函数式程序设计

郭　炜

http://weibo.com/guoweiofpku

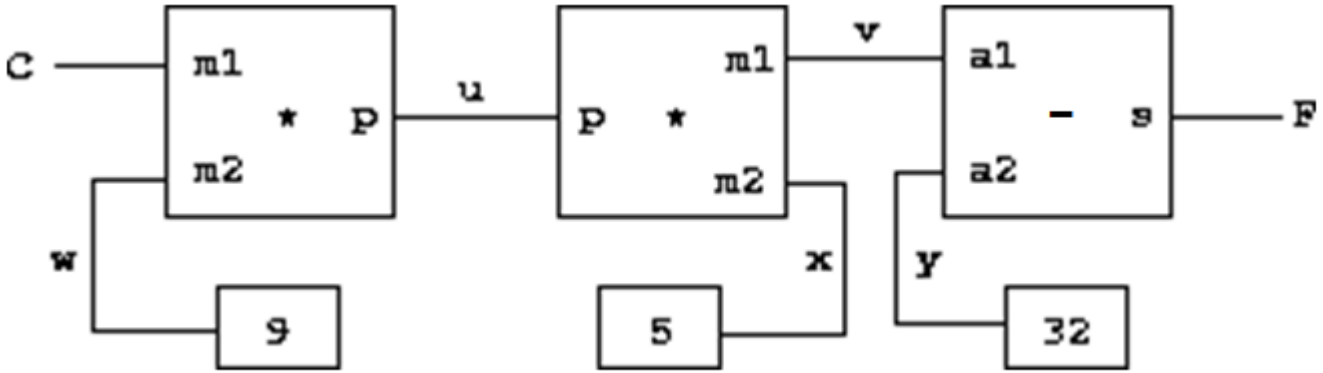
http://blog.sina.com.cn/u/3266490431

第九讲

约束传播系统

●考虑一个摄氏温度和华氏温度的转换系统，系统中有两个 connector，一个是C，代表摄氏温度，一个是F，代表华氏温度。一个connector上的值修改了，另一个connector上的值也自动修改。

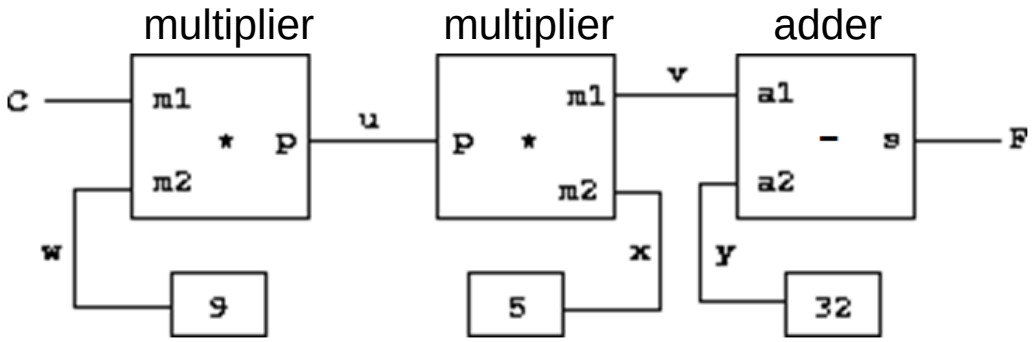
9C = 5(F-32)




```
(define C (make-connector))
(define F (make-connector))
(celsius-fahrenheit-converter C F) ;C-F约束器
```

约束传播系统

```
(define (celsius-fahrenheit-converter c f) ;定义一个约束器
  (let ((u (make-connector))
        (v (make-connector))
        (w (make-connector))
        (x (make-connector))
        (y (make-connector)))
    (multiplier c w u)
    (multiplier v x u)
    (adder v y f)
    (constant 9 w)
    (constant 5 x)
    (constant 32 y)
    'ok))
```



- adder规定了 v+y=F 这个关系， multipller情况类似。它们都是约束器
- constant 规定了 w的值只能是9,x的值只能是5，y的值只能是32，也是约束器
- 开始时，各个connector，包括C和F上都没有值

