Deferred operations

(define the-cons (cons 1 #f))
(set-cdr! the-cons the-cons)

(define (run-in-circles l)
  (+ (run-in-circles (cdr l))))
Deferred operations

(define the-cons (cons 1 #f))
(set-cdr! the-cons the-cons)

(define (run-in-circles l)
  (+ (run-in-circles (cdr l)))))

."
Deferred operations

(define the-cons (cons 1 #f))
(set-cdr! the-cons the-cons)

(define (run-in-circles l)
  (+ (run-in-circles (cdr l)))))

..
Deferred operations

(define the-cons (cons 1 #f))
(set-cdr! the-cons the-cons)

(define (run-in-circles l)
  (+ (run-in-circles (cdr l)))))

..“The program ran out of memory”
(define the-cons (cons 1 #f))
(set-cdr! the-cons the-cons)

(define (run-in-circles l)
  (run-in-circles (cdr l)))
Tail recursion in action

(define the-cons (cons 1 #f))
(set-cdr! the-cons the-cons)

(define (run-in-circles l)
  (run-in-circles (cdr l)))
(define the-cons (cons 1 #f))
(set-cdr! the-cons the-cons)

(define (run-in-circles l)
  (run-in-circles (cdr l)))

..
(define the-cons (cons 1 #f))
(set-cdr! the-cons the-cons)

(define (run-in-circles l)
    (run-in-circles (cdr l)))
Tail recursion in action

(define the-cons (cons 1 #f))
(set-cdr! the-cons the-cons)

(define (run-in-circles l)
  (run-in-circles (cdr l)))

......
Tail recursion in action

(define the-cons (cons 1 #f))
(set-cdr! the-cons the-cons)

(define (run-in-circles l)
  (run-in-circles (cdr l)))
(define the-cons (cons 1 #f))
(set-cdr! the-cons the-cons)

(define (run-in-circles l)
    (run-in-circles (cdr l)))
Tail recursion in action

(define the-cons (cons 1 #f))
(set-cdr! the-cons the-cons)

(define (run-in-circles l)
  (run-in-circles (cdr l)))

...............
What if we never had any deferred operations?
What if we never had any deferred operations?

Instead of *returning a value* with deferred operations, the function is passed a *continuation procedure*, which we call to return a value.
What if we never had any deferred operations?

Instead of *returning a value* with deferred operations, the function is passed a *continuation procedure*, which we call to return a value.

Which means that all function calls are *tail-recursive*.
(define (add-17 x)
  (+ x 17))
(define (add-17 x)
    (+ x 17))

(define (add-17 x cont)
    (cont (+ x 17)))
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1))))))
(define (factorial n)
  (if (= n 0)
    1
    (* n (factorial (- n 1)))))

(define (factorial n cont)
  (if (= n 0)
    (cont 1)
    (factorial (- n 1)
      (lambda (x) (cont (* n x)))))))
(define (factorial n)
  (if (= n 0)
    1
    (* n (factorial (- n 1)))))

(define (factorial n cont)
  (if (= n 0)
    (cont 1)
    (factorial (- n 1)
      (lambda (x) (cont (* n x)))))

(factorial 10 (lambda (x) x))
(define (factorial n cont)
  (if (= n 0)
      (cont 1)
      (factorial (- n 1)
                  (lambda (x) (cont (* n x)))))))

(factorial 10 (lambda (x) x))

- No deferred operations
(define (factorial n cont)
  (if (= n 0)
      (cont 1)
      (factorial (- n 1)
                  (lambda (x) (cont (* n x))))))

(factorial 10 (lambda (x) x))

- No deferred operations
- We craft a **new** continuation, based on the previous one, and pass that to our recursive call
(define (factorial n cont)
  (if (= n 0)
      (cont 1)
      (factorial (- n 1)
                   (lambda (x) (cont (* n x))))))

(factorial 10 (lambda (x) x))

- No deferred operations
- We craft a new continuation, based on the previous one, and pass that to our recursive call
- Asks the question, “What will I do with the return value of the recursive call?”
Factorial in CPS

(define (factorial n cont)
  (if (= n 0)
      (cont 1)
      (factorial (- n 1)
                  (lambda (x) (cont (* n x))))))

(factorial 10 (lambda (x) x))

- No deferred operations
- We craft a new continuation, based on the previous one, and pass that to our recursive call
- Asks the question, "What will I do with the return value of the recursive call?"
- "Multiply it by n, and call my continuation with that value"
(define (sum-interval a b cont)
  (if (= a b)
      (cont a)
      (sum-interval
       (+ a 1)
       b
       (lambda (x) (cont (+ a x))))))
(define (append l1 l2)
  (if (eq? l1 '())
      l2
      (cons (car l1) (append (cdr l1) l2)))))
(define (append l1 l2)
  (if (eq? l1 '())
      l2
      (cons (car l1) (append (cdr l1) l2)))))

(define (cs-append l1 l2 cont)
  (if (eq? l1 '())
      (cont l2)
      (cs-append
       (cdr l1)
       l2
       (lambda (appended-cdr)
        (cons (car l1) appended-cdr))))))
(define (append l1 l2)
  (if (eq? l1 '())
      l2
      (cons (car l1) (append (cdr l1) l2))))

(define (cs-append l1 l2 cont)
  (if (eq? l1 '())
      (cont l2)
      (cs-append
        (cdr l1)
        l2
        (lambda (appended-cdr)
          (cont (cons (car l1) appended-cdr))))))
(define (flatten tree)
  (cond ((null? tree) '())
        ((not (pair? tree)) (list tree))
        (else (append (flatten (car tree))
                      (flatten (cdr tree))))))

(define (cs-flatten tree cont)
  (cond ((null? tree) (cont '()))
        ((not (pair? tree)) (cont (list tree))
        (else (cs-flatten (car tree)
                         (lambda (car-leaves)
                           (cs-flatten (cdr tree)
                                        (lambda (cdr-leaves)
                                          (cont (append car-leaves cdr-leaves))))))))))
(define (flatten tree)
  (cond ((null? tree) '())
    ((not (pair? tree)) (list tree))
    (else (append (flatten (car tree))
                  (flatten (cdr tree))))))

(define (cs-flatten tree cont)
  (cond ((null? tree) (cont '()))
    ((not (pair? tree)) (cont (list tree)))
    (else (cs-flatten
            (car tree)
            (lambda (car-leaves)
                (cs-flatten
                 (cdr tree)
                 (lambda (cdr-leaves)
                     (cont
                      (append car-leaves cdr-leaves)
                      ))))))))
(define (flatten tree)
  (cond ((null? tree) '())
    ((not (pair? tree)) (list tree))
    (else (append (flatten (car tree))
                  (flatten (cdr tree))))))

(define (cs-flatten tree cont)
  (cond ((null? tree) (cont '()))
    ((not (pair? tree)) (cont (list tree)))
    (else (cs-flatten
            (car tree)
            (lambda (car-leaves)
              (cs-flatten
               (cdr tree)
               (lambda (cdr-leaves)
                (cont
                 (append car-leaves cdr-leaves)
                 ))))))))
(define (flatten tree)
  (cond ((null? tree) '())
        ((not (pair? tree)) (list tree))
        (else (append (flatten (car tree))
                          (flatten (cdr tree))))))

(define (cs-flatten tree cont)
  (cond ((null? tree) (cont '()))
       ((not (pair? tree)) (cont (list tree)))
       (else (cs-flatten
               (car tree)
               (lambda (car-leaves)
                (cs-flatten
                 (cdr tree)
                 (lambda (cdr-leaves)
                  (cont
                   (append car-leaves cdr-leaves)
                   ))))))))
(define (flatten tree)
   (cond ((null? tree) '())
         ((not (pair? tree)) (list tree))
         (else (append (flatten (car tree))
                         (flatten (cdr tree))))))

(define (cs-flatten tree cont)
   (cond ((null? tree) (cont '()))
         ((not (pair? tree)) (cont (list tree)))
         (else (cs-flatten
                 (car tree)
                 (lambda (car-leaves)
                     (cs-flatten
                      (cdr tree)
                      (lambda (cdr-leaves)
                          (cont
                           (append car-leaves cdr-leaves)
                          ))))))))
(define (flatten tree)
  (cond ((null? tree) '())
    ((not (pair? tree)) (list tree))
    (else (append (flatten (car tree))
      (flatten (cdr tree))))))

