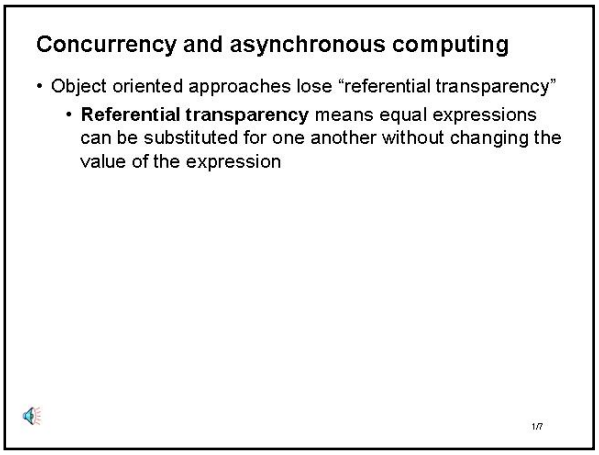
6.001 Notes: Section 32.1

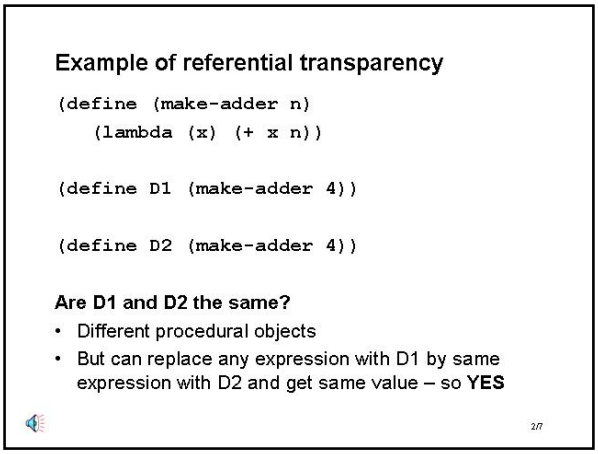
Slide 32.1.1



We have spent some time looking at computational objects that are built around the use of local state variables, and methods for changing those variables to reflect evolution in the state of the object. In particular, we saw an example of building a system around an object oriented paradigm, in which the central component of our system was a set of communicating objects, that took messages are arguments, and returned methods that could be applied to objects and other arguments to simulate interactions in complex systems.

We saw a hint of the power of orienting system design around such principles, but we also saw that this power of using local state to model systems also extracts a price: the loss of **referential transparency**.

So what does this mean? A language with referential transparency means that equal expressions can be substituted for one another without changing the value of the expression.



Slide 32.1.2

For example consider the code shown on the slide. `Make-adder` creates a procedure that adds a fixed number to its argument. We can use it to create two adders, as shown, called `D1` and `D2`.

The question in which we are interested is whether `D1` and `D2` are the same? We would argue that in one sense the answer is no, since they point to different procedural structures, but in another sense the answer is yes, since we can replace any expression involving `D1` with an equivalent expression involving `D2` and we will get exactly the same behavior.

