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:

\[(\text{set! } x \ x \ (+ \ x \ 1))\]
\[(\text{set! } x \ x \ (+ \ x \ 1))\]

- P1: Lookup \(x\)
- P1: Add 1 to \(x\)
- P1: Set \(x\)

\(x: 100\ 101\ 102\)

If these execute in parallel:

\[(\text{set! } x \ x \ (+ \ x \ 1))\] \[(\text{set! } x \ x \ (+ \ x \ 1))\]

- P1: Lookup \(x\)
- P1: Add 1 to \(x\)
- P1: Set \(x\)
- P2: Lookup \(x\)
- P2: Add 1 to \(x\)
- P2: Set \(x\)

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

\(x: 100\ 101\ 101\)

How many possible outcomes?

\[(\text{define } x \ 100)\]
\[(\text{parallel-execute})\]
\[(\text{lambda}() \ (\text{set! } x \ x \ (+ \ x \ 6)))\]
\[(\text{lambda}() \ (\text{set! } x \ x \ (+ \ x \ 5)))\]

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

Thunk: Lambda of with no arguments

If these execute in parallel...

Write out the sequence of events:

\[(\text{define } x \ 10)\]
\[(\text{set! } x \ (+ \ x \ x))\] \[(\text{set! } x \ (+ \ x \ 1))\]

- P1: Lookup \(x\)
- P1: Lookup \(x\)
- P1: Add \(x\) to \(x\)
- P1: 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

E) Critical section

If these execute in parallel...

Write out the sequence of events:

\[(\text{define } x \ 10)\]
\[(\text{set! } x \ (+ \ x \ x))\] \[(\text{set! } x \ (+ \ x \ 1))\]

- P1: Lookup \(x\)
- P1: Add \(x\) to \(x\)
- P1: Set \(x\)
- 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

E) Critical section
Possible outcomes

\[(\text{set! } x \ (+ \ x \ x))\] \[(\text{set! } x \ (+ \ l \ x))\]

- P1 then P2
  - P1: Lookup x
  - P1: Lookup x
  - P1: Set x
  - P2: Lookup x
  - P2: Set x
- P2 then P1
  - P2: Lookup x
  - P2: Set x
  - P1: Lookup x
  - P1: Set x
- P2 between P1's lookups
  - P1: Lookup x
  - P2: Lookup x
  - P1: Lookup x
  - P2: Set x
  - P1: Lookup x
  - P1: Set x
- P2 clobbers P1
  - P1: Lookup x
  - P2: Lookup x
  - P1: Set x
- P1 clobbers P2
  - P1: Lookup x
  - P2: Lookup x
  - P2: Set x
  - P1: Set x

Our definition of a “correct answer”?

\[
\begin{align*}
\text{(define } x \ 10) \\
\text{(parallel-execute)} \\
&\quad (\text{lambda}() \ (\text{set! } x \ (+ \ x \ x))) \\
&\quad (\text{lambda}() \ (\text{set! } x \ (+ \ l \ x)))
\end{align*}
\]

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

\[
\begin{align*}
\text{(define stephanie \ (make-serializer))} \\
\text{(define phill \ (make-serializer))} \\
\text{(define hamilton \ (make-serializer))}
\end{align*}
\]

Serializer \text{stephanie-x} will make sure nothing she protects happen concurrently.
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! x (+ x 1))
      (set! x (+ x l)))))))

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 l)))))))

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))
Write test-and-set!

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

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

```
(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

```
(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)))
```

“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