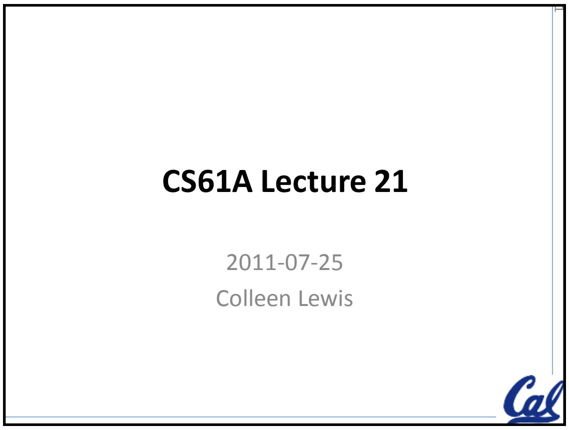
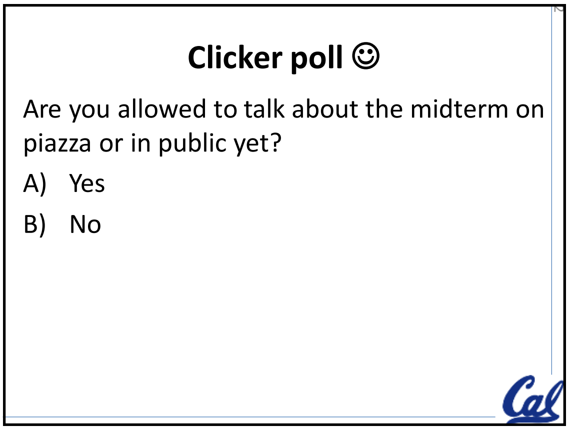


CS61A Lecture 21

2011-07-25

Colleen Lewis

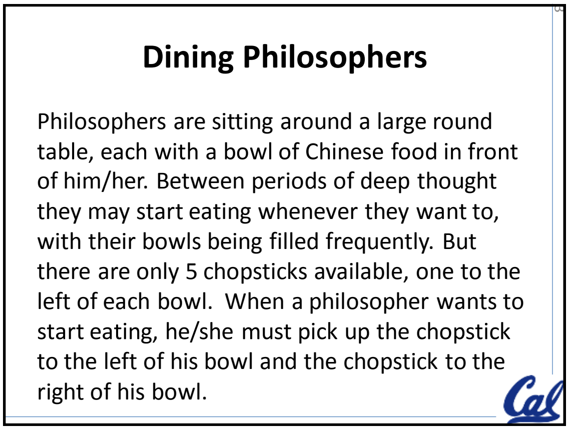


Clicker poll ☺

Are you allowed to talk about the midterm on piazza or in public yet?

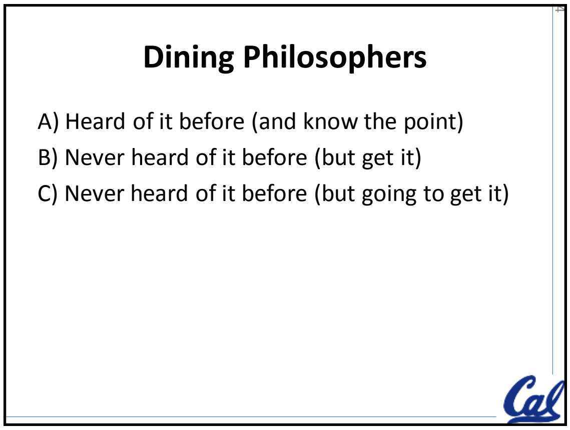
A) Yes

B) No



Dining Philosophers

Philosophers are sitting around a large round table, each with a bowl of Chinese food in front of him/her. Between periods of deep thought they may start eating whenever they want to, with their bowls being filled frequently. But there are only 5 chopsticks available, one to the left of each bowl. When a philosopher wants to start eating, he/she must pick up the chopstick to the left of his bowl and the chopstick to the right of his bowl.