(define (cs-flatten tree cont)
  (cond ((null? tree) (cont '()))
    ((not (pair? tree)) (cont (list tree)))
    (else (cs-flatten
      (car tree)
      (lambda (car-leaves)
        (cs-flatten
          (cdr tree)
          (lambda (cdr-leaves)
            (cont
              (append car-leaves cdr-leaves)
            ))))))))
(define (flatten tree)
  (cond ((null? tree) '())
     ((not (pair? tree)) (list tree))
     (else (append (flatten (car tree))
                   (flatten (cdr tree))))))

(define (cs-flatten tree cont)
  (cond ((null? tree) (cont '()))
     ((not (pair? tree)) (cont (list tree)))
     (else (cs-flatten
            (car tree)
            (lambda (car-leaves)
              (cs-flatten
               (cdr tree)
               (lambda (cdr-leaves)
                 (cont
                  (append car-leaves cdr-leaves)
                  ))))))))
Continuation-passing style is also very useful in controlling program flow
Control flow

- Continuation-passing style is also very useful in controlling program flow
- Error handling and exceptions is a classic case:

\[
\text{define (divide a b success fail)}\\
\text{(if (= b 0)}\\
\text{  (fail "divide-by-zero") }\\
\text{  (success (/ a b)))}
\]
Control flow

- Continuation-passing style is also very useful in controlling program flow
- Error handling and exceptions is a classic case:

```scheme
(define (divide a b success fail)
  (if (= b 0)
    (fail "divide-by-zero")
    (success (/ a b)))
)```
We can write a Scheme interpreter in continuation-passing style
Continuations in the interpreter

We can write a Scheme interpreter in continuation-passing style

```scheme
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (if (eq? input '**quit**)
       'c-eval-done
       (c-eval
        input
        the-global-environment
        (lambda (output)
           (announce-output output-prompt output)
           (display output)
           (driver-loop))))))
```

Alex Vandiver (MIT)  Continuations and Concurrency  Lecture 7
(define (c-eval exp env cont)
  (cond ((self-evaluating? exp) (cont exp))
    ((variable? exp) (cont (lookup-variable-value exp env)))
    ((quoted? exp) (cont (text-of-quotatio
(define (c-eval exp env cont)
  (cond ((self-evaluating? exp)
    (cont exp))
    ((variable? exp)
    (cont (lookup-variable-value exp env)))
    ((quoted? exp)
    (cont (text-of-quotiation exp)))
    ((assignment? exp)
    (eval-assignment exp env cont))
    ((definition? exp)
    (eval-definition exp env cont))
    ((if? exp) (eval-if exp env cont))
    ((lambda? exp)
    (cont (make-procedure (lambda-parameters exp)
                             (lambda-body exp) env)))
    ((begin? exp)
    (eval-sequence (begin-actions exp) env cont))
    ((cond? exp)
    (c-eval (cond->if exp) env cont))
...
(define (c-eval exp env cont)
  (cond ((self-evaluating? exp)
         (cont exp))
        ((variable? exp)
         (cont (lookup-variable-value exp env)))
        ((quoted? exp)
         (cont (text-of-quotation exp)))
        ((assignment? exp)
         (eval-assignment exp env cont))
        ((definition? exp)
         (eval-definition exp env cont))
        ((if? exp) (eval-if exp env cont))
        ((lambda? exp)
         (cont (make-procedure (lambda-parameters exp)
                                   (lambda-body exp) env)))
        ((begin? exp)
         (eval-sequence (begin-actions exp) env cont))
        ((cond? exp)
         (c-eval (cond->if exp) env cont))
...
(define (c-eval exp env cont)
  (cond ((self-evaluating? exp)
             (cont exp))
        ((variable? exp)
             (cont (lookup-variable-value exp env)))
        ((quoted? exp)
             (cont (text-of-quotational exp)))
        ((assignment? exp)
             (eval-assignment exp env cont))
        ((definition? exp)
             (eval-definition exp env cont))
        ((if? exp) (eval-if exp env cont))
        ((lambda? exp)
             (cont (make-procedure (lambda-parameters exp)
                                     (lambda-body exp) env)))
        ((begin? exp)
             (eval-sequence (begin-actions exp) env cont))
        ((cond? exp)
             (c-eval (cond->if exp) env cont))
        ...
(define (eval-if exp env cont)
  (c-eval
    (if-predicate exp) env
    (lambda (test-value)
      (if test-value
        (c-eval (if-consequent exp) env cont)
        (c-eval (if-alternative exp) env cont))))))
(define (eval-if exp env cont)
  (c-eval
   (if-predicate exp) env
   (lambda (test-value)
     (if test-value
       (c-eval (if-consequent exp) env cont)
       (c-eval (if-alternative exp) env cont))))
)

(define (eval-sequence exps env cont)
  (if (last-exp? exps)
      (c-eval (first-exp exps) env cont)
      (c-eval (first-exp exps) env
        (lambda (ignored)
          (eval-sequence
           (eval-sequence
            (rest-exps exps)
            env cont))))))
Continuations with the interpreter

What if the evaluator made its continuations available to the language?

(define (eval-call-with-current-continuation exp env cont)
  (c-eval
   (call/cc-proc exp) env
   (lambda (proc-to-call)
     (c-apply proc-to-call
               (list (make-continuation cont)) cont))))

;; in c-apply
((continuation? procedure)
  (apply (continuation-internal-cont procedure) arguments))
What if the evaluator made its continuations available to the language?

call-with-current-continuation procedure
What if the evaluator made its continuations available to the language? 

`call-with-current-continuation` procedure

```scheme
;; Special form for evaluator
(define (eval-call-with-current-continuation exp env cont)
  (c-eval
   (call/cc-proc exp) env
   (lambda (proc-to-call)
     (c-apply proc-to-call
       (list (make-continuation cont))
       cont)))))
```

```scheme
;; in c-apply
((continuation? procedure)
 (apply (continuation-internal-cont procedure)
        arguments))
```
call/cc example

(+ (* 3 (call-with-current-continuation
    (lambda (cont)
      (cont 5))))
  10)

(c 6)

(+ 100 (c 6))
call/cc example

(+ (* 3 (call-with-current-continuation
    (lambda (cont)
      (cont 5)))))

10)
; => 25
call/cc example

(+ (* 3 (call-with-current-continuation
    (lambda (cont)
        (cont 5))))
    10)
; => 25
(define c #f)
(+ (* 3 (call-with-current-continuation
    (lambda (cont)
        (set! c cont)
        (cont 5))))
    10)
call/cc example

(+ (* 3 (call-with-current-continuation
    (lambda (cont)
      (cont 5))))
  10)
; => 25

(define c #f)
(+ (* 3 (call-with-current-continuation
    (lambda (cont)
      (set! c cont)
      (cont 5))))
  10)
; => 25
call/cc example

(+ (* 3 (call-with-current-continuation
    (lambda (cont)
      (cont 5))))
   10)
; => 25
(define c #f)
(+ (* 3 (call-with-current-continuation
    (lambda (cont)
      (set! c cont)
      (cont 5))))
   10)
; => 25
(c 6)
(define c #f)
(+ (* 3 (call-with-current-continuation
    (lambda (cont)
      (cont 5))))
    10)
; => 25
(c 6)
; => 28
(+ 100 (c 6))
; => 28
(+ (* 3 (call-with-current-continuation
    (lambda (cont)
      (cont 5)))))
  10)
; => 25
(define c #f)
(+ (* 3 (call-with-current-continuation
    (lambda (cont)
      (set! c cont)
      (cont 5)))))
  10)
; => 25
(c 6)
; => 28
(+ 100 (c 6))
call/cc example

(+ (* 3 (call-with-current-continuation
    (lambda (cont)
      (cont 5)))))
  10)
; => 25
(define c #f)
(+ (* 3 (call-with-current-continuation
    (lambda (cont)
      (set! c cont)
      (cont 5)))))
  10)
; => 25
(c 6)
; => 28
(+ 100 (c 6))
; => 28
call-with-current-continuation (or call/cc, as it is usefully shortened to) takes a procedure as an argument, and passes it the evaluator's current continuation. The return value of call/cc is the same as the return value of the procedure. . . or the procedure could just call the continuation it was given. Which is exactly identical in meaning! The continuation of the call/cc expression, the continuation of the procedure that it calls, and the value that it passes as an argument to that procedure, are all the same!
call/cc explained

- call-with-current-continuation (or call/cc, as it is usefully shortened to) takes a procedure as an argument, and passes it the evaluator’s current continuation
- The return value of call/cc is the same as the return value of the procedure
call-with-current-continuation (or call/cc, as it is usefully shortened to) takes a procedure as an argument, and passes it the evaluator’s current continuation.

The return value of call/cc is the same as the return value of the procedure.

... or the procedure could just call the continuation it was given. *Which is exactly identical in meaning!*
call/cc explained

- call-with-current-continuation (or call/cc, as it is usefully shortened to) takes a procedure as an argument, and passes it the evaluator’s current continuation.
- The return value of call/cc is the same as the return value of the procedure.
- ... or the procedure could just call the continuation it was given. Which is exactly identical in meaning!
- The continuation of the call/cc expression, the continuation of the procedure that it calls, and the value that it passes as an argument to that procedure, are all the same!
Storing continuations

- Stored continuations can be saved away to “jump back” at any later point in time

(define cont #f)
(if (call/cc (lambda (c)
(set! cont c)
#t))
'something
'other-thing)
; => 'something
(cont #f)
; => 'other-thing
Stored continuations can be saved away to “jump back” at any later point in time

(define cont #f)
(if (call/cc (lambda (c)
               (set! cont c)
               #t))
   'something
   'other-thing)
Storing continuations

- Stored continuations can be saved away to “jump back” at any later point in time

