In structuring computational models to match our perception of the physical world, we came up with Object oriented programming (OOP).

Building models in terms of computational objects with local state forces us to confront *time* as an essential concept in programming. Objects in the world do not change one at a time in sequence. In several cases, these objects can be acting concurrently—all at once.

It is often natural to model systems as collections of computational processes that execute concurrently.

Concurrency helps make modular programs

Even if the programs are to be executed on a sequential computer, the practice of writing programs as if they were to be executed concurrently forces the programmer to avoid inessential timing constraints and thus makes programs more modular.

For example:-

```
> (define x 1)
> (define y 2)
> (+ x y)
3
```

Here, even though x is defined before y, it would not have made any difference if we had interchanged the order in which they were defined.

Concurrent computation can provide a speed advantage over sequential computation.

Sequential computers execute only one operation at a time. Therefore, the time taken to perform a task is $\propto$ to the total number of operations performed.

Concurrent computation can provide a speed advantage over sequential computation.

Sequential computers execute only one operation at a time. Therefore, the time taken to perform a task is $\propto$ to the total number of operations performed.

If it is possible to decompose a problem into pieces that are

- relatively independent and
- need to communicate only rarely,

it may be possible to allocate pieces to separate computing processors, producing a speed advantage proportional to the number of processors available.

Concurrent computation can provide a speed advantage over sequential computation.

Sequential computers execute only one operation at a time. Therefore, the time taken to perform a task is $\propto$ to the total number of operations performed.

If it is possible to decompose a problem into pieces that are

- relatively independent and
- need to communicate only rarely,

it may be possible to allocate pieces to separate computing processors, producing a speed advantage proportional to the number of processors available.

Unfortunately, the complexities introduced by assignment become even more problematic in the presence of concurrency.

Time can be seen an ordering imposed on events.

For any events A and B,
either A occurs before B,
A and B are simultaneous, or
A occurs after B.

However, things get complicated when events A and B fall in neither of the aforementioned possibilities.

```
(set! balance (- balance amount))
```

One single set instruction can cause a lot of chaos if not dealt with care.

Suppose that Peter withdraws \$10 and Paul withdraws \$25 from a joint account that initially contains \$100, leaving \$65 in the account.

Depending on the order of the two withdrawals, the sequence of balances in the account is either
\$100 $\rightarrow$ \$90 $\rightarrow$ \$65 or
\$100 $\rightarrow$ \$75 $\rightarrow$ \$65.

This view is incomplete and can be become a problem.

Suppose that Peter withdraws \$10 and Paul withdraws \$25 from a joint account that initially contains \$100, leaving \$65 in the account.

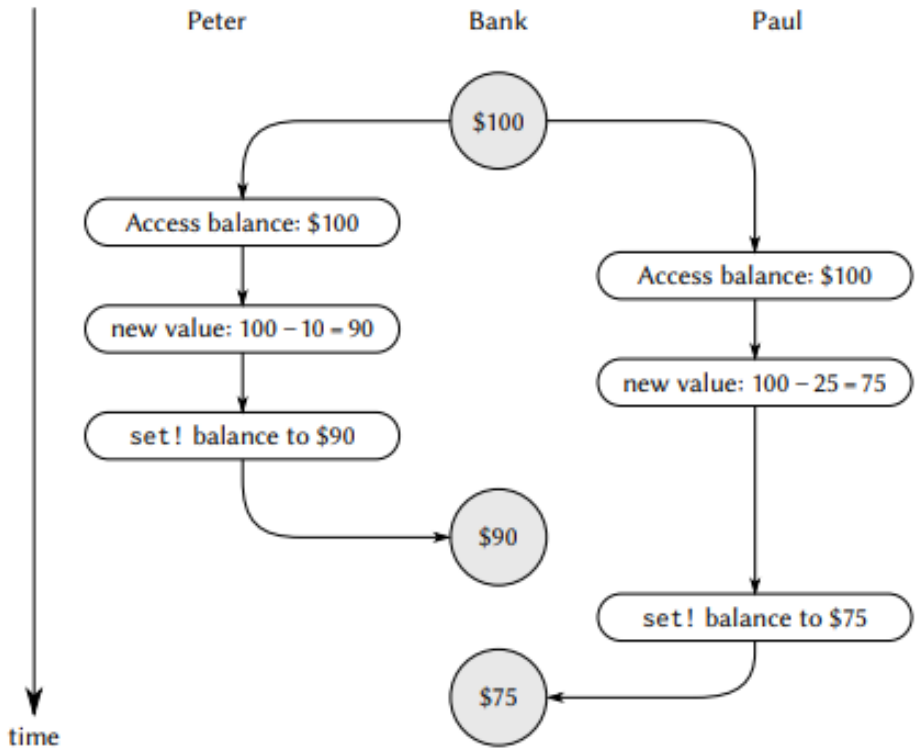
Depending on the order of the two withdrawals, the sequence of balances in the account is either
\$100 $\rightarrow$ \$90 $\rightarrow$ \$65 or
\$100 $\rightarrow$ \$75 $\rightarrow$ \$65.