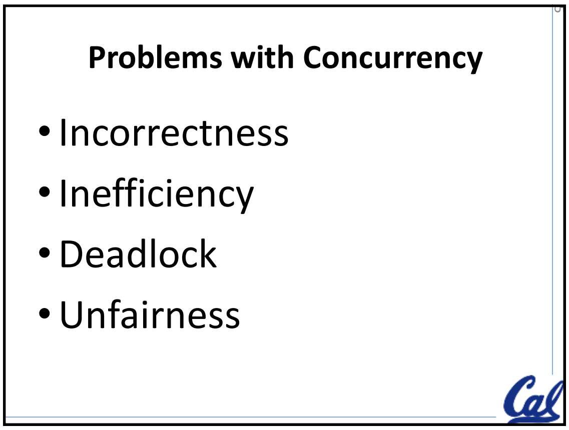


Dining Philosophers

A) Heard of it before (and know the point)

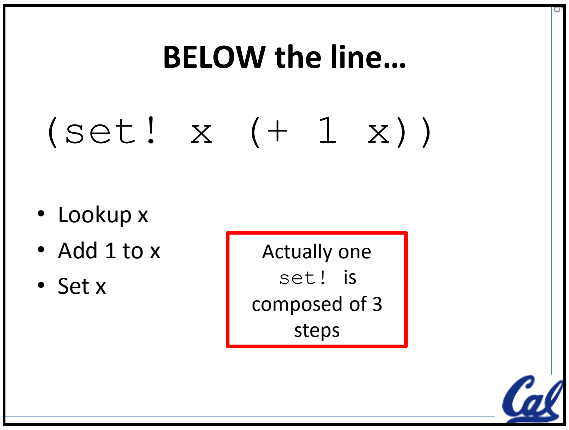
B) Never heard of it before (but get it)

C) Never heard of it before (but going to get it)



Problems with Concurrency

- Incorrectness
- Inefficiency
- Deadlock
- Unfairness

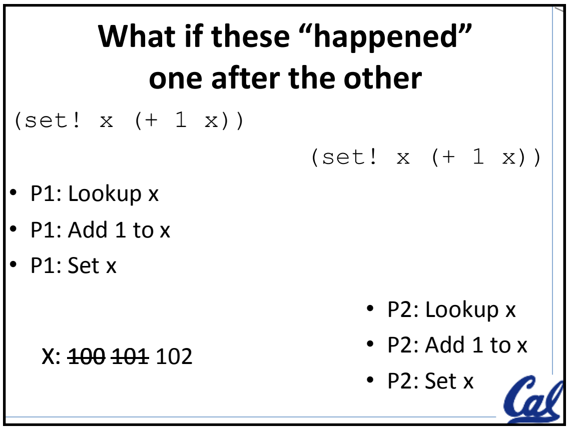


BELOW the line…

```
(set! x (+ 1 x))
```

- Lookup x
- Add 1 to x
- Set x

Actually one `set!` is composed of 3 steps



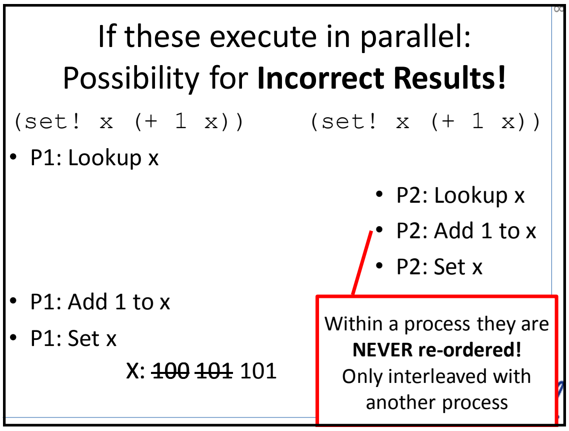
What if these "happened" one after the other

```
(set! x (+ 1 x))
                        (set! x (+ 1 x))
```