```scheme
(define cont #f)
(if (call/cc (lambda (c)
            (set! cont c)
            #t))
    'something
    'other-thing)
; => 'something
```
Storing continuations

- Stored continuations can be saved away to “jump back” at any later point in time

```
(define cont #f)
(if (call/cc (lambda (c)
               (set! cont c)
               #t))
   'something
   'other-thing)
; => 'something
(cont #f)
```
Storing continuations

- Stored continuations can be saved away to “jump back” at any later point in time

```
(define cont #f)
(if (call/cc (lambda (c)
               (set! cont c)
               #t))
   'something
   'other-thing)
; => 'something
(cont #f)
; => 'other-thing
```
(define (fib-func)
  (let ((prev 0)
        (cur 1))
    (define (loop)
      (define next (+ prev cur))
      (set! prev cur)
      (set! cur next)
      prev)
    loop))
(define (fib-func)
  (let ((prev 0)
         (cur 1))
    (define (loop)
      (define next (+ prev cur))
      (set! prev cur)
      (set! cur next)
      prev)
    loop))
(define test (fib-func))
(define (fib-func)
  (let ((prev 0)
         (cur 1))
    (define (loop)
      (define next (+ prev cur))
      (set! prev cur)
      (set! cur next)
      prev)
    loop))
(define test (fib-func))
(test) ; => 1
(define (fib-func)
  (let ((prev 0)
         (cur 1))
    (define (loop)
      (define next (+ prev cur))
      (set! prev cur)
      (set! cur next)
      prev
    loop))
  (define test (fib-func))
  (test) ; => 1
  (test) ; => 1
(define (fib-func)
  (let ((prev 0)
        (cur 1))
    (define (loop)
      (define next (+ prev cur))
      (set! prev cur)
      (set! cur next)
      prev)
    loop))

(define test (fib-func))
(test) ; => 1
(test) ; => 1
(test) ; => 2
(define (fib-func)
  (let ((prev 0)
         (cur 1))
    (define (loop)
      (define next (+ prev cur))
      (set! prev cur)
      (set! cur next)
      prev)
    loop))

(define test (fib-func))
(test) ; => 1
(test) ; => 1
(test) ; => 2
(test) ; => 3
(define (fib-func)
  (let ((prev 0)
         (cur 1))
    (define (loop)
      (define next (+ prev cur))
      (set! prev cur)
      (set! cur next)
      prev)
    loop))
(define test (fib-func))
(test) ; => 1
(test) ; => 1
(test) ; => 2
(test) ; => 3
(test) ; => 5
(define resume #f)
(define (fib-cont)
  (let ((prev 0)
         (cur 1))
    (define (loop)
      (define next (+ prev cur))
      (set! prev cur)
      (set! cur next)
      (if (call/cc
           (lambda (c)
             (lambda (c)
               (set! resume (lambda () (c #f)))
               (c #t)))
        prev
        (loop))
    (loop)))
(define resume #f)
(define (fib-cont)
  (let ((prev 0)
         (cur 1))
    (define (loop)
      (define next (+ prev cur))
      (set! prev cur)
      (set! cur next)
      (if (call/cc
           (lambda (c)
             (set! resume (lambda () (c #f)))))
          prev
          (loop)))
    (loop)))
(fib-cont) ; => 1
(define resume #f)
(define (fib-cont)
  (let ((prev 0)
         (cur 1))
    (define (loop)
      (define next (+ prev cur))
      (set! prev cur)
      (set! cur next)
      (if (call/cc
           (lambda (c)
             (set! resume (lambda () (c #f)))
             (c #t)))
        prev
        (loop)))
    (loop))
(fib-cont) ; => 1
(resume)   ; => 1
(define resume #f)
(define (fib-cont)
  (let ((prev 0)
         (cur 1))
    (define (loop)
      (define next (+ prev cur))
      (set! prev cur)
      (set! cur next)
      (if (call/cc
           (lambda (c)
             (set! resume (lambda () (c #f)))
             (c #t)))
        prev
        (loop)))
    (loop)))
(fib-cont) ; => 1
(resume) ; => 1
(resume) ; => 2
(define resume #f)
(define (fib-cont)
  (let ((prev 0)
         (cur 1))
    (define (loop)
      (define next (+ prev cur))
      (set! prev cur)
      (set! cur next)
      (if (call/cc
           (lambda (c)
             (set! resume (lambda () (c #f)))
             (c #t)))
          prev
          (loop)))
    (loop)))
(fib-cont) ; => 1
(resume) ; => 1
(resume) ; => 2
(resume) ; => 3
(define resume #f)
(define (fib-cont)
  (let ((prev 0)
         (cur 1))
    (define (loop)
      (define next (+ prev cur))
      (set! prev cur)
      (set! cur next)
      (if (call/cc
           (lambda (c)
             (set! resume (lambda () (c #f)))))
          prev
          (loop)))
    (loop)))
(fib-cont) ; => 1
(resume) ; => 1
(resume) ; => 2
(resume) ; => 3
(resume) ; => 5
Coroutines

- Save the continuation, return **true** now
- Save the continuation, return `true` now
- But call the continuation with `false` again, sometime in the future, to take the other branch
• Save the continuation, return **true** now
• But call the continuation with **false** again, sometime in the future, to take the other branch
• In this case, resumes the loop!
Coroutines

- Save the continuation, return `true` now
- But call the continuation with `false` again, sometime in the future, to take the other branch
- In this case, resumes the loop!
- This pattern is known as a coroutine
Coroutines

- Save the continuation, return `true` now
- But call the continuation with `false` again, sometime in the future, to take the other branch
- In this case, resumes the loop!
- This pattern is known as a coroutine
- Poor man’s threading (running multiple things at once)
Coroutines

- Save the continuation, return **true** now
- But call the continuation with **false** again, sometime in the future, to take the other branch
- In this case, resumes the loop!
- This pattern is known as a **coroutine**
- Poor man’s threading (running multiple things at once)
- ...but we can do better...
Co-operative multithreading

- Only one bit of code can run at once, but we have multiple tasks to do
Co-operative multithreading

- Only one bit of code can run at once, but we have multiple tasks to do
- Make each task declare when it's done doing some computation, and then swap
Co-operative multithreading

- Only one bit of code can run at once, but we have multiple tasks to do
- Make each task declare when it’s done doing some computation, and then swap
- “Co-operative” because tasks need to declare when they want to let someone else have a turn
Co-operative multithreading

- Only one bit of code can run at once, but we have multiple tasks to do.
- Make each task declare when it’s done doing some computation, and then swap.
- “Co-operative” because tasks need to declare when they want to let someone else have a turn.
- Used by Mac OS 9, Windows 3.1.
Co-operative multithreading plan

- Keep a list of threads (thunks which call continuations)
Co-operative multithreading plan

- Keep a list of threads (thunks which call continuations)
- The first thread in the list is always the currently-running one.
Co-operative multithreading plan

- Keep a list of threads (thunks which call continuations)
- The first thread in the list is always the currently-running one.
- When we yield, we store the current continuation in the list, rotate the list of continuations by one, and jump into the new first continuation.
Co-operative multithreading plan

- Keep a list of threads (thunks which call continuations)
- The first thread in the list is always the currently-running one.
- When we **yield**, we store the current continuation in the list, rotate the list of continuations by one, and jump into the new first continuation.
- When we **fork**, add a new thunk to end of the list, which calls the fork’d code then removes itself from the list of threads.
(define threads ’())
(define (reset-threads!)
  (set! threads (list ’threads)))
(define (threads-list) (rest threads))
(define (current-thread) (first (threads-list)))
Adding, saving, and removing threads

\begin{verbatim}
(define (add-thread! thunk)
  (set-cdr! (last-pair threads)
    (list (lambda ()
      (thunk)
      (drop-and-schedule!))))

(define (save-thread! thunk)
  (set-car! (threads-list) thunk))

(define (pop-thread!)
  (set-cdr! threads (rest (threads-list))))
\end{verbatim}
(define (rotate-threads! keep)
  (if keep (set-cdr! (last-pair threads)
        (list (current-thread)))))

(pop-thread!)
(if (null? (threads-list))
  threads-done
  (current-thread)))

(define (drop-and-schedule!)
  ((rotate-threads! #f)))

(define (schedule)
  ((rotate-threads! #t)))
(define threads-done #f)
(define (start-threads thunk)
  (reset-threads!)
  (add-thread! thunk)
  (if (call/cc
       (lambda (cont)
         (set! threads-done (lambda () (cont #t)))
         #f))
    'all-done
    (schedule)))
(define fork add-thread!)

(define (yield)
  (let ((cont #f))
    (if (call/cc (lambda (k) (set! cont k) #f))
        'yield-done
        (begin
          (save-thread! (lambda () (cont #t)))
          (schedule)))))
(define (test)
  (define (h1 n thread-name)
    (if (= n 0) 0
      (begin
        (display thread-name)
        (display " - ")
        (display n)
        (newline)
        (yield)
        (h1 (- n 1) thread-name))))
  (fork (lambda () (h1 5 'thread1)))
  (fork (lambda () (h1 7 'thread2)))
  (fork (lambda () (h1 9 'thread3))))

(start-threads test)
<table>
<thead>
<tr>
<th>Thread</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread1</td>
<td>5</td>
</tr>
<tr>
<td>thread2</td>
<td>7</td>
</tr>
<tr>
<td>thread3</td>
<td>9</td>
</tr>
<tr>
<td>thread1</td>
<td>4</td>
</tr>
<tr>
<td>thread2</td>
<td>6</td>
</tr>
<tr>
<td>thread3</td>
<td>8</td>
</tr>
<tr>
<td>thread1</td>
<td>3</td>
</tr>
<tr>
<td>thread2</td>
<td>5</td>
</tr>
<tr>
<td>thread3</td>
<td>7</td>
</tr>
<tr>
<td>thread1</td>
<td>2</td>
</tr>
<tr>
<td>thread2</td>
<td>4</td>
</tr>
<tr>
<td>thread3</td>
<td>6</td>
</tr>
<tr>
<td>thread1</td>
<td>1</td>
</tr>
</tbody>
</table>