This view is incomplete and can be become a problem.

The set! balance instruction consists of three steps:

1. accessing the value of the balance variable
2. computing the new balance
3. setting balance to this new value

If Peter and Paul's withdrawals execute this statement concurrently, then the two withdrawals might interleave the order in which they access balance and set it to the new value.



At the end of the two transactions, the correct value should have been $65 instead of $75.

The general phenomenon captured by this example is that several processes may share a common state variable and more than one process may be trying to manipulate the shared state at the same time.

The general phenomenon captured by this example is that several processes may share a common state variable and more than one process may be trying to manipulate the shared state at the same time.

For the bank account example, during each transaction, each customer should be able to act as if the other customers did not exist.

When a customer changes the balance in a way that depends on the balance, he must be able to assume that, just before the moment of change, the balance is still what he thought it was.

If the new value of balance did not depend on the old value of balance, they would not cause such inconsistencies.
For example

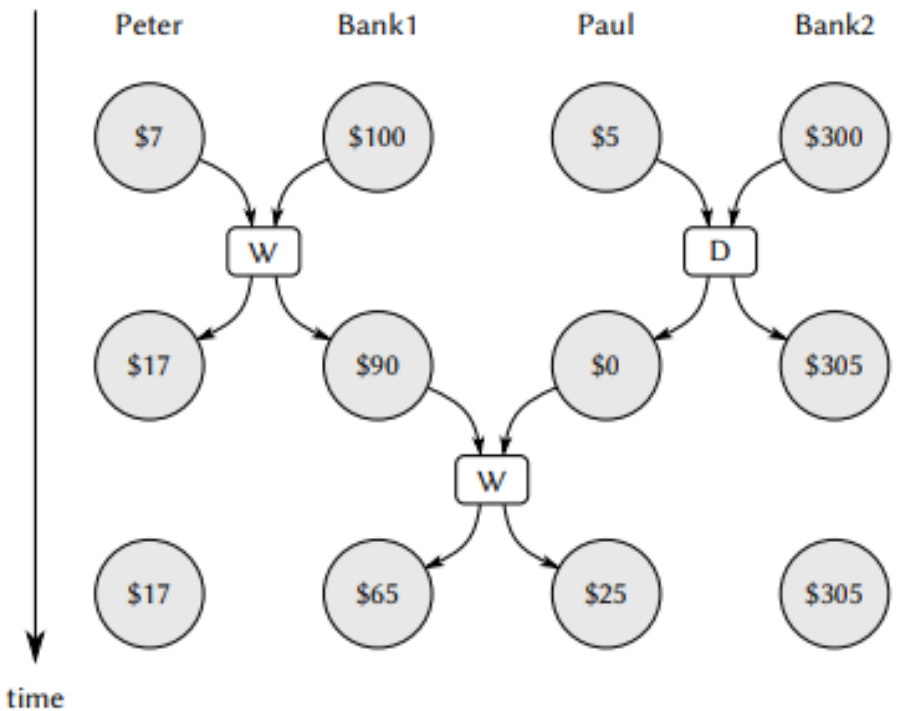
set! balance $20
followed by
set! balance $15

If two instructions occur in the same order whether they interleave or not, would produce the same result $15, because the updated value of balance does not depend on the pre-existing value of balance.