- P1: Lookup x
- P1: Add 1 to x
- P1: Set x

X: ~~100~~ ~~101~~ 102

- P2: Lookup x
- P2: Add 1 to x
- P2: Set x



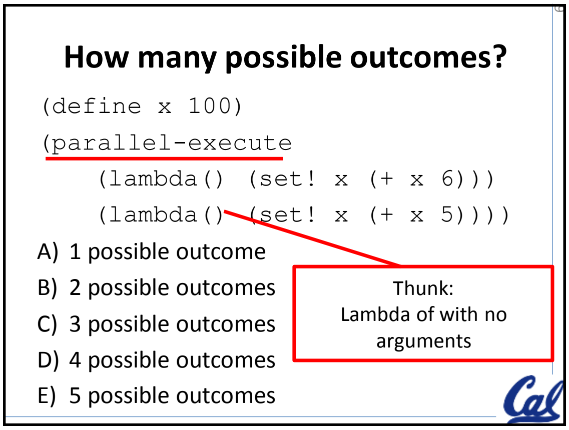
If these execute in parallel: Possibility for **Incorrect Results!**

```
(set! x (+ 1 x))        (set! x (+ 1 x))
```

- P1: Lookup x
- P2: Lookup x
- P2: Add 1 to x
- P2: Set x
- P1: Add 1 to x
- P1: Set x

X: ~~100~~ ~~101~~ 101

Within a process they are **NEVER re-ordered!** Only interleaved with another process

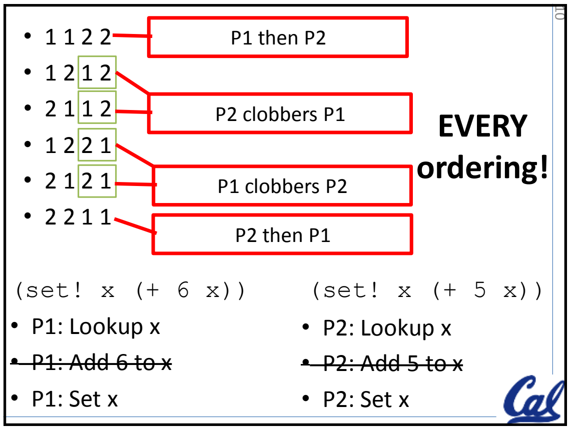


How many possible outcomes?

```
(define x 100)
(parallel-execute
     (lambda () (set! x (+ x 6)))
     (lambda () (set! x (+ x 5))))
```

Thunk: Lambda of with no arguments

A) 1 possible outcome
B) 2 possible outcomes
C) 3 possible outcomes
D) 4 possible outcomes
E) 5 possible outcomes



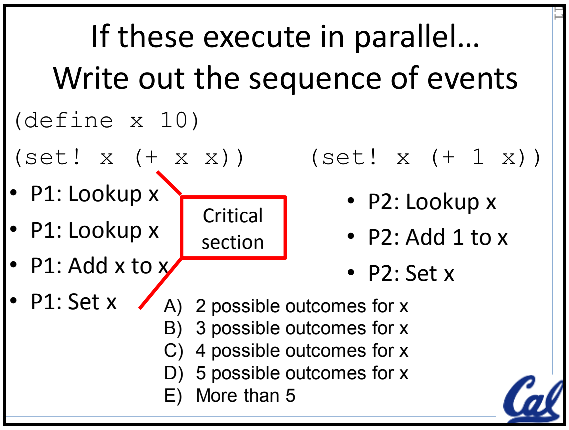
- 1 1 2 2 — P1 then P2
- 1 2 1 2
- 2 1 1 2 — P2 clobbers P1
- 1 2 2 1
- 2 1 2 1 — P1 clobbers P2
- 2 2 1 1 — P2 then P1

EVERY ordering!

```
(set! x (+ 6 x))        (set! x (+ 5 x))
```

- P1: Lookup x
- ~~P1: Add 6 to x~~
- P1: Set x

- P2: Lookup x
- ~~P2: Add 5 to x~~
- P2: Set x



If these execute in parallel… Write out the sequence of events

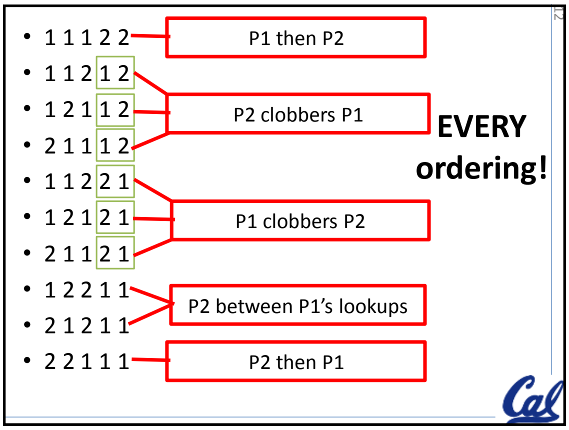
```
(define x 10)
(set! x (+ x x))        (set! x (+ 1 x))
```