```latex
\text{all-done}
```

> (\texttt{start-threads test})

```latex
\text{thread2} \rightarrow 3
\text{thread3} \rightarrow 5
\text{thread2} \rightarrow 2
\text{thread3} \rightarrow 4
\text{thread2} \rightarrow 1
\text{thread3} \rightarrow 3
\text{thread3} \rightarrow 2
\text{thread3} \rightarrow 1
\text{all-done}
> 
```
“Mr. Osborne, may I be excused? My brain is full.”
Processor speed

Microprocessor Transistor Counts 1971-2011 & Moore’s Law

curve shows transistor count doubling every two years


Alex Vandiver (MIT)  Continuations and Concurrency  Lecture 7  31 / 56
Processor speed

Nowadays every laptop has multiple “cores” in it.
Multiple processors

- Nowadays every laptop has multiple “cores” in it.
- Fastest “supercomputers” all have thousands of processors.
Multiple processors

- Nowadays every laptop has multiple “cores” in it
- Fastest “supercomputers” all have thousands of processors
- Not a new problem – Connection Machine (Lisp!) had 65,000 processors (1980s)
Concurrency

- All of our code only makes use of one processor
Concurrency

- All of our code only makes use of one processor
- **Concurrency** is the ability to do more than one computation in parallel
Concurrency

- All of our code only makes use of one processor
- **Concurrency** is the ability to do more than one computation in parallel
- Is a lot easier on the computer than on the programmer!
Objects with state

- In purely functional programming, time of evaluation is irrelevant:
  
  `(define (add-17 x) (+ x 17))
  (add-17 10)
  ; => 27
  ; ...later:
  (add-17 10)
  ; => 27

  Just run sequences on a different processors and we're done!

  ...except, this does not work for objects with state:
  `(define alexmv (create-person-and-brain 'alexmv great-court create-student-brain 3 3))
  (ask (ask alexmv 'LOCATION) 'NAME)
  ; => great-court
  ; ...later:
  (ask (ask alexmv 'LOCATION) 'NAME)
  ; => great-dome

  Alex Vandiver (MIT)
  Continuations and Concurrency
  Lecture 7 35 / 56
Objects with state

- In purely functional programming, time of evaluation is irrelevant:

  (define (add-17 x) (+ x 17))
  (add-17 10)
  ; => 27

...except, this does not work for objects with state:

  (define alexmv (create-person-and-brain 'alexmv great-court create-student-brain 3 3))
  (ask (ask alexmv 'LOCATION) 'NAME)
  ; => great-court

  ...later:

  (ask (ask alexmv 'LOCATION) 'NAME)
  ; => great-dome
Objects with state

- In purely functional programming, time of evaluation is irrelevant:
  
  ```scheme
  (define (add-17 x) (+ x 17))
  (add-17 10)
  ; => 27
  ; ...later:
  (add-17 10)
  ```
Objects with state

- In purely functional programming, time of evaluation is irrelevant:
  
  ```scheme
  (define (add-17 x) (+ x 17))
  (add-17 10)
  ; => 27
  ; ...later:
  (add-17 10)
  ; => 27
  ```
Objects with state

- In purely functional programming, time of evaluation is irrelevant:
  
  ```scheme
  (define (add-17 x) (+ x 17))
  (add-17 10)
  ; => 27
  ; ...later:
  (add-17 10)
  ; => 27
  ```

- Just run sequences on a different processors and we’re done!
Objects with state

- In purely functional programming, time of evaluation is irrelevant:
  
  ```scheme
  (define (add-17 x) (+ x 17))
  (add-17 10)
  ; => 27
  ; ...later:
  (add-17 10)
  ; => 27
  
