MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1998

Lecture Notes – November 3, 1998

Concurrency

Bank Account (Message Passing)

```
(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)
```

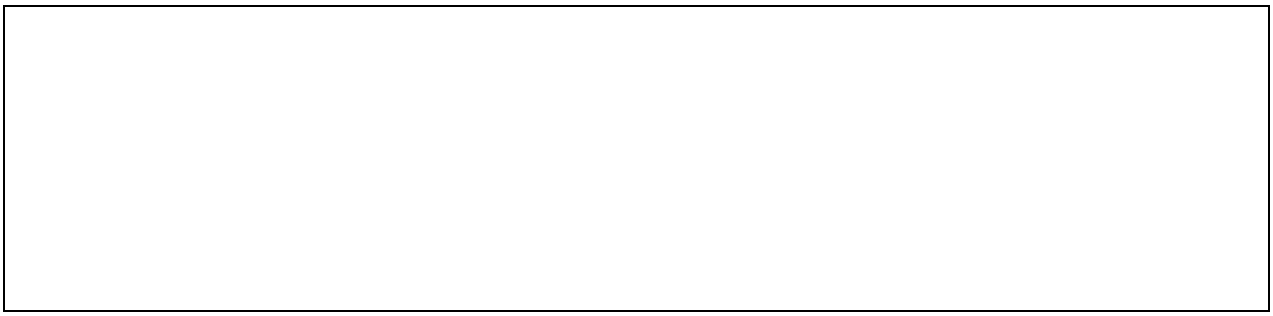
Create a *shared* bank account:

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

Consider what might happen if the following two processes occur concurrently:

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

Problem:



Parallel Execution Example

```
(define x 10)

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

(parallel-execute p3 p4)

a: lookup first x in p3
b: lookup second x in p3
c: assign product of a and b to x

d: lookup x in p4
e: assign sum of d and 1 to x
```

Possible results – consistent with **partial orderings** of p3 & p4: 101, 121, 110, 11, 100

Possible results – consistent with any **sequential ordering** of p3 & p4 – that is, no interleaving of the parts within p3 & p4: {p3 p4} => 101 and {p4 p3} => 121

Approach: Serializers to "Mark" Critical Regions

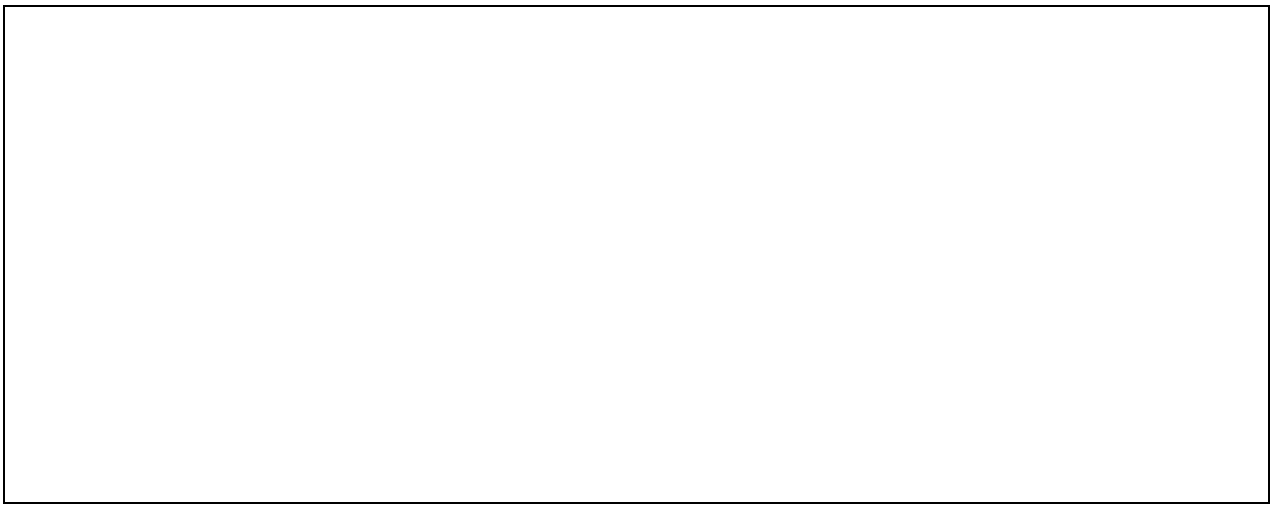
We can mark *critical regions* of code that cannot overlap execution in time. This adds an additional *constraint* to the *partial ordering* imposed by the separate processes.

```
(define mark-red (make-serializer))

(define (p1)
  a ((mark-red (lambda () b c d))) e f g h)

(define (p2)
  J K ((mark-red (lamda () L M N))) O)

(parallel-execute p1 p2)
```



Serializers in Bank Example

```
(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 ((marker (make-serializer)))
    (define (dispatch m)
      (cond ((eq? m 'withdraw) (marker withdraw))
            ((eq? m 'deposit) (marker deposit))
            ((eq? m 'balance) balance)
            (else (error "unknown request" m))))
    dispatch))
```