- P1: Lookup x
- P1: Lookup x
- P1: Add x to x
- P1: Set x

Critical section

- P2: Lookup x
- P2: Add 1 to x
- P2: Set x

A) 2 possible outcomes for x
B) 3 possible outcomes for x
C) 4 possible outcomes for x
D) 5 possible outcomes for x
E) More than 5



- 1 1 1 2 2 — P1 then P2
- 1 1 2 1 2
- 1 2 1 1 2
- 2 1 1 1 2 — P2 clobbers P1
- 1 1 2 2 1
- 1 2 1 2 1
- 2 1 1 2 1 — P1 clobbers P2
- 1 2 2 1 1
- 2 1 2 1 1 — P2 between P1's lookups
- 2 2 1 1 1 — P2 then P1

EVERY ordering!

Possible outcomes

```
(set! x (+ x x))      (set! x (+ 1 x))
```

P1 then P2

- P1: Lookup x
- P1: Lookup x
- P1: Set x
- P2: Lookup x
- P2: Set x

P2 clobbers P1

- P1: Lookup x
- P1: Lookup x
- P2: Lookup x
- P1: Set x
- P2: Set x

P2 then P1

- P2: Lookup x
- P2: Set x
- P1: Lookup x
- P1: Lookup x
- P1: Set x

P1 clobbers P2

- P1: Lookup x
- P1: Lookup x
- P2: Lookup x
- P2: Set x
- P1: Set x

P2 between P1's lookups

- P1: Lookup x
- P2: Lookup x
- P2: Set x
- P1: Lookup x
- P1: Set x

Our definition of a "correct answer"?

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

Correct answers:

21 (line 1 first)

22 (line 2 first)

"Ensure that a concurrent system produces the same result as if the processes had run sequentially in **some** order."

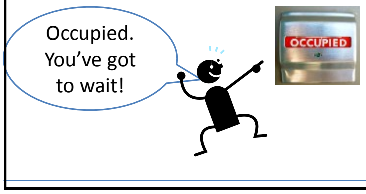
Protecting from incorrectness

Serializers protect things

And make things they protect serial

(def: taking place in a series)

```
(define stephanie (make-serializer))
(define phill (make-serializer))
(define hamilton (make-serializer))
```



Serializers protect things

And make things they protect serial

(def: taking place in a series)

```
(define stephanie-x (make-serializer))
(parallel-execute
  (stephanie-x (lambda () (set! x (+ x 1)))
  (stephanie-x (lambda () (set! x (+ x x)))
  (stephanie-x (lambda () (set! x (+ x 9))))
```

Serializer `stephanie-x` will make sure nothing she protects happen concurrently.

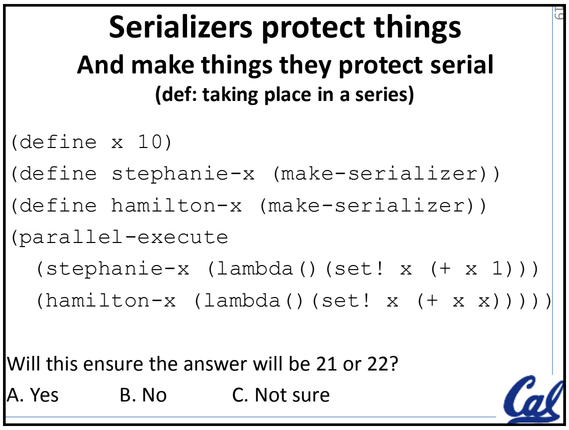
"Ensure that a concurrent system produces the same result as if the processes had run sequentially in **some** order."