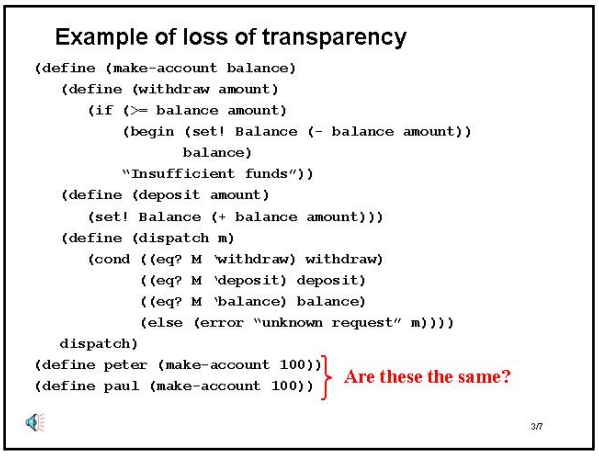
Slide 32.1.3

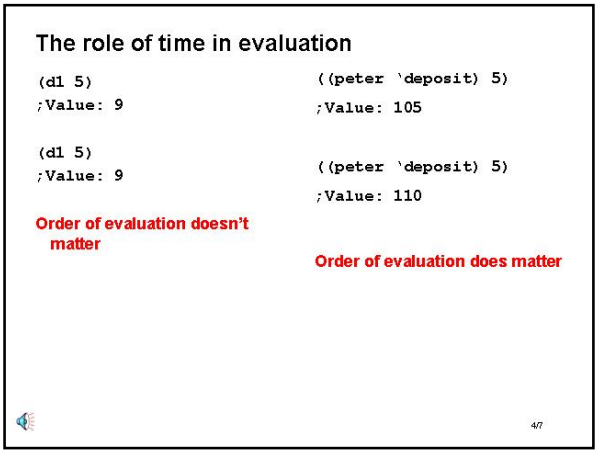
But now consider the code shown on this slide.

Here we have a simple message passing procedure for representing bank accounts.

We can again ask whether `peter` and `paul` are the same. Here, we know intuitively that the answer is no. Even though the expression used to create each is the same, we know that the behavior of these objects is different, because of the local state.

In this case, we do not have referential transparency, since the same expression does not give rise to things that can be substituted for one another.



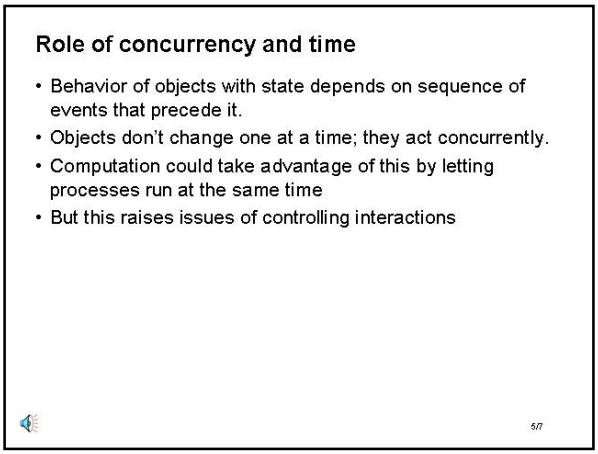


Slide 32.1.4

The apparently simple introduction of local state and mutation into our language thus has some drastic consequences: it raises questions about sameness and change.

The central issue lurking beneath the complexity of state, sameness, and change is that by introducing assignment we are forced to admit **time** into our computational models. Before we introduced assignment, all our programs were timeless, in the sense that any expression that has a value always has the same value. Thus, calling `(d1 5)` would always return the same value. In contrast, look at our modeling of deposits to a bank account, that returns the resulting balance. Here successive evaluations of the same expression yield different values. This behavior arises from the fact that the execution of assignment statements (in this case, assignments to the variable `balance`) delineates **moments in time** when values change. The result of evaluating an expression depends not only on the expression itself, but also on whether the evaluation occurs before or after these moments. Building models in terms of computational objects with local state forces us to confront time as an essential concept in programming.

Slide 32.1.5

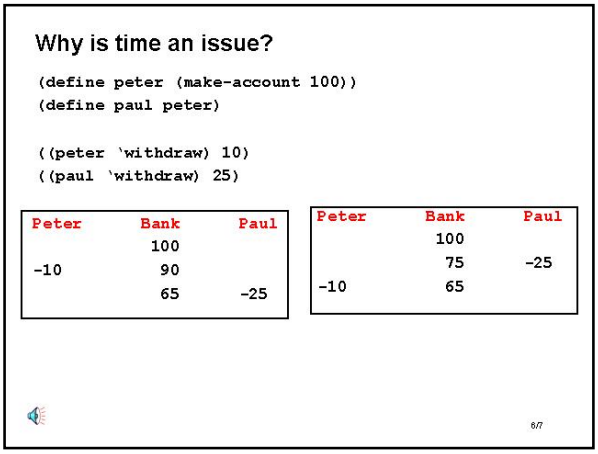


We can go further in structuring computational models to match our perception of the physical world. Objects in the world do not change one at a time in sequence. Rather we perceive them as acting **concurrently**---all at once. So it is often natural to model systems as collections of computational processes that execute concurrently. Just as we can make our programs modular by organizing models in terms of objects with separate local state, it is often appropriate to divide computational models into parts that evolve separately and concurrently. 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.