约束传播系统

```
;为C安装一个探测器，名为 Celsius temp。C值修改时探测器被激活
(probe "Celsius temp" C)
(probe "Fahrenheit temp" F)  ;为F安装一个探测器，名为 Fahrenheit temp
(set-value! C 25 'user)   ; 对象 'user将 C的值设置为 25
```

Probe: Celsius temp = 25
Probe: Fahrenheit temp = 77
done

C值的改变激活了探测器Celsius temp，导致输出。C 值的改变经由约束传播系统 celsius-fahrenheit-converter 改变了F的值，F的值改变激活探测器 Fahrenheit temp,导致输出

约束传播系统

(set-value! F 212 'user)
Error! Contradiction (77 212)

上面的输出也是探测器引发的。因为本系统规定，如果一个**connector**已经有了值，要修改它，则需要先让它 “忘掉”其已有的值：

(forget-value! C 'user)
Probe: Celsius temp = ?
Probe: Fahrenheit temp = ?
done
(set-value! F 212 'user)
Probe: Fahrenheit temp = 212
Probe: Celsius temp = 100　　发现**C**的值变为了**100**，以满足$9C = 5(F-32)$约束条件
done

约束传播系统

- 和connector相关的一些过程：

```
(define (has-value? connector)
  (connector 'has-value?))
(define (get-value connector)
  (connector 'value))
(define (set-value! connector new-value informant)
  ((connector 'set-value!) new-value informant))
(define (forget-value! connector retractor)
  ((connector 'forget) retractor))
(define (connect connector new-constraint)
  ((connector 'connect) new-constraint))
```

;informant、retractor和new-constraint都是命令connector进行改变的那个命令者，一般来说会是一个约束器（constraint），如adder,multiplier或celsius-fahrenheit-converter，也可以是 'user 之类

约束传播系统

●约束器的实现 **adder**：

```
(define (adder a1 a2 sum);返回值是个闭包 me
  (define (process-new-value);有connector值被改变时被调用，用来传播约束
    (cond ((and (has-value? a1) (has-value? a2))
           (set-value! sum
                       (+ (get-value a1) (get-value a2))
                       me))
          ((and (has-value? a1) (has-value? sum))
           (set-value! a2
                       (- (get-value sum) (get-value a1))
                       me))
          ((and (has-value? a2) (has-value? sum))
           (set-value! a1
                       (- (get-value sum) (get-value a2))
                       me))))
```

约束传播系统

```
(define (process-forget-value);有connector值被忘记时被调用，用来传播约束
  (forget-value! sum me)
  (forget-value! a1 me)
  (forget-value! a2 me)
  (process-new-value)) ;为何忘记值也需要调用process-new-value?
(define (me request)
  (cond ((eq? request 'I-have-a-value)
         (process-new-value))
        ((eq? request 'I-lost-my-value)
         (process-forget-value))
        (else
         (error "Unknown request -- ADDER" request))))
(connect a1 me);把a1连接到自身，即将me加入到a1的约束器列表
(connect a2 me)
(connect sum me)
me)   ;每个connector有个与之相连的约束器列表。connector上的值变化时，会调用约束器列表中约束器的 process-forget-value或 process-new-value
```

约束传播系统

●约束器的实现 multiplier：

```
(define (multiplier m1 m2 product)
  (define (process-new-value)
    (cond ((or (and (has-value? m1) (= (get-value m1) 0))
               (and (has-value? m2) (= (get-value m2) 0)))
           (set-value! product 0 me))
          ((and (has-value? m1) (has-value? m2))
           (set-value! product
                       (* (get-value m1) (get-value m2))
                       me))
          ((and (has-value? product) (has-value? m1))
           (set-value! m2
                       (/ (get-value product) (get-value m1))
                       me))
          ((and (has-value? product) (has-value? m2))
           (set-value! m1
                       (/ (get-value product) (get-value m2))
                       me))))
```

约束传播系统

```
(define (process-forget-value)
  (forget-value! product me)
  (forget-value! m1 me)
  (forget-value! m2 me)
  (process-new-value))
(define (me request)
  (cond ((eq? request 'I-have-a-value)
         (process-new-value))
        ((eq? request 'I-lost-my-value)
         (process-forget-value))
        (else
         (error "Unknown request -- MULTIPLIER" request))))
(connect m1 me)
(connect m2 me)
(connect product me)
me)
```