Serializers protect things

And make things they protect serial

(def: taking place in a series)

```
(define hamilton-x (make-serializer))
(define phill-y (make-serializer))
(parallel-execute
  (hamilton-x (lambda () (set! x (+ x 1)))
  (hamilton-x (lambda () (set! x (+ x x)))
  (phill-y    (lambda () (set! y (+ y 1)))
  (phill-y    (lambda () (set! y (+ y y)))
  (hamilton-x (lambda () (set! x (+ x 9))))
```



Serializers protect things

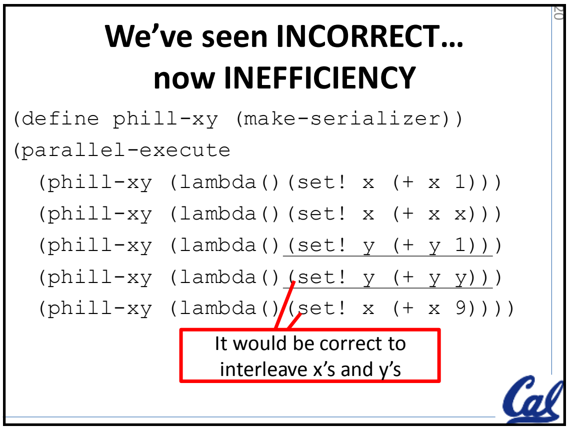
And make things they protect serial

(def: taking place in a series)

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

Will this ensure the answer will be 21 or 22?

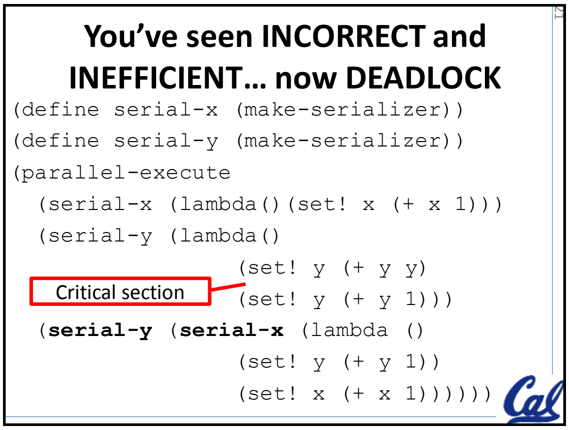
A. Yes B. No C. Not sure



We've seen INCORRECT… now INEFFICIENCY

```
(define phill-xy (make-serializer))
(parallel-execute
  (phill-xy (lambda()(set! x (+ x 1)))
  (phill-xy (lambda()(set! x (+ x x)))
  (phill-xy (lambda()(set! y (+ y 1)))
  (phill-xy (lambda()(set! y (+ y y)))
  (phill-xy (lambda()(set! x (+ x 9))))
```

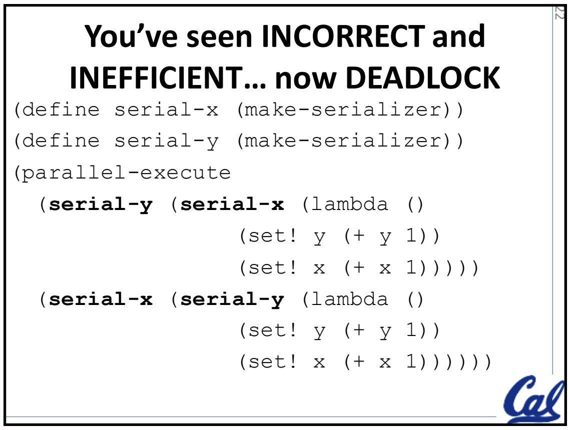
It would be correct to interleave x's and y's