To make concurrent programs behave correctly, we may have to place some restrictions on concurrent execution.
Restriction:- **No two operations that change any shared state variables can occur at the same time.**

To make concurrent programs behave correctly, we may have to place some restrictions on concurrent execution.
Restriction:- **No two operations that change any shared state variables can occur at the same time.**

This would be both inefficient and overly conservative.

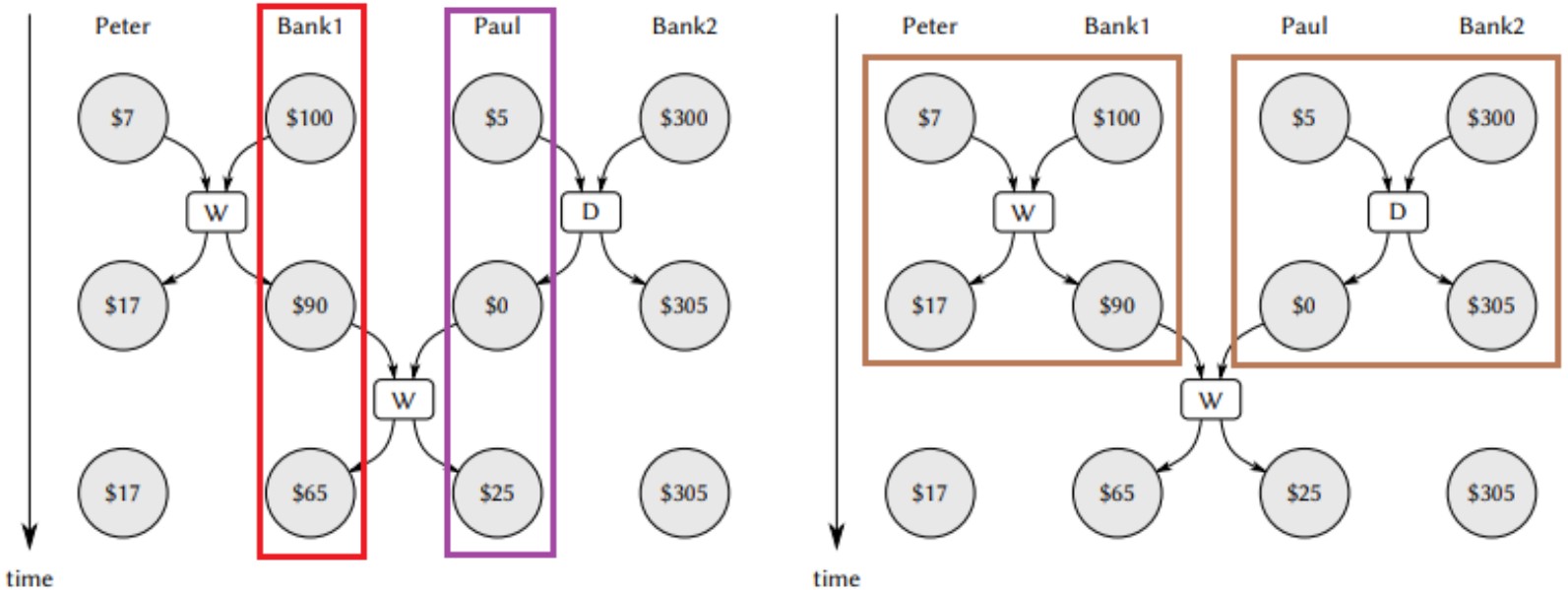


What cannot be concurrent

- The two withdrawals from the joint account.
- Paul's deposit and withdrawal.

What can be concurrent or sequential

- Paul's deposit in Bank2 and Peter's withdrawal from Bank1



What should be concurrent

- Deducting $7 from Bank1 to add $7 to Peter's wallet.