In addition to making programs more modular, concurrent computation can provide a speed advantage over sequential computation. Sequential computers execute only one operation at a time, so the amount of time it takes to perform a task is proportional to the total number of operations performed. However, 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. The fact of concurrent execution, either because the world operates in parallel or because our computers do, entails additional complexity in our understanding of time.

Today, we are going to look at those issues and ways to try to get around them.

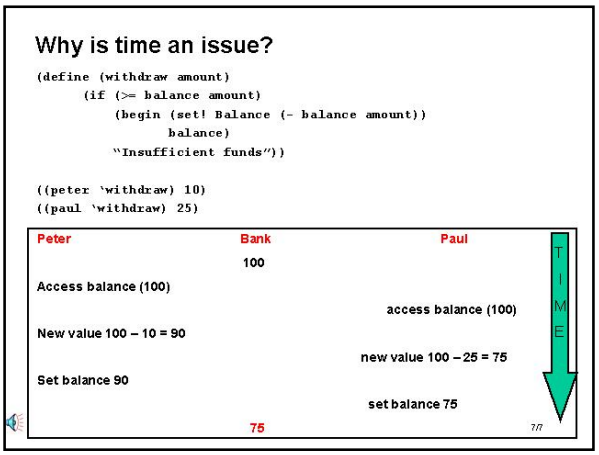


Slide 32.1.6

Now why should time be an issue? For any two events, A and B, either A occurs before B, A and B are simultaneous, or A occurs after B.

But let's look at that carefully. Suppose we create an account for Peter, that Paul shares. Now suppose that that Peter withdraws 10 dollars and Paul withdraws 25 dollars from this joint account that initially contains 100 dollars, leaving 65 in the account. Depending on the order of the two withdrawals, the sequence of balances in the account is either 100, then 90 then 65, or 100, then 75, then 65. In a computer implementation of the banking system, this changing sequence of balances could be modeled by successive assignments to the variable `balance`.

Slide 32.1.7



In complex situations, however, such a view can be problematic. Suppose that Peter and Paul, and many other people besides, are accessing the same bank account through a network of banking machines distributed all over the world. The actual sequence of balances in the account will depend critically on the detailed timing of the accesses and the details of the communication among the machines. The size of the event matters in determining whether the outcome occurs correctly.

This indeterminacy in the order of events can pose serious problems in the design of concurrent systems. For instance, suppose that the withdrawals made by Peter and Paul are implemented as two separate processes sharing a common variable `balance`, each process specified by the `withdraw` procedure:

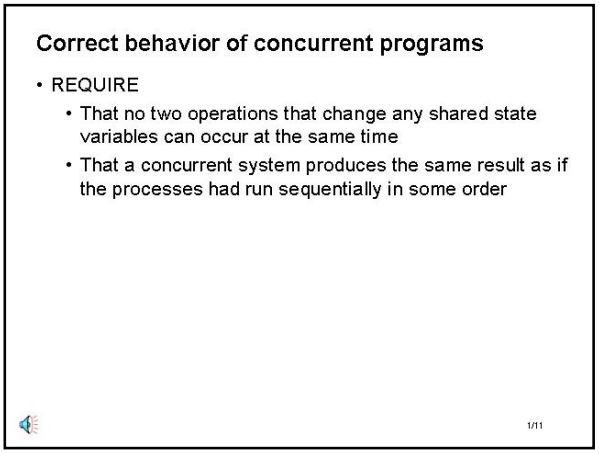
Consider the expression `(set! balance (- balance amount))` executed as part of each withdrawal process. This consists of three steps: (1) accessing the value of the `balance` variable; (2) computing the new balance; (3) setting `balance` to this new value. If Peter and Paul's withdrawals execute this statement concurrently, then the two withdrawals might interleave the order in which they access `balance` and set it to the new value.

The timing diagram depicts an order of events where `balance` starts at 100, Peter withdraws 10, Paul withdraws 25, and yet the final value of `balance` is 75. As shown in the diagram, the reason for this anomaly is that Paul's assignment of 75 to `balance` is made under the assumption that the value of `balance` to be decremented is 100. That assumption, however, became invalid when Peter changed `balance` to 90. This is a catastrophic failure for the banking system, because the total amount of money in the system is not conserved. Before the transactions, the total amount of money was 100. Afterwards, Peter has 10, Paul has 25, and the bank has 75.