约束传播系统

- 实现常量约束器 **constant**：

```
(define (constant value connector)
  (define (me request)
    (error "Unknown request -- CONSTANT" request))
  (connect connector me);将me加入connector的约束器列表。不要是否可以？
  (set-value! connector value me);me命令connector将其值改为value
  me)
```

约束传播系统

●实现探测器probe：

```
(define (probe name connector)
  (define (print-probe value)
    (newline) (display "Probe: ")   (display name) (display " = ")
    (display value))
  (define (process-new-value)
    (print-probe (get-value connector)))
  (define (process-forget-value)
    (print-probe "?"))
  (define (me request)
    (cond ((eq? request 'I-have-a-value)
           (process-new-value))
          ((eq? request 'I-lost-my-value)
           (process-forget-value))
          (else
           (error "Unknown request -- PROBE" request))))
  (connect connector me)
  me)
```

约束传播系统

●和约束器相关的过程：

```
(define (inform-about-value constraint)
  (constraint 'I-have-a-value))

(define (inform-about-no-value constraint)
  (constraint 'I-lost-my-value))
```

;make-connector中用到。connector值发生变化时，connector通过这两个过程对其约束器列表中的约束器进行通知，以传递约束。

约束传播系统

- connector的实现：

```
(define (make-connector);返回值是闭包 me
  (let ((value false) (informant false) (constraints '()))
;三项分别为：初始值,上一个命令其改变值的命令者，它加入的约束器列表
    (define (set-my-value newval setter);setter是命令者
      (cond ((not (has-value? me))
             (set! value newval)
             (set! informant setter)
             (for-each-except setter ;依此通知列表中的约束器
                              inform-about-value
                              constraints))
            ((not (= value newval))
             (error "Contradiction" (list value newval)))
            (else 'ignored)))
```

约束传播系统

```
(define (forget-my-value retractor);retractor是命令者
  (if (eq? retractor informant);只有命令者才有权取消自己设置的值！！！！
      (begin (set! informant false)
             (for-each-except retractor
                              inform-about-no-value
                              constraints))
      'ignored))
(define (connect new-constraint)
  (if (not (memq new-constraint constraints)) ;memq是基本过程，用eq?
判断new-constraint 是否是 constraints成员
      (set! constraints
            (cons new-constraint constraints)))
  (if (has-value? me)
      (inform-about-value new-constraint))
  'done)
```

约束传播系统

```
(define (me request)
  (cond ((eq? request 'has-value?)
         (if informant true false))
        ((eq? request 'value) value)
        ((eq? request 'set-value!) set-my-value)
        ((eq? request 'forget) forget-my-value)
        ((eq? request 'connect) connect)
        (else (error "Unknown operation -- CONNECTOR"
                     request))))
me))
```

约束传播系统

在 (adder a1 a2 sum) 中：

```
(define (process-forget-value);有connector值被忘记时被调用，用来传播约束
  (forget-value! sum me) ;实际上，只有被me设置了值的connector才会忘掉值
  (forget-value! a1 me)
  (forget-value! a2 me)
  (process-new-value)) ;为何忘记值也需要调用process-new-value?本程序中去掉此句是否可以?
```

```
(define (process-new-value);有connector值被改变时被调用，用来传播约束
  (cond ((and (has-value? a1) (has-value? a2))
         (set-value! sum
                     (+ (get-value a1) (get-value a2))
                     me))
        ((and (has-value? a1) (has-value? sum))
         (set-value! a2
                     (- (get-value sum) (get-value a1))
                     me))
        ((and (has-value? a2) (has-value? sum))
         (set-value! a1
                     (- (get-value sum) (get-value a2))
                     me))))
```

约束传播系统

```
(define (for-each-except exception procedure list)
  (define (loop items)
    (cond ((null? items) 'done)
          ((eq? (car items) exception) (loop (cdr items)))
          (else (procedure (car items))
                (loop (cdr items)))))
  (loop list))
```