  - Just run sequences on a different processors and we’re done!
  - ...except, this does **not** work for objects with state:
  ```
Objects with state

- In purely functional programming, time of evaluation is irrelevant:
  ```scheme
  (define (add-17 x) (+ x 17))
  (add-17 10)
  ; => 27
  ; ...later:
  (add-17 10)
  ; => 27
  ```

- Just run sequences on a different processors and we’re done!

- ...except, this does **not** work for objects with state:
  ```scheme
  (define alexmv (create-person-and-brain
                   'alexmv great-court create-student-brain 3 3))
  (ask (ask alexmv 'LOCATION) 'NAME)
  ; => great-court
  ```
Objects with state

- In purely functional programming, time of evaluation is irrelevant:
  (define (add-17 x) (+ x 17))
  (add-17 10)
  ; => 27
  ; ...later:
  (add-17 10)
  ; => 27

- Just run sequences on a different processors and we’re done!

- ...except, this does not work for objects with state:
  (define alexmv (create-person-and-brain
                      'alexmv great-court create-student-brain 3 3))
  (ask (ask alexmv 'LOCATION) 'NAME)
  ; => great-court
  ; ...later:
  (ask (ask alexmv 'LOCATION) 'NAME)
In purely functional programming, time of evaluation is irrelevant:

```scheme
(define (add-17 x) (+ x 17))
(add-17 10)
; => 27
; ...later:
(add-17 10)
; => 27
```

Just run sequences on a different processors and we’re done!

...except, this does not work for objects with state:

```scheme
(define alexmv (create-person-and-brain
    'alexmv great-court create-student-brain 3 3))
(ask (ask alexmv 'LOCATION) 'NAME)
; => great-court
; ...later:
(ask (ask alexmv 'LOCATION) 'NAME)
; => great-dome
```
Concurrency and time

- Behavior of objects with state depends on sequence of events in real time
Concurrency and time

- Behavior of objects with state depends on sequence of events in real time
- This is fine in a concurrent program where that state is not shared explicitly or implicitly between threads
Concurrency and time

- Behavior of objects with state depends on sequence of events in real time.
- This is fine in a concurrent program where that state is not shared explicitly or implicitly between threads.
- For example, autonomous objects in our adventure game conceptually act concurrently. We’d like to take advantage of this by letting them run at the same time.
Concurrency and time

- Behavior of objects with state depends on sequence of events in real time.
- This is fine in a concurrent program where that state is not shared explicitly or implicitly between threads.
- For example, autonomous objects in our adventure game conceptually act concurrently. We’d like to take advantage of this by letting them run at the same time.
- But how does the order of execution affect the interactions between them?
(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 msg)
    (cond ((eq? msg 'withdraw) withdraw)
          ((eq? msg 'deposit) deposit)
          ((eq? msg 'balance) balance)
          (else (error "unknown request" msg))))
  dispatch)
Why is time an issue?

(define alex (make-account 100))
(define ben alex)

((alex 'withdraw) 10)
((ben 'withdraw) 25)
Why is time an issue?

(define alex (make-account 100))
(define ben alex)

((alex 'withdraw) 10)
((ben 'withdraw) 25)
Why is time an issue?

(define alex (make-account 100))
(define ben alex)

((alex 'withdraw) 10)
((ben 'withdraw) 25)

<table>
<thead>
<tr>
<th>Alex</th>
<th>Bank</th>
<th>Ben</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>-10</td>
<td></td>
<td>90</td>
</tr>
</tbody>
</table>
Why is time an issue?

(define alex (make-account 100))
(define ben alex)

((alex 'withdraw) 10)
((ben 'withdraw) 25)

<table>
<thead>
<tr>
<th>Alex</th>
<th>Bank</th>
<th>Ben</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-10</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>-25</td>
<td></td>
</tr>
</tbody>
</table>
Why is time an issue?

(define alex (make-account 100))
(define ben alex)

((alex 'withdraw) 10)
((ben 'withdraw) 25)

<table>
<thead>
<tr>
<th>Alex</th>
<th>Bank</th>
<th>Ben</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-10</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>-25</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Alex</th>
<th>Bank</th>
<th>Ben</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Why is time an issue?

(define alex (make-account 100))
(define ben alex)

((alex 'withdraw) 10)
((ben 'withdraw) 25)

<table>
<thead>
<tr>
<th>Alex</th>
<th>Bank</th>
<th>Ben</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-10</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>-25</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Alex</th>
<th>Bank</th>
<th>Ben</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>-25</td>
<td></td>
</tr>
</tbody>
</table>
Why is time an issue?

(define alex (make-account 100))
(define ben alex)

((alex 'withdraw) 10)
((ben 'withdraw) 25)

<table>
<thead>
<tr>
<th>Alex</th>
<th>Bank</th>
<th>Ben</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-10</td>
<td>90</td>
<td></td>
</tr>
<tr>
<td>65</td>
<td>-25</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Alex</th>
<th>Bank</th>
<th>Ben</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-10</td>
<td>75</td>
<td>-25</td>
</tr>
<tr>
<td></td>
<td>-10</td>
<td>65</td>
</tr>
</tbody>
</table>
(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
             balance)
      "Insufficient funds")

((alex 'withdraw) 10)
((ben 'withdraw) 25)
(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
        balance)
    "Insufficient funds"))
((alex 'withdraw) 10)
((ben 'withdraw) 25)
(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
           balance)
      "Insufficient funds")
((alex 'withdraw) 10)
((ben 'withdraw) 25)

<table>
<thead>
<tr>
<th></th>
<th>Bank</th>
<th>Ben</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Access balance (100)</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
        balance)
      "Insufficient funds")
)((alex 'withdraw) 10)
((ben 'withdraw) 25)

Alex Bank Ben
Access balance (100) 100

100 Access balance (100)
Race conditions

(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
              balance)
      "Insufficient funds")
((alex 'withdraw) 10)
((ben 'withdraw) 25)

<table>
<thead>
<tr>
<th></th>
<th>Bank</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Alex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Access balance (100)</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>New value 100 - 10 = 90</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Ben</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access balance (100)</td>
<td>100</td>
</tr>
</tbody>
</table>

(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
             balance)
      "Insufficient funds")
((alex 'withdraw) 10)
((ben 'withdraw) 25)

<table>
<thead>
<tr>
<th></th>
<th>Alex</th>
<th>Bank</th>
<th>Ben</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Access balance (100)</td>
<td>100</td>
<td>100</td>
<td>Access balance (100)</td>
</tr>
<tr>
<td>New value 100 - 10 = 90</td>
<td>100</td>
<td>100</td>
<td>New value 100 - 25 = 75</td>
</tr>
</tbody>
</table>
(define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
            balance)
        "Insufficient funds")
)

((alex 'withdraw) 10)
((ben 'withdraw) 25)

<table>
<thead>
<tr>
<th>Alex</th>
<th>Bank</th>
<th>Ben</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access balance (100)</td>
<td>100</td>
<td>Access balance (100)</td>
</tr>
<tr>
<td>New value 100 - 10 = 90</td>
<td>100</td>
<td>New value 100 - 25 = 75</td>
</tr>
<tr>
<td>Set balance 90</td>
<td>90</td>
<td></td>
</tr>
</tbody>
</table>
(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
           balance)
       "Insufficient funds")
((alex 'withdraw) 10)
((ben 'withdraw) 25)

<table>
<thead>
<tr>
<th></th>
<th>Alex</th>
<th>Bank</th>
<th>Ben</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>balance</td>
<td>100</td>
<td>100</td>
<td>Access balance (100)</td>
</tr>
<tr>
<td>New value</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>90</td>
<td></td>
<td>New value 100 - 25 = 75</td>
</tr>
<tr>
<td>Set balance</td>
<td></td>
<td>75</td>
<td></td>
</tr>
</tbody>
</table>
(define (withdraw amount)
  (if (>= balance amount)
      (begin (set! balance (- balance amount))
        balance)
      "Insufficient funds"))

((alex 'withdraw) 10)
((ben 'withdraw) 25)

<table>
<thead>
<tr>
<th>Alex</th>
<th>Bank</th>
<th>Ben</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access balance (100)</td>
<td>100</td>
<td>Access balance (100)</td>
</tr>
<tr>
<td>New value 100 - 10 = 90</td>
<td>100</td>
<td>New value 100 - 25 = 75</td>
</tr>
<tr>
<td>Set balance 90</td>
<td>90</td>
<td>Set balance 75</td>
</tr>
<tr>
<td></td>
<td>75</td>
<td></td>
</tr>
</tbody>
</table>
Correct behavior of concurrent programs

Require:

- No two operations that change any shared state can occur at the same time.
- Guarantees correctness, but too conservative?
- A concurrent system produces the same result as if the processes had run sequentially in some order.
- Does not require the processes to actually run sequentially, only to produce results as if they had run sequentially.
- There may be more than one "correct" result as a consequence!
Correct behavior of concurrent programs

Require:
- That no two operations that change any shared state can occur at the same time
Correct behavior of concurrent programs

Require:
- That no two operations that change any shared state can occur at the same time
- Guarantees correctness, but too conservative?
Correct behavior of concurrent programs

Require:
- That no two operations that change any shared state can occur at the same time
  - Guarantees correctness, but too conservative?
- That a concurrent system produces the same result as if the processes had run sequentially in some order
Correct behavior of concurrent programs

Require:

- That no two operations that change any shared state can occur at the same time
  - Guarantees correctness, but too conservative?
- That a concurrent system produces the same result as if the processes had run sequentially in some order
  - Does not require the processes to actually run sequentially, only to produce results as if they had run sequentially
  - There may be more than one “correct” result as a consequence!
(define x 10)
(define f1 (lambda () (set! x (* x x))))
(define f2 (lambda () (set! x (+ x 1))))

(parallel-execute f1 f2)
(define x 10)
(define f1 (lambda () (set! x (* x x))))
(define f2 (lambda () (set! x (+ x 1))))

(parallel-execute f1 f2)

\[ a \rightarrow \text{Look up first } x \text{ in } f1 \]

f1
(define x 10)
(define f1 (lambda () (set! x (* x x))))
(define f2 (lambda () (set! x (+ x 1))))

(parallel-execute f1 f2)

  a – Look up first \texttt{x} in \texttt{f1}

\texttt{f1}  b – Look up second \texttt{x} in \texttt{f1}
(define x 10)
(define f1 (lambda () (set! x (* x x))))
(define f2 (lambda () (set! x (+ x 1))))

(parallel-execute f1 f2)

a – Look up first x in f1
b – Look up second x in f1
c – Assign product of a and b to x
(define x 10)
(define f1 (lambda () (set! x (* x x))))
(define f2 (lambda () (set! x (+ x 1))))

(parallel-execute f1 f2)

\[
\begin{align*}
\text{a} & \rightarrow \text{Look up first } x \text{ in } f1 \\
\text{b} & \rightarrow \text{Look up second } x \text{ in } f1 \\
\text{c} & \rightarrow \text{Assign product of } a \text{ and } b \text{ to } x
\end{align*}
\]
(define x 10)
(define f1 (lambda () (set! x (* x x))))
(define f2 (lambda () (set! x (+ x 1))))

(parallel-execute f1 f2)

<table>
<thead>
<tr>
<th>a – Look up first x in f1</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1</td>
</tr>
<tr>
<td>b – Look up second x in f1</td>
</tr>
<tr>
<td>c – Assign product of a and b to x</td>
</tr>
<tr>
<td>f2</td>
</tr>
<tr>
<td>d – Look up x in f2</td>
</tr>
</tbody>
</table>
(define x 10)
(define f1 (lambda () (set! x (* x x))))
(define f2 (lambda () (set! x (+ x 1))))

(parallel-execute f1 f2)

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>–</td>
<td>Look up first x in f1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>–</td>
<td>Look up second x in f1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>–</td>
<td>Assign product of a and b to x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>–</td>