The general phenomenon illustrated here is that **several processes may share a common state variable**. What makes this complicated is that 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.

6.001 Notes: Section 32.2

Slide 32.2.1

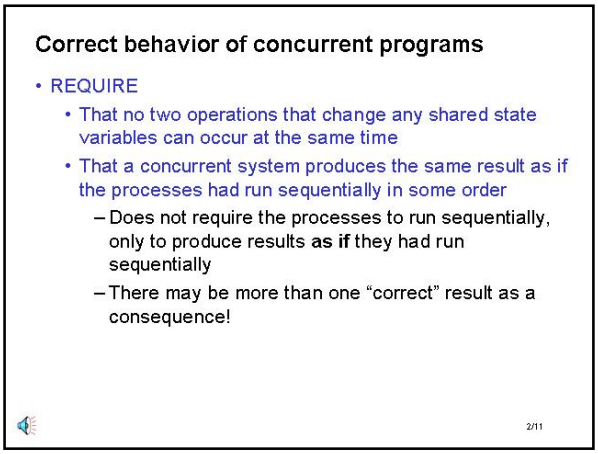


The above example typifies the subtle bugs that can creep into concurrent programs. The root of this complexity lies in the assignments to variables that are shared among the different processes. We already know that we must be careful in writing programs that use `set!`, because the results of a computation depend on the order in which the assignments occur.

With concurrent processes we must be especially careful about assignments, because we may not be able to control the order of the assignments made by the different processes. If several such changes might be made concurrently (as with two depositors accessing a joint account) we need some way to ensure that our system behaves correctly. For example, in the case of withdrawals from a joint bank account, we must ensure that money is conserved. To make concurrent programs behave correctly, we may have to place some restrictions on concurrent execution.

One possible restriction on concurrency would stipulate that **no two operations that change any shared state variables can occur at the same time**. This is an extremely stringent requirement. For distributed banking, it would require the system designer to ensure that only one transaction could proceed at a time. This would be both inefficient and overly conservative.

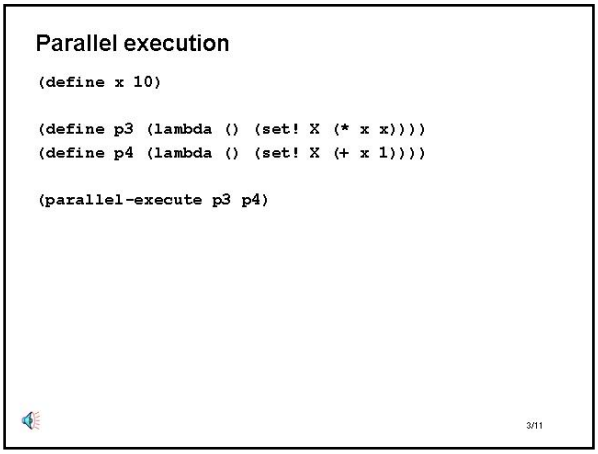
A less stringent restriction on concurrency would ensure **that a concurrent system produces the same result as if the processes had run sequentially in some order.**



Slide 32.2.2

There are two important aspects to this requirement. First, 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. For our previous example, the designer of the bank account system can safely allow Paul's deposit and Peter's withdrawal to happen concurrently, because the net result will be the same as if the two operations had happened sequentially. Second, there may be more than one possible correct result produced by a concurrent program, because we require only that the result be the same as for some sequential order.

Slide 32.2.3

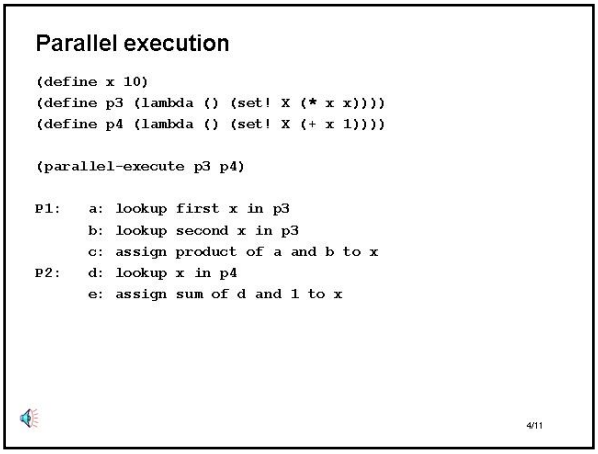


Let's look at this a bit more carefully.