并发系统中的时间问题

- **使用赋值可以表示带内部状态的对象。其带来的缺陷：**
  - 代换模型不再适用
  - **“同一个”的问题也不再简单清晰**
- **出现这些情况，背后的核心问题是时间**
  - 无赋值程序的意义与时间无关，一个表达式总算出同一个值
  - 有赋值以后就必须考虑时间的影响。赋值可改变表达式所依赖的变量的状态，同一个变量在不同时间可能有不同的值。这样，表达式的值基于变量，因此就与求值的时间相关
- **用带有局部状态的计算对象建立计算模型时必须考虑时间带来的问题和影响**

并发系统中的时间问题

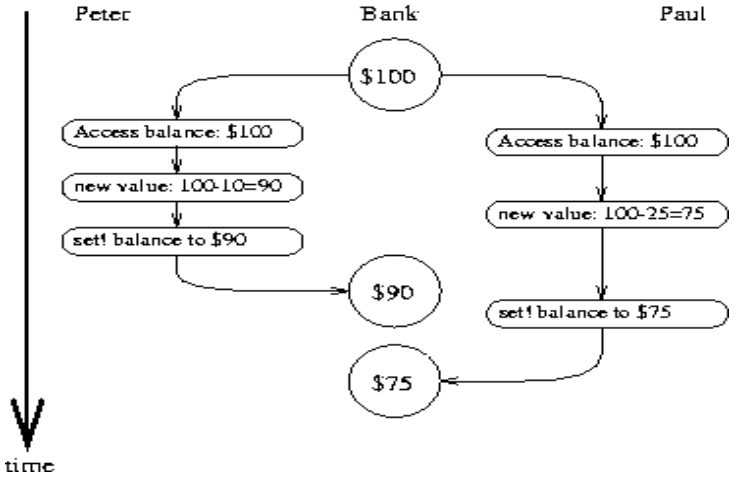
●多核处理器的流行，导致程序的并发执行称为一种趋势。程序的不同部分并发执行时，哪个部分先执行哪个部分后执行，具有不确定性。这可能会导致结果不正确。

假设**Peter** 和**Paul** 共享银行账号，的取款工作由两个独立运行的进程完成，账号余额变量balance由两个进程共享，两个进程执行取款都是调用 **withdraw**：

```
(define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount)) balance)
           "Insufficient funds"))
```

并发系统中的时间问题

```
(define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount)) balance)
           "Insufficient funds"))
```