You've seen INCORRECT and INEFFICIENT… now DEADLOCK

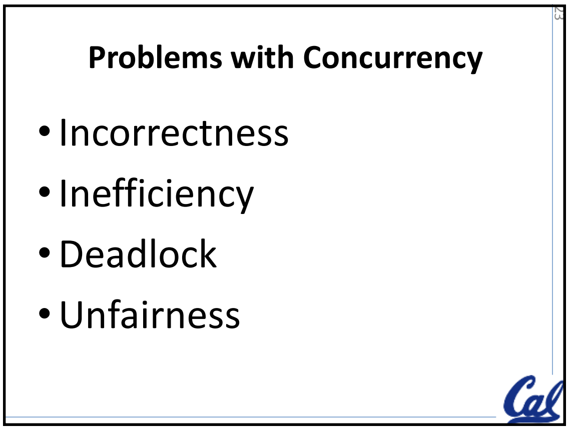
```
(define serial-x (make-serializer))
(define serial-y (make-serializer))
(parallel-execute
  (serial-x (lambda()(set! x (+ x 1)))
  (serial-y (lambda()
                      (set! y (+ y y)
                      (set! y (+ y 1)))
  (serial-y (serial-x (lambda ()
                      (set! y (+ y 1))
                      (set! x (+ x 1))))))
```

Critical section



You've seen INCORRECT and INEFFICIENT… now DEADLOCK

```
(define serial-x (make-serializer))
(define serial-y (make-serializer))
(parallel-execute
  (serial-y (serial-x (lambda ()
                      (set! y (+ y 1))
                      (set! x (+ x 1)))))
  (serial-x (serial-y (lambda ()
                      (set! y (+ y 1))
                      (set! x (+ x 1))))))
```



Problems with Concurrency

- Incorrectness
- Inefficiency
- Deadlock
- Unfairness

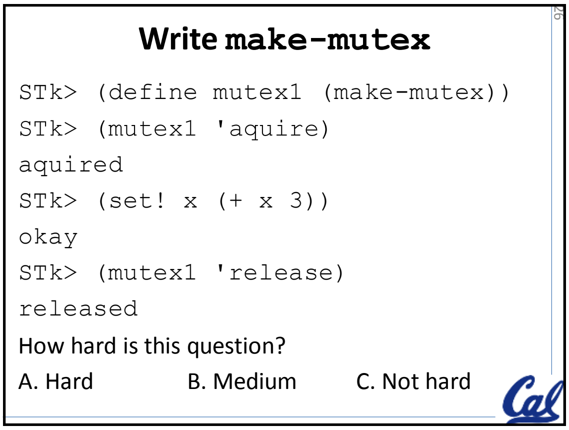
Implementing Serializers

With a mutex

Wrinkles and I want to share

Is-someone-using-the-slide?

NO

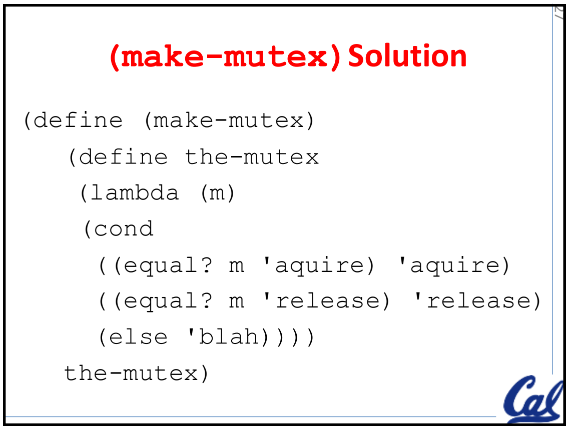


Write make-mutex

```
STk> (define mutex1 (make-mutex))
STk> (mutex1 'aquire)
aquired
STk> (set! x (+ x 3))
okay
STk> (mutex1 'release)
released
```

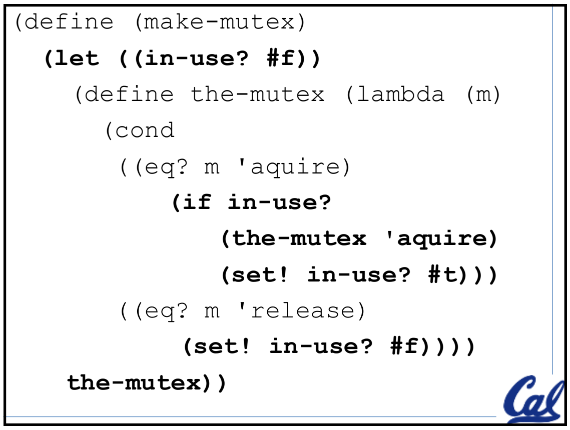
How hard is this question?