To make the above mechanism more concrete, suppose that we have extended Scheme to include a procedure called `parallel-execute`: this procedure takes a set of procedures of no arguments as input. It then creates a separate process for each such procedure (think of this as a separate evaluator, with its own environment) and applies such procedure within that process. All these processes (or evaluators) then run concurrently.

As an example of how this is used, consider the example shown here.

This creates two concurrent processes: P1, which sets `x` to `x` times `x`, and P2, which increments `x`.



Slide 32.2.4

To understand what would be legitimate results under concurrent operation of P1 and P2, let's break down the stages a bit more finely.

P1 executes three different stages, as shown.

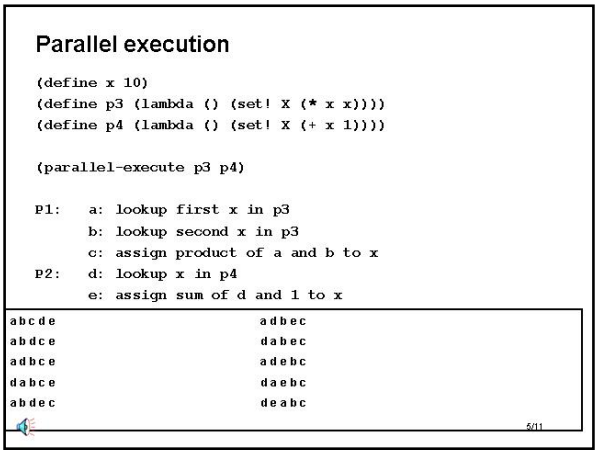
And P2 executes two different stages, as shown.

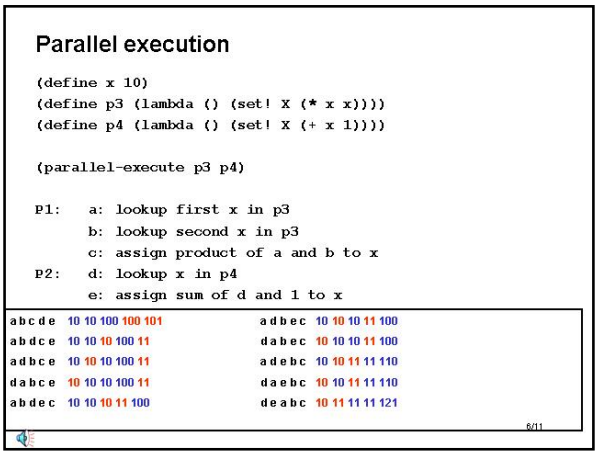
Slide 32.2.5

For these processes to operate correctly, P1 simply needs to ensure that the ordering `a, b, c` takes places, and P2 simply needs to ensure that the ordering `d, e` occurs.

Here are the different ways in which we can preserve these orderings, while allowing for intertwining of stages between the two processes.

In other words, each of these orderings is a legitimate sequence, since the result would reflect a correct sequence of operations from each process. All that differs is how the sequences intertwine.