两进程并行时会导致不同步：

1. Peter 检查余额确定取90 元合法（例如当时有100 元）

2. Paul 的进程实际地从账户中取出25 元

3. Peter 的进程实际取款90元（因为前面已经确定余额够用）

最后这个动作已经非法（当时的余额不足）

并发系统中的时间问题

●为防止不同步的现象发生，就需要控制多个进程执行的过程，进程不能随意并行。

●可能的限制策略：

一个进程正在修改共享变量时，
不允许其他进程进行任何操作

并发系统中的时间问题

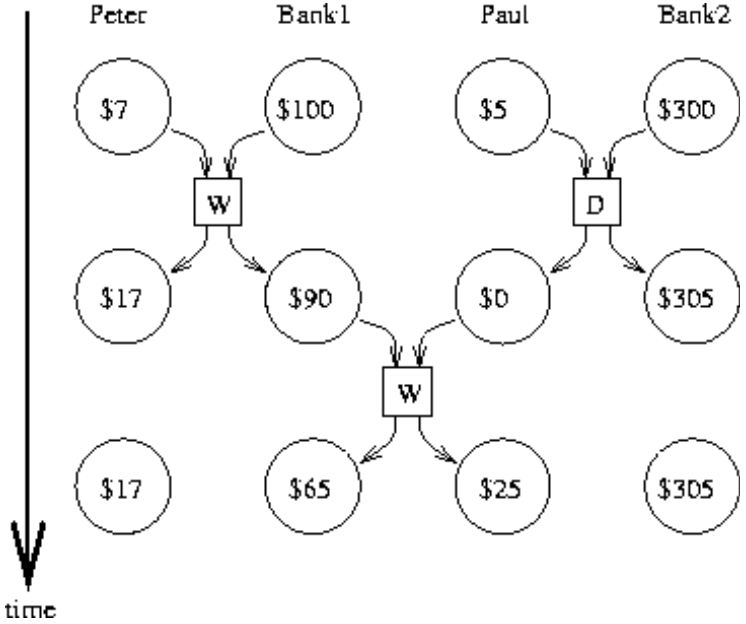
●为防止不同步的现象发生，就需要控制多个进程执行的过程，进程不能随意并行。

●可能的限制策略：

一个进程正在修改共享变量时，不允许其他进程进行任何操作

限制太死，不利于并行的效率

如果 peter在操作共享账户时，应该允许paul操作自己的账户



并发系统中的时间问题

●为防止不同步的现象发生，就需要控制多个进程执行的过程，进程不能随意并行。

改进：

peter和paul可以同时修改其各自账户，但是不能同时修改共享账户

●假定两 进程的事件序列为(a,b,c) 和(x,y,z) 并发执行可能产生的实际执行序列：

(a,b,c,x,y,z) (a,x,b,y,c,z) (x,a,b,c,y,z) (x,a,y,z,b,c)
(a,b,x,c,y,z) (a,x,b,y,c,z) (x,a,b,y,c,z) (x,y,a,b,c,z)
(a,b,x,y,c,z) (a,x,y,b,c,z) (x,a,b,y,z,c) (x,y,a,b,z,c)
(a,b,x,y,z,c) (a,x,y,b,c,z) (x,a,y,b,c,z) (x,y,a,z,b,c)
(a,x,b,c,y,z) (a,x,y,z,b,c) (x,a,y,b,z,c) (x,y,z,a,b,c)

设计并发系统时要考虑所有这些交错行为是否都可以接受

用串行控制器解决并发问题

- 假设scheme有以下过程：

```
(parallel-execute <p1> <p2> ... <pk>)
```

其作用是为每一个 pi创建一个独立的进程，这些进程会执行pi(pi必须无参数）

```
(define x 10)
(parallel-execute (lambda () (set! x (* x x))) ;p1
                  (lambda () (set! x (+ x 1)))) p2
```

建立两个并发进程，执行结果有5种：

101：先乘后加
121：先加后乘
110：p2的修改动作出现在p1两次访问x之间
11：p2访问x，然后p1将x设为100，然后p2设置x
100：p1访问x(两次),p2将x设为11，然后p1设置x

用串行控制器解决并发问题

● 自己写一个 **make-serializer**，返回一个闭包(串行化组)。该闭包参数为过程**A**，返回值也是个过程**B**，过程**A**和**B**有相同的功能。同一闭包被调用返回的过程，都属于同一个集合。该集合称为“串行化集合”。

●串行化集合：

在每个时刻，任何一个串行化集合中，至多只有一个过程在执行。如果某个集合中一个过程**A**正在执行，二另一进程企图执行这个集合中的任何过程，则该进程都必须等到**A**执行结束后才能继续执行。

```
(define x 10)
(define s (make-serializer))   ;写法见后
(parallel-execute (s (lambda () (set! x (* x x))))
                  (s (lambda () (set! x (+ x 1)))))
```

两个**lambda**过程属于同一个串行化集合，不会出现一个过程执行到一半，另一个过程就投入执行的情况。

用串行控制器解决并发问题

- 串行化的 **make-account**

```
(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)
  (let ((protected (make-serializer)))
    (define (dispatch m)
      (cond ((eq? m 'withdraw) (protected withdraw))
            ((eq? m 'deposit) (protected deposit))
            ((eq? m 'balance) balance)
            (else (error "Unknown request -- MAKE-ACCOUNT"
                         m))))
    dispatch))
```

用串行控制器解决并发问题

- 串行化的 **make-account**

```
(define peter (make-account 100))
(define paul peter) ;如果 (define paul (make-account 100)) 会怎么样?
(define (p1) ((peter 'withdraw) 90))
(define (p2) ((paul 'withdraw) 30))
(parallel-execute p1 p2)
```

上面程序不会出问题了!

用串行控制器解决并发问题

- 串行化的 **make-account**

```
(define peter (make-account 100))
(define paul (make-account 100))

(define (p1) ((peter 'withdraw) 90))
(define (p2) ((paul 'withdraw) 30))
(parallel-execute p1 p2)
```

peter的串行化withdraw和 paul的串行化withdraw不会位于同一个串行集合中，可以并行！
因为它们不是同一闭包被调用返回的过程，因此不在同一个串行化集合内！

多重共享资源的复杂性

```
(define (exchange account1 account2)   ;交换两个账号的余额
  (let ((difference (- (account1 'balance)
                       (account2 'balance))))
    ((account1 'withdraw) difference)
    ((account2 'deposit) difference)))


(define (p1) (exchange a1 a2))
(define (p2) (exchange a2 a3))

(parallel-execute p1 p2)
```

p1算出a1,a2差额，然后p2在p1完成交换之前改变了a2的余额－> e r r o r !