In this case, the changes are happening for two different state variables and even then one cannot serialize them.

Otherwise, the invariant that the total cash in the system remaining the same at every instance in time would be invalidated.

A less stringent restriction on concurrency would ensure that the result that a concurrent system produces is same as the result if the processes had run sequentially in *some* order.

A less stringent restriction on concurrency would ensure that the result that a concurrent system produces is same as the result if the processes had run sequentially in *some* order.

Consequences

1. It does not require the processes to actually run sequentially, but only to produce results that are the same as if they had run sequentially.
   $\rightarrow$ For example:- Paul's action of withdrawing \$10 from Bank1 and Peter's action of depositing \$5 in Bank2 would have the same effect, if they had been executed sequentially or concurrently.

2. There may be more than one possible "correct/acceptable" result produced by a concurrent program, because we require only that the result be the same as for *some* sequential order.

   $\rightarrow$ For example, suppose that Peter and Paul's joint account starts out with \$100, and Peter deposits \$40 while Paul concurrently withdraws half the money in the account.
   Then sequential execution could result in the account balance being either \$70 or \$90.

This less stringent restriction does resolve the inter-leaving problem.

A practical approach to the design of concurrent systems is to devise general mechanisms to constrain the interleaving of concurrent processes so that we can be sure that the program behavior is correct.
One such mechanism is called the *serializer*.

A practical approach to the design of concurrent systems is to devise general mechanisms to constrain the interleaving of concurrent processes so that we can be sure that the program behavior is correct.
One such mechanism is called the *serializer*.

Serialization creates distinguished sets of procedures such that only one execution of a procedure in each serialized set is permitted to happen at a time.

If some procedure in the set is being executed, then a process that attempts to execute any procedure in the set will be forced to wait until the first execution has finished.

```
(require (planet dyoo/sicp-concurrency:1:2/sicp-concurrency))

(define x 10)

(parallel-execute
 (lambda () (set! x (+ x 15))) ; Procedure #1
 (lambda () (set! x (* x x)))  ; Procedure #2
)

(display "After the block is executed \n")
(display "The final value of x is ")
(display x)
```

The first line provides the required libraries for the remainder of the course.
Assume that we have extended Scheme to include a procedure called parallel-execute:

$$(\text{parallel-execute} < p_1 >< p_2 > \ldots < p_k >)$$

Each $< p >$ must be a procedure of no arguments. parallel-execute creates a separate process for each $< p >$.

Description of the output

```
#|

Everytime we execute this code, we can get a different answer,
based on order in which the internal procedure executes.

The final value of x can be

625  :- (10+15)^2      (#1 -> #2)
115  :- 10^2+15        (#2 -> #1)
100  :- 10^2           (Procedure #1, #2 simultaneously access x and
                       Procedure #2 makes the final change to x)
25   :- 10+15         (Procedure #1, #2 simultaneously access x and
                       Procedure #1 makes the final change to x)
250  :- 10*15         (Procedure #1,#2 simultaneously access x and

                       value of x changes between the two consecutive
                       access's to x made by Procedure 2)

|#
```

Outputs 100, 25 and 250 occur due to inter-leaving. Inter-leaving can create an illusion that one or more of the procedures did not execute. First of all, we would like to avoid the inter-leave cases by using serialization.

Serialization implements the following idea:
Certain collections of procedures are forced to not be executed concurrently.

Serialization creates distinguished sets of procedures such that only one execution of a procedure in each serialized set is permitted to happen at a time.

Serialization implements the following idea:
Certain collections of procedures are forced to not be executed concurrently.

Serialization creates distinguished sets of procedures such that only one execution of a procedure in each serialized set is permitted to happen at a time.

For example,
set-1 has procedures $< p_1 >, < p_2 > \ldots < p_m >$ and
set-2 has procedures $< q_1 >, < q_2 > \ldots < q_n >$

then p_i and p_j cannot be allowed to execute at the same time
q_i and q_j cannot be allowed to execute at the same time.