A. Hard B. Medium C. Not hard

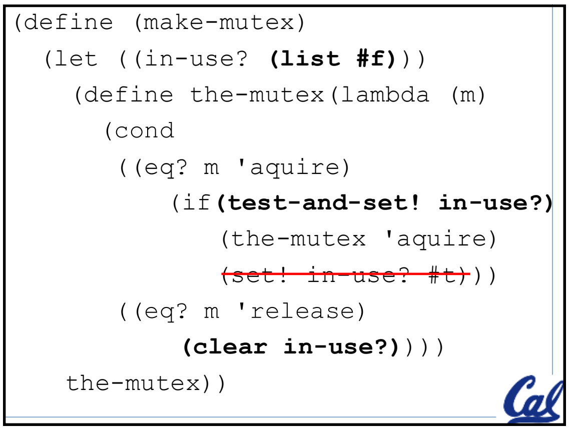


(make-mutex) Solution

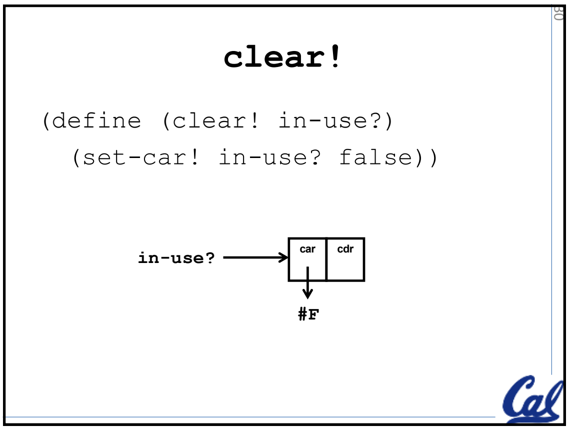
```
(define (make-mutex)
   (define the-mutex
    (lambda (m)
    (cond
     ((equal? m 'aquire) 'aquire)
     ((equal? m 'release) 'release)
     (else 'blah))))
   the-mutex)
```



```
(define (make-mutex)
  (let ((in-use? #f))
    (define the-mutex (lambda (m)
      (cond
       ((eq? m 'aquire)
          (if in-use?
              (the-mutex 'aquire)
              (set! in-use? #t)))
       ((eq? m 'release)
           (set! in-use? #f))))
   the-mutex))
```



```
(define (make-mutex)
  (let ((in-use? (list #f)))
    (define the-mutex(lambda (m)
      (cond
       ((eq? m 'aquire)
          (if(test-and-set! in-use?)
              (the-mutex 'aquire)
              ~~(set! in-use? #t)~~))
       ((eq? m 'release)
           (clear in-use?))))
   the-mutex))
```



clear!

```
(define (clear! in-use?)
  (set-car! in-use? false))
```

```
STk> (define in-use? (list #f))
in-use?
STk> (test-and-set! in-use?)
#f
STk> in-use?
(#t)
STk> (define in-use? (list #t))
in-use?
STk> (test-and-set! in-use?)
#t
STk> in-use?
(#t)
```

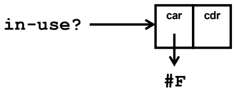
A) Student chalk
B) Student emacs
C) Colleen chalk
D) Colleen emacs

Write `test-and-set!`

`test-and-set!` SOLUTION

```
(define (test-and-set! in-use?)
  (if (car in-use?)
      #t
      (begin
           (set-car! in-use? true)
           #f)))
```

Normally built into hardware! And this can have concurrency problems too!



```
(define (make-mutex)
  (let ((in-use? (list #f)))
    (define the-mutex (lambda (m)
      (cond
       ((eq? m 'aquire)
             (if(test-and-set! in-use?)
                (the-mutex 'aquire)
                ~~(set! in-use? #t)))~~
       ((eq? m 'release)
             (clear in-use?)))))
the-mutex))
```

Does our `test-and-set!` Code work with this `make-mutex` code? A) Yes B) No

“Correct answers”

```
(define y 4)
(parallel-execute
  (lambda ()
    (set! y (* y y)))
  (lambda ()
    (set! y (+ y 2)))
  (lambda ()
    (set! y (/ y 2))))
```

How many correct answers?

A) 1
B) 3
C) 4
D) 5
E) 6