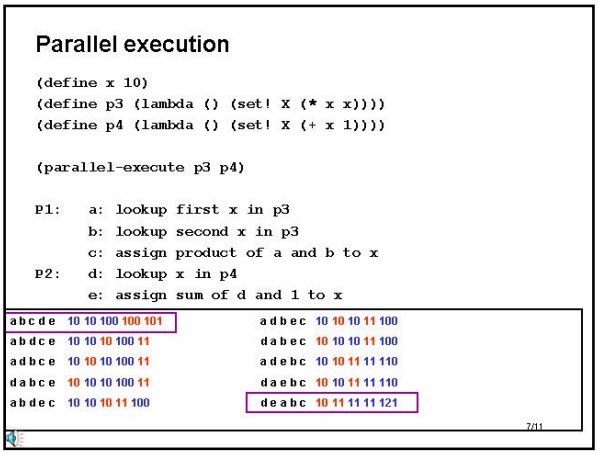


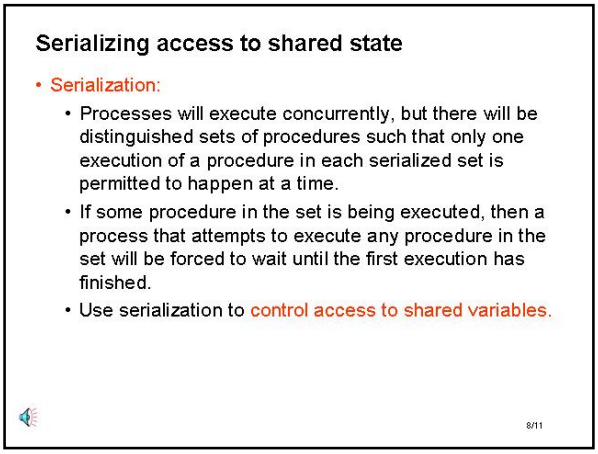
Slide 32.2.6

After execution is complete, `x` will be left with one of five possible values, depending on the interleaving of the events of P1 and P2. We can see this by marking the value of `x` at each stage as shown. The blue values come from P1, the red from P2. If we allow any intertwining of the stages of each process, we see that there are five different possible final values for `x`: 11, 100, 101, 110, and 121.

Slide 32.2.7

On the other hand, if we insist that only sequential orderings of the two processes occur, i.e., no intertwining of intermediate stages, then there are only two possible outcomes.





Slide 32.2.8

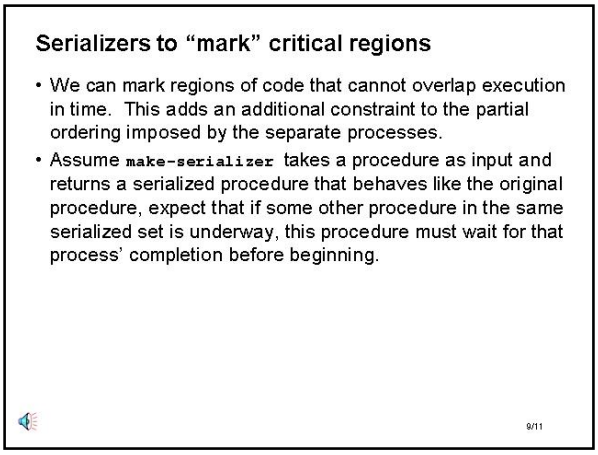
So we see that if we are to allow for concurrent processes to take place, we need some way of specifying units of computation that need to take place as a whole, before any other computation can start.

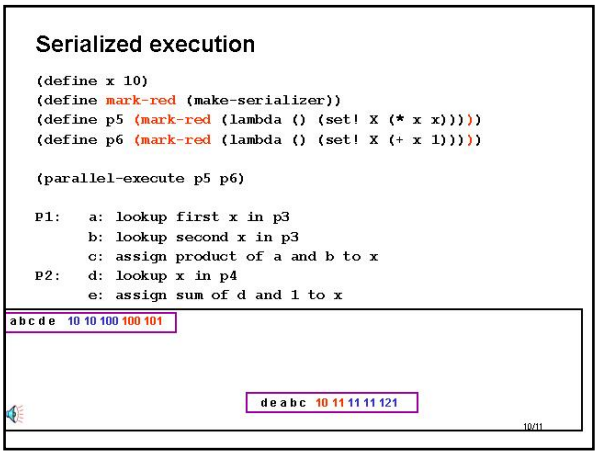
Serialization implements the following idea: Processes will execute concurrently, but there will be certain collections of procedures that cannot be executed concurrently. More precisely, serialization creates distinguished sets of procedures such that only one execution of a procedure in each serialized 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.

We can use **serialization to control access to shared variables.** For example, if we want to update a shared variable based on the previous value of that variable, we put the access to the previous value of the variable and the assignment of the new value to the variable in the same procedure. We then ensure that no other procedure that assigns to the variable can run concurrently with this procedure by serializing all of these procedures with the same serializer. This guarantees that the value of the variable cannot be changed between an access and the corresponding assignment.

Slide 32.2.9

We can constrain the concurrency by using serialized procedures. These are created by serializers, which are constructed by `make-serializer`, whose implementation we will get to shortly. 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. This means that a procedure may not begin execution if another procedure from the same set has not yet completed execution.