However, p_i and q_j can be allowed to be executed concurrently.

To reduce the possible outcomes arising by concurrency, we can make use of **serializers**.

Serializers are constructed by make-serializer in scheme.
A serializer takes a procedure as argument and returns a serialized procedure that behaves like the original procedure.

All calls to a given serializer return serialized procedures in the same set.

```
(define x 10)

(define (parallel-execute . procs)
  (map thread-wait
       (map (lambda (proc) (thread proc))
            procs)))

(define s (make-serializer))

(parallel-execute
 (s (lambda () (set! x (+ x 15)))) ; Procedure #1
 (s (lambda () (set! x (* x x))))  ; Procedure #2
)

(display x)

#|
115 and 625 are both acceptable and possible outcomes,
whereas
100, 25 and 250 are possibilities that have been avoided.
|#
```

Using serializers is relatively straightforward when there is only a single shared resource (such as a single bank account).

However, concurrent programming can be very difficult when there are multiple shared resources.

We can implement serializers in terms of a more primitive synchronization mechanism called a **mutex**.
A mutex is like a baton in a relay race. In a relay race, only the constestant holding the baton can run.

We can implement serializers in terms of a more primitive synchronization mechanism called a **mutex**.
A mutex is like a baton in a relay race. In a relay race, only the constestant holding the baton can run.

Similarly, a mutex is an object that supports two operations—

1. the mutex can be acquired, and
2. the mutex can be released.

We can implement serializers in terms of a more primitive synchronization mechanism called a **mutex**.
A mutex is like a baton in a relay race. In a relay race, only the constestant holding the baton can run.

Similarly, a mutex is an object that supports two operations—

1. the mutex can be acquired, and
2. the mutex can be released.

Once a mutex has been acquired, no other acquire operations on that mutex may proceed until the mutex is released.

Each serializer has an associated mutex.

Given a procedure $< p >$, the serializer

1. returns a procedure that acquires the mutex,
2. runs $< p >$, and then
3. releases the mutex.

The mutex is a mutable object (say a one-element list, referred to as a cell) that can hold the value true or false.

cell's value $\rightarrow$ false $\Longrightarrow$ the mutex is available to be acquired.

cell's value $\rightarrow$ true $\Longrightarrow$ the mutex is unavailable, and any process that attempts to acquire the mutex must wait.

There is a crucial subtlety here, which is the essential place where concurrency control enters the system:

The mutex is a mutable object (say a one-element list, referred to as a cell) that can hold the value true or false.

cell's value $\rightarrow$ false $\Longrightarrow$ the mutex is available to be acquired.

cell's value $\rightarrow$ true $\Longrightarrow$ the mutex is unavailable, and any process that attempts to acquire the mutex must wait.

There is a crucial subtlety here, which is the essential place where concurrency control enters the system:

We must guarantee that, once a process has tested the cell and found it to be false, the cell contents will actually be set to true before any other process can test the cell.

Imagine that Peter attempts to exchange a_1 with a_2 while Paul concurrently attempts to exchange a_2 with a_1.

Suppose that Peter's process reaches the point where it has entered a serialized procedure protecting a_1 and, just after that, Paul's process enters a serialized procedure protecting a_2.

Imagine that Peter attempts to exchange a_1 with a_2 while Paul concurrently attempts to exchange a_2 with a_1.

Suppose that Peter's process reaches the point where it has entered a serialized procedure protecting a_1 and, just after that, Paul's process enters a serialized procedure protecting a_2.

Now Peter cannot proceed (to enter a serialized procedure protecting a_2) until Paul exits the serialized procedure protecting a_2. Similarly, Paul cannot proceed until Peter exits the serialized procedure protecting a_1.

Each process is stalled forever, waiting for the other.

This situation is called a **deadlock**.

Deadlock is always a danger in systems that provide concurrent access to multiple shared resources.

There are other situations that require sophisticated deadlock-avoidance techniques,
and sometimes deadlock cannot be avoided at all.