多重共享资源的复杂性

```
(define (make-account-and-serializer balance)   ;改进 make-account
  (define (withdraw amount)
    (if (>= balance amount)
        (begin (set! balance (- balance amount))
               balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (let ((balance-serializer (make-serializer)))
    (define (dispatch m)
      (cond ((eq? m 'withdraw) withdraw)
            ((eq? m 'deposit) deposit)
            ((eq? m 'balance) balance)
            ((eq? m 'serializer) balance-serializer)
            (else (error "Unknown request -- MAKE-ACCOUNT"
                         m))))
    dispatch))
```

多重共享资源的复杂性

```
; 增加 serialized-exchange
(define (serialized-exchange account1 account2)
  (let ((serializer1 (account1 'serializer))
        (serializer2 (account2 'serializer)))
    ((serializer1 (serializer2 exchange))
     account1
     account2)))

(define a1 (make-account-and-serializer 100))
(define a2 (make-account-and-serializer 100))
(define a3 (make-account-and-serializer 100))
(define (p1) (serialized-exchange a1 a2))
(define (p2) (serialized-exchange a2 a3))
(parallel-execute p1 p2)      ;无忧!
```

多重共享资源的复杂性

```
; 增加 serialized-exchange
(define (serialized-exchange account1 account2)
  (let ((serializer1 (account1 'serializer))
        (serializer2 (account2 'serializer)))
    ((serializer1 (serializer2 exchange))
     account1
     account2)))

(define (p1) (serialized-exchange a1 a2))
(define (p2) (serialized-exchange a2 a3))
(parallel-execute p1 p2)      ;无忧!
```

p1中要执行的是 :

```
a1.balance-serializer(a2.balance-serializer(exchange)) //c++写法
```

p2中要执行的是:

```
a2.balance-serializer(a3.balance-serializer(exchange))
```

多重共享资源的复杂性

p1中要执行的是 ：

```
a1.balance-serializer(a2.balance-serializer(exchange))  //c++写法
```

p2中要执行的是：

```
a2.balance-serializer(a3.balance-serializer(exchange))
```

```
a2.balance-serializer(exchange)和
a2.balance-serializer(a3.balance-serializer(exchange))
```

在同一个串行集合中，因此必然是一个执行完了才执行另一个

串行化的实现

```
(define (make-serializer) ;生成一个串行化组
  (let ((mutex (make-mutex)))
    (lambda (p)
      (define (serialized-p . args)
        (mutex 'acquire)
        (let ((val (apply p args)))
          (mutex 'release)
          val))
      serialized-p)))
```

引入“互斥量”的概念。每个串行化组内部有一个“互斥量”。对于作为参数的过程p，串行化组返回一个过程q,q先获取“互斥量”，然后调用 p，然后释放“互斥量”。如果无法获取互斥量，则循环等待，直到获取为止。互斥量一旦被一个进程获取，其他进程则无法获取。

串行化的实现

```
(define (clear! cell)
  (set-car! cell false))
(define (make-mutex)
  (let ((cell (list false)))
    (define (the-mutex m)
      (cond ((eq? m 'acquire)
             (if (test-and-set! cell);值为true表示cell被占用
                 (the-mutex 'acquire))) ; retry
            ((eq? m 'release) (clear! cell))))
    the-mutex))

(define (test-and-set! cell)
  (if (car cell)
      true
      (begin (set-car! cell true)
             false)))
```

串行化的实现

```
(define (test-and-set! cell)
  (if (car cell)
      true
      (begin (set-car! cell true)
             false)))
```

test-and-set!操作必须以原子操作方式执行！(需操作系统提供API支持，如P操作)

