x,a,y,b,c,z)
(x,y,a,z,b,c)  (a,x,b,c,y,z)  (a,x,y,z,b,c)
(x,a,y,b,z,c)  (x,y,z,a,b,c)
```

SERIALIZING ACCESS TO SHARED STATE

Processes will execute concurrently, but there will be certain collections of procedures that cannot be executed concurrently

We can use serialization to control access to shared variables.

SERIALIZERS IN SCHEME

```
(parallel-execute ⟨p1⟩
                  ⟨p2⟩
                  …
                  ⟨pk⟩)
```

Using parallel-execute

```
  (define x 10)
(parallel-execute (lambda () (set! x (* x x)))
                  (lambda () (set! x (+ x 1))))
```

This creates two processes

- P1: X*X
- P2: X++

How many possible values are there?

```
(define x 10)
      (parallel-execute (lambda () (set! x (* x x)))
                        (lambda () (set! x (+ x 1))))
```

value of x	
101	p1 sets x to 100, p2 increments to 101
121	p2 increments to 11, p1 multiplies x*x
110	p2 changes x to 11 while p1 accesses x, 11*10

11	p2 accesses x, p1 calculates x*x, p2 sets x to 11
100	p1 accesses x twice, p2 sets x to 11, p1 sets x

Using make-serializer

```
(define x 10)
(define s (make-serializer))
(parallel-execute
 (s (lambda () (set! x (* x x))))
 (s (lambda () (set! x (+ x 1)))))
```

Eliminates the two cases where the value of x changes between lookups. Possible values are now 101, and 121.

Using make-serializer

```
(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))
    balance)
  (let ((protected (make-serializer)))
    (define (dispatch m)
```

```
(cond ((eq? m 'withdraw)
       (protected withdraw))
      ((eq? m 'deposit)
       (protected deposit))
```

COMPLEXITY OF USING MULTIPLE SHARED RESOURCES

Using serializers is relatively straightforward when there is only a single shared resource (such as a single bank account), concurrent programming can be treacherously difficult when there are multiple shared resources

difficult when there are multiple shared resources

IMPLEMENTING SERIALIZERS

- Mutex: control single access to a shared resource. You aquire and release a 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)))
```

- Semaphore: limit access to n concurrent users

IMPLEMENTING SERIALIZERS

Demo: shared resources in perl

DEADLOCK

Peter’s process entered a serialized procedure protecting a1

Paul’s process enters a serialized procedure protecting a2

Now Peter cannot proceed until Paul exits the serialized procedure protecting a2.

Paul cannot proceed until Peter exits the serialized procedure