Topic 20
Concurrency

Section 3.4

Concurrency

- In the real world, many processes act concurrently
- If they interact with each other, as when sharing a resource, strange things can happen

Simplified bank account example

- Simplified withdraw procedure

```
; withdraw an amount from the account
; balance
(define (withdraw amount)
  (set! balance
        (- balance amount)))
```

- **Withdraw process must read balance, then do a computation, then set balance to new value**

Interaction of two withdraw processes

- What happens if we have multiple people with a shared account?
- Each process does a read of balance and a set of balance to a new value
- There are six ways to interleave these events
- Some of these sequences of events make sense and some don't

Consider two withdrawals

- balance = 100 initially
- process 1 = (withdraw 20)
- process 2 = (withdraw 30)

Sequence 1

process 1	process 2
	reads balance = 100
	sets balance = 70
reads balance = 70	
sets balance = 50	

Sequence 2

process 1	process 2
	reads balance = 100
reads balance = 100	
	sets balance = 70
sets balance = 80	

???

Sequence 3

process 1	process 2
	reads balance = 100
reads balance = 100	
sets balance = 80	
	sets balance = 70

???

Sequence 4

process 1	process 2
reads balance = 100	
	reads balance = 100
	sets balance = 70
sets balance = 80	

???

Sequence 5

process 1	process 2
reads balance = 100	
	reads balance = 100
sets balance = 80	
	sets balance = 70

???

Sequence 6

process 1	process 2
reads balance = 100	
sets balance = 80	
	reads balance = 80
	sets balance = 50

Interactions can be more complicated

- Let x = 10
- Let process 1 = (lambda () (set! x (* x x)))
- Let process 2 = (lambda () (set! x (+ x 1)))
- Process 1 must do two reads of x
- Process 2 might change x between reads by process 1
- Five different final values for x are possible

Notation

- R1(n) means process 1 reads x, obtains n
- R2(n) means process 2 reads x, obtains n
- S1(n) means process 1 sets x to n
- S2(n) means process 2 sets x to n
- Process 1 does two reads, then a set
- Process 2 does one read, then a set

Five different results

x = 10	x = 10	x = 10	x = 10	x = 10
R1(10)	R2(10)	R2(10)	R1(10)	R1(10)
R1(10)	S2(11)	R1(10)	R1(10)	R1(10)
S1(100)	R1(11)	S2(11)	R2(10)	R2(10)
R2(100)	R1(11)	R1(11)	S1(100)	S2(11)
S2(101)	S1(121)	S1(110)	S2(11)	S1(100)
x = 101	x = 121	x = 110	x = 11	x = 100

The only way

The only way to prevent unwanted interactions is to serialize certain processes, that is, they have to occur sequentially rather than in parallel

Concurrent processing in DrScheme

- Must use Pretty-Big language
- Procedures to be run concurrently must have no arguments
- Each procedure is run in a separate thread

Parallel-execute

```
; cause parallel execution of args
(define (parallel-execute . args)
  (map thread args))
```

- **Map creates a thread for each procedure**
- **They start running immediately**
- **Example:**

```
; executes the two procedures in parallel
(parallel-execute
  (lambda () (set! x (* x x)))
  (lambda () (set! x (+ x 1))))
```


Serialization

- Issue – several processes may share (and be able to change) a common state variable. Need some way to isolate changes so that only one process can access/change the data at a time.
- There are several ways to force procedures to run sequentially – generally, allow us to interleave programs but constrain interleaving
- Generally – have the process “acquire a flag” – while that process has the flag, no other process can run until that flag is released.

Making a Serializer

- Use a primitive synchronization mechanism called a mutex.
- Mutex's can be acquired and released.
- A mutex is a mutable object (e.g., a 1 element list) that can hold the value of true or false.
- When the value is false, it is available for being acquired.
- When the value is true, it is unavailable and the process that wants it must wait…

What is required?

- To implement a mutex, need a mechanism for testing it and setting it that can not be interrupted…

Test-and-set!

- Conceptually behaves like

```
; if cell is already #t then return #t otherwise
; set it to #t and return #f
(define (test-and-set! cell)
  (if (car cell)
      #t
      (begin (set-car! cell #t) #f)))
```

- **For this to work, it must not be interruptable**
- **Exists as a single machine instruction on some machines**

Test-and-set! behavior

```
(define cell1 (list #f))
(car cell1) --> #f
(test-and-set! cell1) --> #f
(car cell1) --> #t
(test-and-set! cell1) --> #t

What do we do with a flag like this?  Use
  it for mutual exclusion purposes.  Share
  flag among several processes - let them
  run only when they get the flag.
  Specific example of a semaphore.
```


Mutual exclusion flag (mutex)

- A boolean flag that is controlled by one process at a time (because it is manipulated by test-and-set!)
- Accepts 'acquire and 'release messages
- A set of processes (that must be mutually exclusive) will share the same flag. Grab the flag before they start, and release it when they are done.
- In this way, only one process at a time can manipulate the variable.