<td>Look up x in f2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>–</td>
<td>Assign sum of d and 1 to x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(define x 10)
(define f1 (lambda () (set! x (* x x))))
(define f2 (lambda () (set! x (+ x 1))))

(parallel-execute f1 f2)

<table>
<thead>
<tr>
<th></th>
<th>a – Look up first x in f1</th>
<th>b – Look up second x in f1</th>
<th>c – Assign product of a and b to x</th>
<th>Internal order preserved</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>d – Look up x in f2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>e – Assign sum of d and 1 to x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(define x 10)
(define f1 (lambda () (set! x (* x x))))
(define f2 (lambda () (set! x (+ x 1))))
(parallel-execute f1 f2)

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f1</td>
<td>f2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a – Look up first (x) in (f1)</td>
<td>d – Look up (x) in (f2)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b – Look up second (x) in (f1)</td>
<td>e – Assign sum of (d) and 1 to (x)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c – Assign product of (a) and (b) to (x)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Internal order preserved
(define x 10)
(define f1 (lambda () (set! x (* x x))))
(define f2 (lambda () (set! x (+ x 1))))

(parallel-execute f1 f2)

<table>
<thead>
<tr>
<th></th>
<th>a – Look up first x in f1</th>
<th>b – Look up second x in f1</th>
<th>c – Assign product of a and b to x</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1</td>
<td></td>
<td></td>
<td>Internal order preserved</td>
</tr>
<tr>
<td></td>
<td>d – Look up x in f2</td>
<td>e – Assign sum of d and 1 to x</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Internal order preserved</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>a d b e c</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>a d b e c</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td>d a b e c</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td>d a e b c</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td>d e a b c</td>
<td></td>
</tr>
</tbody>
</table>

Internal order preserved.
(define x 10)
(define f1 (lambda () (set! x (* x x))))
(define f2 (lambda () (set! x (+ x 1))))

(parallel-execute f1 f2)

<table>
<thead>
<tr>
<th>f1</th>
<th>a – Look up first x in f1</th>
<th>Internal order preserved</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>b – Look up second x in f1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c – Assign product of a and b to x</td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td>d – Look up x in f2</td>
<td>Internal order preserved</td>
</tr>
<tr>
<td></td>
<td>e – Assign sum of d and 1 to x</td>
<td></td>
</tr>
</tbody>
</table>

| 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 |
Processes will execute concurrently, but there will certain sets of procedures such that **only one** execution of a procedure in each set is permitted to happen at a time.
Processes will execute concurrently, but there will certain sets of procedures such that **only one** execution of a procedure in each set is permitted to happen at a time.

If some procedure in the set is being executed, then any other process that attempts to execute any procedure in the set will be forced to wait until the first execution has finished.
Processes will execute concurrently, but there will certain sets of procedures such that **only one** execution of a procedure in each set is permitted to happen at a time.

If some procedure in the set is being executed, then any other 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.
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.
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` returns a procedure that takes a procedure as input and returns a serialized procedure that behaves like the original, except that if another procedure in the same serialized set is underway, this procedure must wait.
Serializers “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` returns a procedure that takes a procedure as input and returns a serialized procedure that behaves like the original, except that if another procedure in the same serialized set is underway, this procedure must wait.

- Where do we put the serializers?
(define x 10)
(define kelloggs (make-serializer))
(define f1 (kelloggs (lambda () (set! x (* x x)))))
(define f2 (kelloggs (lambda () (set! x (+ x 1)))))
(parallel-execute f1 f2)
(define x 10)
(define kelloggs (make-serializer))
(define f1 (kelloggs (lambda () (set! x (* x x))))))
(define f2 (kelloggs (lambda () (set! x (+ x 1))))))
(parallel-execute f1 f2)

---

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>c</td>
<td>d</td>
<td>e</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>f1</td>
<td>a – Look up first x in f1</td>
<td>b – Look up second x in f1</td>
<td>c – Assign product of a and b to x</td>
<td></td>
</tr>
<tr>
<td>f2</td>
<td></td>
<td>d – Look up x in f2</td>
<td>e – Assign sum of d and 1 to x</td>
<td></td>
</tr>
</tbody>
</table>
(define x 10)
(define kelloggs (make-serializer))
(define f1 (kelloggs (lambda () (set! x (* x x)))))
(define f2 (kelloggs (lambda () (set! x (+ x 1)))))

(parallel-execute f1 f2)

\begin{align*}
\text{f1} & \quad a \rightarrow \text{Look up first } x \text{ in } f1 \\
 & \quad b \rightarrow \text{Look up second } x \text{ in } f1 \\
 & \quad c \rightarrow \text{Assign product of } a \text{ and } b \text{ to } x \\
\text{f2} & \quad d \rightarrow \text{Look up } x \text{ in } f2 \\
 & \quad e \rightarrow \text{Assign sum of } d \text{ and } 1 \text{ to } x
\end{align*}

\begin{align*}
a & \quad b & \quad c & \quad d & \quad e & \quad 10 & \quad 10 & \quad 100 & \quad 100 & \quad 101 & \quad d & \quad e & \quad a & \quad b & \quad c & \quad 10 & \quad 11 & \quad 11 & \quad 11 & \quad 121
\end{align*}
(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 ((kelloggs (make-serializer)))
    (define (dispatch msg)
      (cond ((eq? msg 'withdraw) (kelloggs withdraw))
             ((eq? msg 'deposit) (kelloggs deposit))
             ((eq? msg 'balance) balance)
             (else (error "unknown request" msg)))))))
  dispatch)
Multiple shared resources

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

(parallel-execute (lambda () (exchange alex ben))
   (lambda () (exchange alex mpp)))

; alex = 300, ben = 200, mpp = 100
(define (exchange account1 account2)
  (let ((difference (- (account1 'balance)
                     (account2 'balance))))
    ((account1 'withdraw) difference)
    ((account2 'deposit) difference)))

(parallel-execute (lambda () (exchange alex ben))
                    (lambda () (exchange alex mpp)))

; alex = 300, ben = 200, mpp = 100

1. Difference (- alex ben) = 100
(define (exchange account1 account2)
  (let ((difference (- (account1 'balance)
                      (account2 'balance))))
    ((account1 'withdraw) difference)
    ((account2 'deposit) difference)))

(parallel-execute (lambda () (exchange alex ben))
                   (lambda () (exchange alex mpp)))

; alex = 300, ben = 200, mpp = 100

1. Difference (- alex ben) = 100
2. Withdraw 100 from alex (has 200)
(define (exchange account1 account2)
  (let ((difference (- (account1 'balance)
         (account2 'balance))))
    ((account1 'withdraw) difference)
    ((account2 'deposit) difference)))
(parallel-execute (lambda () (exchange alex ben))
  (lambda () (exchange alex mpp)))
; alex = 300, ben = 200, mpp = 100

1. Difference (- alex ben) = 100
2. Withdraw 100 from alex (has 200)
3. Deposit 100 into ben (has 300)
(define (exchange account1 account2)
  (let ((difference (- (account1 'balance)
                      (account2 'balance))))
    ((account1 'withdraw) difference)
    ((account2 'deposit) difference)))

(parallel-execute (lambda () (exchange alex ben))
  (lambda () (exchange alex mpp)))

; alex = 300, ben = 200, mpp = 100

1. Difference (- alex ben) = 100
2. Withdraw 100 from alex (has 200)
3. Deposit 100 into ben (has 300)
4. Difference (- alex mpp) = 100
Multiple shared resources

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

(parallel-execute (lambda () (exchange alex ben))
                   (lambda () (exchange alex mpp)))

; alex = 300, ben = 200, mpp = 100

1. Difference (- alex ben) = 100
2. Withdraw 100 from alex (has 200)
3. Deposit 100 into ben (has 300)
4. Difference (- alex mpp) = 100
5. Withdraw 100 from alex (has 100)
Multiple shared resources

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

(parallel-execute (lambda () (exchange alex ben))
    (lambda () (exchange alex mpp)))

; alex = 300, ben = 200, mpp = 100

1. **Difference** (- alex ben) = 100
2. **Withdraw 100 from** alex (has 200)
3. **Deposit 100 into** ben (has 300)
4. **Difference** (- alex mpp) = 100
5. **Withdraw 100 from** alex (has 100)
6. **Deposit 100 into** mpp (has 200)
(define (exchange account1 account2)
  (let ((difference (- (account1 'balance)
                        (account2 'balance))))
    ((account1 'withdraw) difference)
    ((account2 'deposit) difference)))

(parallel-execute (lambda () (exchange alex ben))
                  (lambda () (exchange alex mpp)))

; alex = 300, ben = 200, mpp = 100

1. Difference (- alex ben) = 100
Multiple shared resources

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

(parallel-execute (lambda () (exchange alex ben))
                   (lambda () (exchange alex mpp)))

; alex = 300, ben = 200, mpp = 100

1. Difference (- alex ben) = 100
4. Difference (- alex mpp) = 200
Multiple shared resources

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

(parallel-execute (lambda () (exchange alex ben))
  (lambda () (exchange alex mpp)))

; alex = 300, ben = 200, mpp = 100

1. Difference (- alex ben) = 100
4. Difference (- alex mpp) = 200
5. Withdraw 200 from alex (has 100)
Multiple shared resources

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

(parallel-execute (lambda () (exchange alex ben))
  (lambda () (exchange alex mpp)))

; alex = 300, ben = 200, mpp = 100

1. Difference (- alex ben) = 100
4. Difference (- alex mpp) = 200
5. Withdraw 200 from alex (has 100)
6. Deposit 200 into mpp (has 300)
Multiple shared resources

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

(parallel-execute (lambda () (exchange alex ben))
  (lambda () (exchange alex mpp)))

; alex = 300, ben = 200, mpp = 100

1. Difference (- alex ben) = 100
4. Difference (- alex mpp) = 200
5. Withdraw 200 from alex (has 100)
6. Deposit 200 into mpp (has 300)
2. Withdraw 100 from alex (has 0)
(define (exchange account1 account2)
  (let ((difference (- (account1 'balance)
                      (account2 'balance))))
    ((account1 'withdraw) difference)
    ((account2 'deposit) difference))))

(parallel-execute (lambda () (exchange alex ben))