Deadlock

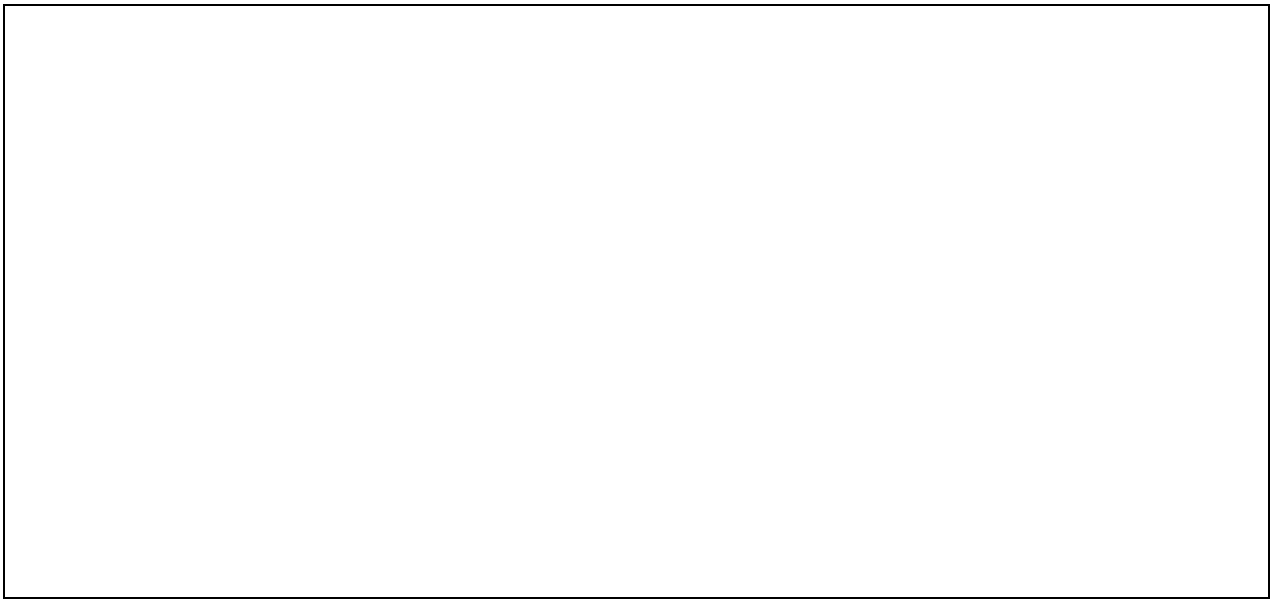
Even with serializers, it is still possible to have difficulties with concurrent processes:

```
(define mark-red (make-serializer))
(define mark-blue (make-serializer))

(define (p1)
  ((mark-blue (lambda () a ((mark-red (lambda () b c d e))) f))))

(define (p2)
  ((mark-red (lambda () J K ((mark-blue (lambda () L M N))) O P))))

(parallel-execute p1 p2)
```



A Simplified Serializer

Analogy: Multiple speakers in a room. How does a speaker succeed in making his speech?

1. He will first check for empty microphone
2. If no-one is speaking, he grabs the microphone
3. He says what he wants to say
4. When done he gives up the microphone.

Code Attempt #1:

```
(define (mark-red some-speech)
  (if (red-microphone-available?)
      (begin (grab-red-microphone)
             (some-speech)
             (give-up-red-microphone))))
```

Code Attempt #2:

```
(define (mark-red some-speech)
  (lambda ()
    (if (red-microphone-available?)
        (begin (grab-red-microphone)
               (some-speech)
               (give-up-red-microphone)))))
```

Code Attempt #3:

```
(define (mark-red some-speech)
  (define (wait-for-free-red-microphone)
    (if (red-microphone-available?)
        'ready
        (begin (wait-for-random-time)
               (wait-for-free-red-microphone))))
  (lambda ()
    (wait-for-free-red-microphone)
    (grab-red-microphone)
    (some-speech)
    (give-up-red-microphone)))
```

Code Attempt #4:

```
(define (mark-red SOME-PROC)
  (define (wait-for-free-red-microphone)
    (if (red-microphone-available?)
        'ready (begin (wait-for-random-time)
                      (wait-for-free-red-microphone))))
  (lambda ()
    (wait-for-free-red-microphone)
    (grab-red-microphone)
    (let ((result (SOME-PROC)))
      (give-up-red-microphone)
      result)))
```

Coordination Procedures

How can we implement the "microphone" coordination procedures?

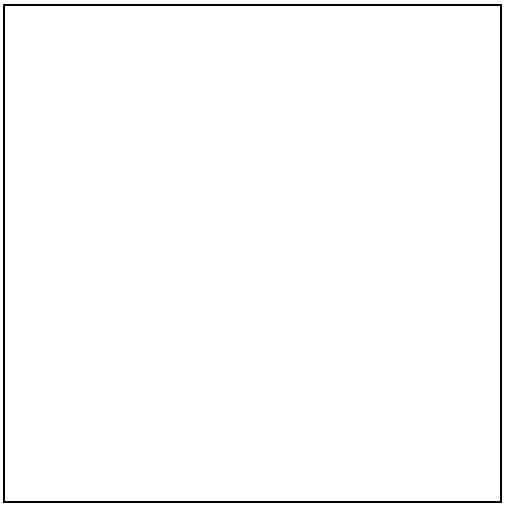
```
(define red-mike-available #t)

(define (red-microphone-available?)
  red-mike-available)

(define (grab-red-microphone)
  (set! red-mike-available #f))

(define (give-up-red-microphone)
  (set! red-mike-available #t))
```

Problem:



Approach: Seek a fine grain "atomic action" - e.g test-and-set!

Mutex

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

(define (clear! cell)
  (set-car! cell #f))

;; SHOULD BE AN ATOMIC ACTION
(define (test-and-set! cell)
  (if (car cell)
      #t
      (begin (set-car! cell #t)
             #f)))
```

Serializer

```
(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)))
```