Make-mutex

```
; provides a cell that can be used for
; mutual exclusion purposes among processes
(define (make-mutex)
  (let ((cell (list #f)))
    (define (the-mutex m)
      (cond ((eq? m 'acquire)
             (if (test-and-set! cell)
                 (the-mutex 'acquire))) ;retry
            ((eq? m 'release)
             (clear! cell))))
    the-mutex))

(define (clear! Cell) (set-car! Cell #f))
```

Serializers

- A serializer is a higher-order procedure that converts other procedures into ones that can only run one at a time
- All the procedures that are processed by the same serializer are run in sequence without interleaving

Make-serializer

```
; make a serializer by creating a mutex
; to be shared by the mutually exclusive
; procedures
(define (make-serializer)
  (let ((mutex (make-mutex)))
    (lambda (p)
      (define (serialized-p . args)
        (mutex 'acquire)
        (let ((value (apply p args)))
          (mutex 'release)
          value))
      serialized-p)))
```


Serialization guaranteed

```
(define s (make-serializer))

(parallel-execute
  (s (lambda () (set! x (* x x))))
  (s (lambda () (set! x (+ x 1)))))
```


Protecting bank accounts

```
; create a bank account that is protected and
; allows parallel execution
(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))
  balance)
```


continued

```
(let ((s (make-serializer)))
   (define (dispatch m)
     (cond ((eq? m 'withdraw) (s withdraw))
           ((eq? m 'deposit) (s deposit))
           ((eq? m 'balance) balance)
           (else
            (error "unknown msg-MAKE-ACCOUNT"
                   m))))
   dispatch))
```


Works for simple transactions

- Muliple processes withdrawing from and depositing to the same account are serialized
- Multiple accounts can still be withdrawn from and deposited to concurrently
- Problems can still arise if two or more resources are involved in a transaction

Even accounts

```
; evens up two accounts by splitting the difference
(define (even-accounts account1 account2)
  (if (< (account1 'balance)
         (account2 'balance))
      (let ((temp account1))  ; swap accounts
        (set! account1 account2)
        (set! account2 temp)))
  (let ((amount (/ (- (account1 'balance)
                      (account2 'balance))
                   2)))
    ((account1 'withdraw) amount)
    ((account2 'deposit) amount)))
```


What can happen

- If two `even-accounts` **processes run concurrently on the same pair of accounts, it is still possible for the two accounts to end up with balances that are different from each other**
- **To prevent this, processes need access to the serializers of both accounts**
- **Consider making the serializer available by exporting it via message passing**

Making serializer accessible

```
; account that can export its serializer via
; Message passing
(define (make-account-and-serializer balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
```


continued

```
(let ((s (make-serializer)))
  (define (dispatch m)
    (cond ((eq? m 'withdraw?) withdraw)
          ((eq? m 'deposit) deposit)
          ((eq? m 'balance) balance)
          ((eq? m 'serializer) s)
          (else
           (error "unknown msg-MAKE-ACCOUNT"
                  m))))
  dispatch))
```


Protected deposit

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


Protected withdraw

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

Disadvantage here is each user of bank-account objects have to explicitly manage the serialization.

But, it allows us to run a protected even-accounts!

Protected even accounts

```
(define (even-accounts account1 account2)
  (define (transfer-funds account1 account2)
    (if (< (account1 'balance) (account2 'balance))
        (let ((temp account1))  ; swap accounts
          (set! account1 account2)
          (set! account2 temp)))
    (let ((amount (/ (- (account1 'balance)
                        (account2 'balance))
                     2)))
      ((account1 'withdraw) amount)
      ((account2 'deposit) amount)))
```


continued

```
(let ((s1 (account1 'serializer))
      (s2 (account2 'serializer)))
  ((s1 (s2 transfer-funds)))))
```


Our problems aren't over yet

- Consider
```
(parallel-execute
  (even-accounts acc1 acc2)
  (even-accounts acc2 acc1))
```
- **Each process can grab one account (first arg) and will be forced to wait forever to grab the other account (second arg)**
- **This is called deadlock**

Preventing deadlock

- One way is to order all the mutexes in a program and always acquire mutexes in this order, release mutexes in reverse order
- Other methods of deadlock avoidance and recovery are being researched
- Deadlock recovery may be needed in situations where all the required resources are not known in advance; some have to be accessed first to decide which others are needed

Extended concurrent processes

- Concurrent processes that have complex interactions over a long period of time often need to synchronize their actions
- These processes are often allowed to communicate with each other to achieve synchronization
- Example: consumer-producer pair of processes