Concurrency

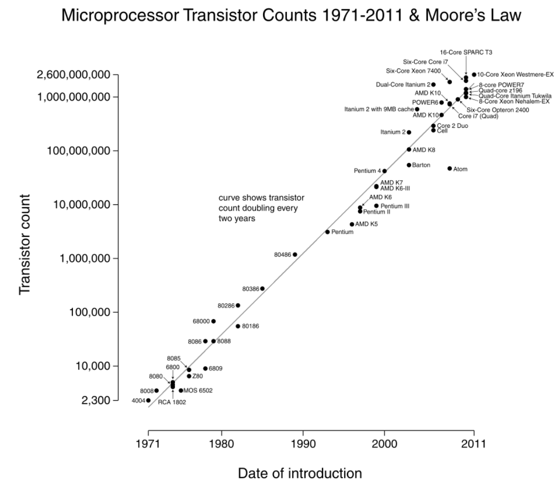
6.037 - Structure and Interpretation of Computer Programs

Mike Phillips <mpp>

Massachusetts Institute of Technology

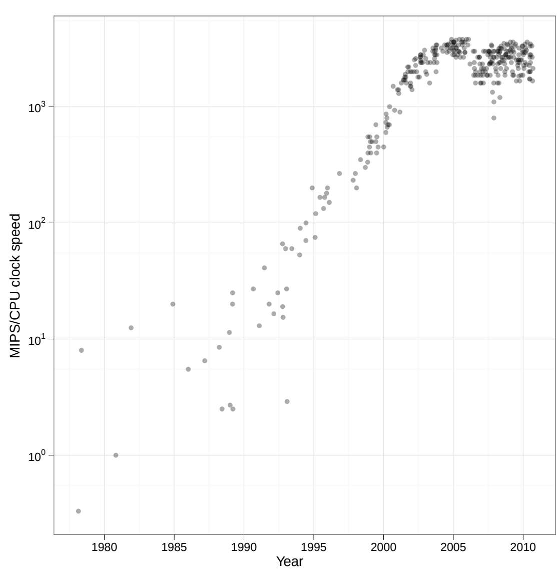
Lecture 8A

Processor speed



http://en.wikipedia.org/wiki/File:Transistor_Count_and_Moore%27s_Law_-_2011.svg

Processor speed



http://csgillespie.wordpress.com/2011/01/25/cpu-and-gpu-trends-over-time/ with data originally from Wikipedia

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** 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 different processors and we're done!
- ...except, this does **not** work for objects with state:

```
(define alexmv
  (new autonomous-person 'alexmv 'great-court 3 3))
((alexmv 'LOCATION) 'NAME)
; => great-court
; ...later:
((alexmv 'LOCATION) 'NAME)
; => great-dome
```


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?

Canonical 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)))
   (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)
```

Alex	Bank	Ben
	100	
-10	90	
	65	-25

Alex	Bank	Ben
	100	
	75	-25
-10	65	

Race conditions

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

Alex	Bank	Ben
	100	
Access balance (100)	100	
	100	Access balance (100)
New value 100 - 10 = 90	100	
	100	New value 100 - 25 = 75
Set balance 90	90	
	75	Set balance 75
	75	

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)
```

f1	`a` – Look up first `x` in `f1`
	`b` – Look up second `x` in `f1`
	`c` – Assign product of `a` and `b` to `x`
		Internal order preserved

f2	`d` – Look up `x` in `f2` `e` – Assign sum of `d` and 1 to `x`	Internal order preserved

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

Serializing access to shared state

- 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

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)
```

f1	`a` – Look up first `x` in `f1`
	`b` – Look up second `x` in `f1`
	`c` – Assign product of `a` and `b` to `x`

f2	`d` – Look up `x` in `f2` `e` – Assign sum of `d` and 1 to `x`

`a b c d e` 10 10 100 100 101   `d e a b c` 10 11 11 11 121


```
(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
```

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)

Serializing object access

```
(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

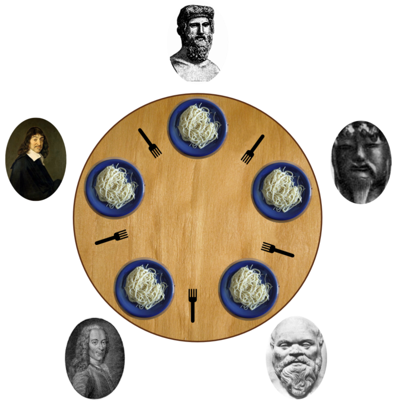
```
(define (deposit account amount)
  (let ((s (account 'serializer))
        (d (account 'deposit)))
    ((s d) amount)))

(define (serialized-exchange acct1 acct2)
   (let ((serializer1 (acct1 'serializer))
         (serializer2 (acct2 'serializer)))
     ((serializer1 (serializer2 exchange))
      acct1
      acct2)))
```


- Suppose Alex attempts to exchange `a1` with `a2`
- And Ben attempts to exchange `a2` with `a1`
- Imagine that Alex gets the serializer for `a1` at the same time that Ben gets the serializer for `a2`.
- Now Alex is stalled waiting for the serializer from `a2`, but Ben is holding it.
- And Ben is similarly waiting for the serializer from `a1`, but Alex is holding it.

DEADLOCK!

Deadlocks

- Classic deadlock case: **Dining Philosophers problem**



Implementation time

- 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** (“**mut**ual **ex**clusion”)

Serializer

```
(define (make-serializer)
  (let ((mutex (make-mutex)))
    (lambda (p)
      (lambda args
        (mutex 'acquire)
        (let ((val (apply p args)))
          (mutex 'release)
          val)))))
```


Mutex

```
(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))))))
```

Parallel evaluation

```
(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
- 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