                    (lambda () (exchange alex mpp)))

; alex = 300, ben = 200, mpp = 100

1. Difference (- alex ben) = 100
4. Difference (- alex mpp) = 200
5. Withdraw 200 from alex (has 100)
6. Deposit 200 into mpp (has 300)
2. Withdraw 100 from alex (has 0)
3. Deposit 100 into ben (has 300)
(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 ((kelloggs (make-serializer)))
    (define (dispatch msg)
      (cond ((eq? msg 'withdraw) withdraw)
            ((eq? msg 'deposit) deposit)
            ((eq? msg 'balance) balance)
            ((eq? msg 'serializer) kelloggs)
            (else (error "unknown request" msg)))
      dispatch))
Serialize access to all 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)))
Serialize access to all 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)))
Suppose Alex attempts to exchange $a_1$ with $a_2$. And Ben attempts to exchange $a_2$ with $a_1$.

Imagine that Alex gets the serializer for $a_1$ at the same time that Ben gets the serializer for $a_2$. Now Alex is stalled waiting for the serializer from $a_2$, but Ben is holding it. And Ben is similarly waiting for the serializer from $a_1$, but Alex is holding it. **DEADLOCK!**
Suppose Alex attempts to exchange \( a_1 \) with \( a_2 \)

And Ben attempts to exchange \( a_2 \) with \( a_1 \)
Suppose Alex attempts to exchange \( a_1 \) with \( a_2 \)
And Ben attempts to exchange \( a_2 \) with \( a_1 \)
Imagine that Alex gets the serializer for \( a_1 \) at the same time that Ben gets the serializer for \( a_2 \).
Suppose Alex attempts to exchange $a_1$ with $a_2$

And Ben attempts to exchange $a_2$ with $a_1$

Imagine that Alex gets the serializer for $a_1$ at the same time that Ben gets the serializer for $a_2$.

Now Alex is stalled waiting for the serializer from $a_2$, but Ben is holding it.
Suppose Alex attempts to exchange $a_1$ with $a_2$

And Ben attempts to exchange $a_2$ with $a_1$

Imagine that Alex gets the serializer for $a_1$ at the same time that Ben gets the serializer for $a_2$.

Now Alex is stalled waiting for the serializer from $a_2$, but Ben is holding it.

And Ben is similarly waiting for the serializer from $a_1$, but Alex is holding it.
Suppose Alex attempts to exchange $a_1$ with $a_2$
And Ben attempts to exchange $a_2$ with $a_1$
Imagine that Alex gets the serializer for $a_1$ at the same time that Ben gets the serializer for $a_2$.
Now Alex is stalled waiting for the serializer from $a_2$, but Ben is holding it.
And Ben is similarly waiting for the serializer from $a_1$, but Alex is holding it.

DEADLOCK!
Deadlocks

- Classic deadlock case: **Dining Philosophers problem**
Classic deadlock case: **Dining Philosophers problem**
Racket has concurrency, using **threads**
Racket has concurrency, using **threads**

We can implement serializers using a primitive synchronization method, called a **semaphore**
Racket has concurrency, using **threads**

- We can implement serializers using a primitive synchronization method, called a **semaphore**
- A semaphore counts the number of available resources
Racket has concurrency, using **threads**

We can implement serializers using a primitive synchronization method, called a **semaphore**

A semaphore counts the number of available resources

Semaphores can be **atomically** incremented and decremented
Racket has concurrency, using **threads**

We can implement serializers using a primitive synchronization method, called a **semaphore**

A semaphore counts the number of available resources

Semaphores can be **atomically** incremented and decremented

Attempts to decrement when there are no available resources causes the thread to **block**
Racket has concurrency, using **threads**

We can implement serializers using a primitive synchronization method, called a **semaphore**

A semaphore counts the number of available resources

Semaphores can be **atomically** incremented and decremented

Attempts to decrement when there are no available resources causes the thread to **block**

If the semaphore has only up to one resource, it is a **mutex** ("**mutual exclusion**")
(define (make-serializer)
  (let ((mutex (make-mutex)))
    (lambda (p)
      (lambda args
        (mutex 'acquire)
        (let ((val (apply p args)))
          (mutex 'release)
          val))))))
(define (make-mutex)
  (let ((sema (make-semaphore 1)))
    (lambda (m)
      (cond ((eq? m 'acquire) (semaphore-wait sema))
           ((eq? m 'release) (semaphore-post sema))
           (else (error "Invalid argument:" m))))))
(define (make-mutex)
  (let ((sema (make-semaphore 1)))
    (lambda (m)
      (cond ((eq? m 'acquire) (semaphore-wait sema))
            ((eq? m 'release) (semaphore-post sema))
            (else (error "Invalid argument:" m))))))
(define (make-mutex)
  (let ((sema (make-semaphore 1)))
    (lambda (m)
      (cond ((eq? m 'acquire) (semaphore-wait sema))
            ((eq? m 'release) (semaphore-post sema))
            (else (error "Invalid argument:" m))))))
(define (make-mutex)
  (let ((sema (make-semaphore 1)))
    (lambda (m)
      (cond ((eq? m 'acquire) (semaphore-wait sema))
            ((eq? m 'release) (semaphore-post sema))
            (else (error "Invalid argument:" m))))))
(define (parallel-execute p1 p2)
  (let* ((parent (current-thread))
         (t1 (thread (send-parent p1 parent)))
         (t2 (thread (send-parent p2 parent)))
         (list (thread-receive) (thread-receive))))

(define (send-parent f parent)
  (lambda () (thread-send parent (f))))
(define (parallel-execute p1 p2)
  (let* ((parent (current-thread))
         (t1 (thread (send-parent p1 parent)))
         (t2 (thread (send-parent p2 parent)))
         (list (thread-receive) (thread-receive))))

(define (send-parent f parent)
  (lambda () (thread-send parent (f)))))
\[
\begin{aligned}
&\text{(define } (\text{parallel-execute } p1 \ p2) \\
&\quad \text{(let* } ((\text{parent } (\text{current-thread})) \\
&\quad \quad (t1 \ (\text{thread } (\text{send-parent } p1 \ \text{parent}))) \\
&\quad \quad (t2 \ (\text{thread } (\text{send-parent } p2 \ \text{parent}))) \\
&\quad \quad (\text{list } (\text{thread-receive}) \ (\text{thread-receive})))))
\end{aligned}
\]

\[
\begin{aligned}
&\text{(define } (\text{send-parent } f \ \text{parent}) \\
&\quad (\lambda () \ (\text{thread-send } \text{parent} \ (f))))
\end{aligned}
\]
(define (parallel-execute p1 p2)
  (let* ((parent (current-thread))
          (t1 (thread (send-parent p1 parent)))
          (t2 (thread (send-parent p2 parent)))
          (list (thread-receive) (thread-receive))))

(define (send-parent f parent)
  (lambda () (thread-send parent (f))))

Parallel evaluation

Alex Vandiver (MIT)
Continuations and Concurrency
Lecture 7
(define (parallel-execute p1 p2)
  (let* ((parent (current-thread))
          (t1 (thread (send-parent p1 parent)))
          (t2 (thread (send-parent p2 parent))))
    (list (thread-receive) (thread-receive))))

(define (send-parent f parent)
  (lambda () (thread-send parent (f)))))
Thread communication

- Shared object state is implicit thread communication
Thread communication

- Shared object state is implicit thread communication
- More formal: `thread-send` and `thread-receive` are a simple asynchronous message queue
Thread communication

- Shared object state is implicit thread communication
- More formal: thread-send and thread-receive are a simple asynchronous message queue
- Inter-process communication through events
Thread communication

- Shared object state is implicit thread communication
- More formal: `thread-send` and `thread-receive` are a simple asynchronous message queue
- Inter-process communication through events
  - Sporadic requests to process something
Thread communication

- Shared object state is implicit thread communication
- More formal: `thread-send` and `thread-receive` are a simple asynchronous message queue
- Inter-process communication through events
  - Sporadic requests to process something
  - Events are a good model for user interaction
Concurrency in large systems

- It’s hard:

  - Coarse-grained locking is inefficient, but ensures correct behavior.
  - Fine-grained locking can be efficient, but is bug-prone and brittle to new kinds of operations.
  - Locks break abstraction barriers.
    - Opens up whole new classes of bugs (race conditions, deadlock, livelock, etc.).
  - These issues are recursive.
  - Also, very irritating to debug – non-deterministic, debugging output intermixed between threads, heisenbugs, etc.

But we need to do it.
Concurrency in large systems

- It’s hard:
  - Coarse-grained locking is inefficient, but ensures correct behavior
Concurrency in large systems

- It’s hard:
  - Coarse-grained locking is inefficient, but ensures correct behavior
  - Fine-grained locking can be efficient, but is bug-prone and brittle to new kinds of operations
Concurrence in large systems

- It’s hard:
  - Coarse-grained locking is inefficient, but ensures correct behavior
  - Fine-grained locking can be efficient, but is bug-prone and brittle to new kinds of operations
  - Locks break abstraction barriers
Concurrent in large systems

- It’s hard:
  - Coarse-grained locking is inefficient, but ensures correct behavior
  - Fine-grained locking can be efficient, but is bug-prone and brittle to new kinds of operations
  - Locks break abstraction barriers
  - Opens up whole new classes of bugs (race conditions, deadlock, livelock, etc.)
Concurrent in large systems

- It’s hard:
  - Coarse-grained locking is inefficient, but ensures correct behavior
  - Fine-grained locking can be efficient, but is bug-prone and brittle to new kinds of operations
  - Locks break abstraction barriers
  - Opens up whole new classes of bugs (race conditions, deadlock, livelock, etc.)
  - These issues are recursive

Also, very irritating to debug – non-deterministic, debugging output intermixed between threads, heisenbugs, etc.

But we need to do it
It’s hard:

- Coarse-grained locking is inefficient, but ensures correct behavior
- Fine-grained locking can be efficient, but is bug-prone and brittle to new kinds of operations
- Locks break abstraction barriers
- Opens up whole new classes of bugs (race conditions, deadlock, livelock, etc.)
- These issues are recursive
- Also, very irritating to debug – non-deterministic, debugging output intermixed between threads, heisenbugs, etc.
Concurrency in large systems

- It's hard:
  - Coarse-grained locking is inefficient, but ensures correct behavior
  - Fine-grained locking can be efficient, but is bug-prone and brittle to new kinds of operations
  - Locks break abstraction barriers
  - Opens up whole new classes of bugs (race conditions, deadlock, livelock, etc.)
  - These issues are recursive
  - Also, very irritating to debug – non-deterministic, debugging output intermixed between threads, heisenbugs, etc.

- But we need to do it