1)方法一：

单CPU系统中，在test-and-set!执行过程中，不允许操作系统中止其执行，将时间片分给其他进程,如MIT scheme:

```
(define (test-and-set! cell)
  (without-interrupts ;without-interrupts是基本过程，在其参数执行过程中禁止时间片中断
    (lambda ()
      (if (car cell)
          true
          (begin (set-car! cell true)
                 false)))))
```

串行化的实现

```
(define (test-and-set! cell)
  (if (car cell)
      true
      (begin (set-car! cell true)
             false)))
```

test-and-set!操作必须以原子操作方式执行！(需操作系统提供API支持，如P操作)

1）方法二：

从CUP的指令上即支持原子操作，比如 “检查与设置”指令，“检查与清除”指令等。

串行化的实现

```
(define (clear! cell)
  (set-car! cell false))
(define (make-mutex)
  (let ((cell (list false)))
    (define (the-mutex m)
      (cond ((eq? m 'acquire)
             (if (test-and-set! cell);值为true表示cell被占用
                 (the-mutex 'acquire))) ; retry
            ((eq? m 'release) (clear! cell))))
    the-mutex))

(define (test-and-set! cell)
  (if (car cell)
      true
      (begin (set-car! cell true)
             false)))
```

test-and-set!操作必须以原子操作方式执行！(需操作系统提供API支持，如P操作)

死锁

```
(define (p1) (serialized-exchange a1 a2))
(define (p2) (serialized-exchange a2 a1))
(parallel-execute p1 p2)
```

p1中要执行的是 :

a1.balance-serializer(a2.balance-serializer(exchange))

- 等待 a1.balance-serializer 中的 mutex
  - 等待 a2.balance-serializer 中的mutex
    - 执行exchange
  - 释放a2.balance-serializer 中的mutext
- 释放a1.balance-serializer 中的mutext

p2中要执行的是:

a2.balance-serializer(a1.balance-serializer(exchange))

- 等待 a2.balance-serializer 中的 mutex
  - 等待 a1.balance-serializer 中的mutex
    - 执行exchange
  - 释放a1.balance-serializer 中的mutex
- 释放a2.balance-serializer 中的mutex

p1 p2同时抵达箭头处，则都无法往下进行，产生死锁

死锁

- 两个以上的进程，由于相互等待其他方释放资源而无法继续前进，这种情况称为死锁
- 如果并发系统里使用到多重资源，那么总存在出现死锁的可能
- 死锁是并发系统的一种不可容忍的，但也很常见的动态错误
- 常遇到计算机系统的许多死机现象，可能就是死锁

死锁的解决

- 在本问题中避免死锁的一种技术是给每个账户一个唯一标识号，修改**serialized-exchange**，让每个调用进程都先获取编号较小的账户的控制权，再获取编号较大的账户的控制权。相当于给账户一种排序，要求使用者按同样顺序操作。

- 死锁问题的情况很多，控制死锁的方法需要针对问题设计
- 一些死锁情况可能需要更复杂的技术
- 不存在对任何情况都有效的死锁控制技术

并发，时间和通信

实际上，在并发系统中，什么是**“共享状态”的问题有时也不清楚：究竟**哪些东西应该看着是共享的资源？

例如，常见的**test-and-set! 等机制要求进程能在任意时刻检查一个全**局共享标志，以便控制不同进程的执行情况

但是，新型高速处理器采用了许多优化技术如（多级）流水线和缓存，弱内存模型等简单的串行化技术越来越不适用（对效率影响太大）

分布式系统里的共享变量可能出现在不同分布式站点的存储器里如何保证整个系统的存储一致性变成了一个大问题

并发，时间和通信

以分布式银行系统为例

一个账户可能在多个分行有当地版本。同版本存放物理上分布于不同地点的不同计算机的存储器里，但都反应着同一个账户的信息，应该相互一致

由于物理分布**/网络延迟/操作代价等，不可能每时每刻都一致**

不同版本需要定期或不定期地同步

这种情况下，某时刻一个账户的余额是多少？存在不确定性

假如**Peter 和Paul 分别向同一账户存款**

账户在某个时刻有多少钱是不清楚的（是存入时，还是下次全系统同步时？**…**）

在不断存取钱的过程中，账户的余额也具有非确定性

状态、并发和共享信息的问题总是与时间密切相关的，而且时很难控制的。函数式编程中不存在时间，有优越性