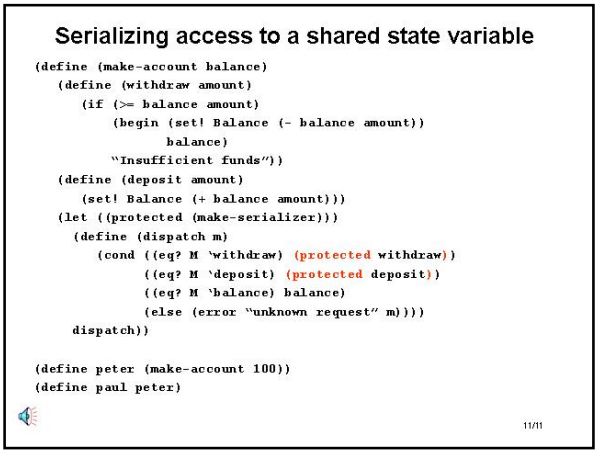
Slide 32.2.10

Thus, in contrast to the example in a previous slide, executing the code shown here can produce only two possible values for `x`, 101 or 121. The other possibilities are eliminated, because the execution of P1 and P2 cannot be interleaved.

Slide 32.2.11

So if we go back to our bank account example, here is an easy way to use this idea to fix the problem. Here we serialize both the deposits and withdrawals.

With this implementation, two processes cannot be withdrawing from or depositing into a single account concurrently. This eliminates the source of the error illustrated in our earlier bank account example, where Peter changes the account balance between the times when Paul accesses the balance to compute the new value and when Paul actually performs the assignment. On the other hand, each account has its own serializer, so that deposits and withdrawals for different accounts can proceed concurrently.

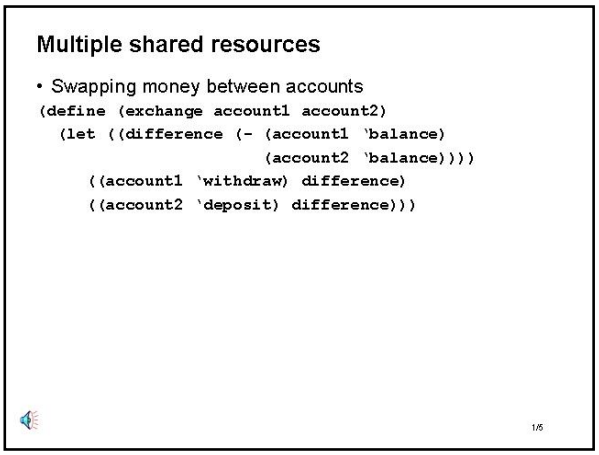


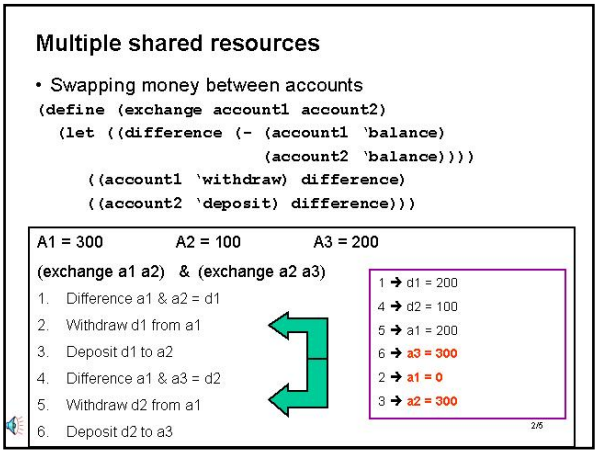
6.001 Notes: Section 32.3

Slide 32.3.1

Serializers provide a powerful abstraction that helps isolate the complexities of concurrent programs so that they can be dealt with carefully and (hopefully) correctly. However, while using serializers is relatively straightforward when there is only a single shared resource (such as a single bank account), concurrent programming can be treacherously difficult when there are multiple shared resources.

To see some of the difficulties that can arise, suppose we wish to swap the balances in two bank accounts. We access each account to find the balance, compute the difference between the balances, withdraw this difference from one account, and deposit it in the other account. We could implement this with the code shown on the slide.





Slide 32.3.2

This procedure works well when only a single process is trying to do the exchange. Suppose, however, that Peter and Paul both have access to accounts a1, a2, and a3, and that Peter exchanges a1 and a2 while Paul concurrently exchanges a1 and a3.

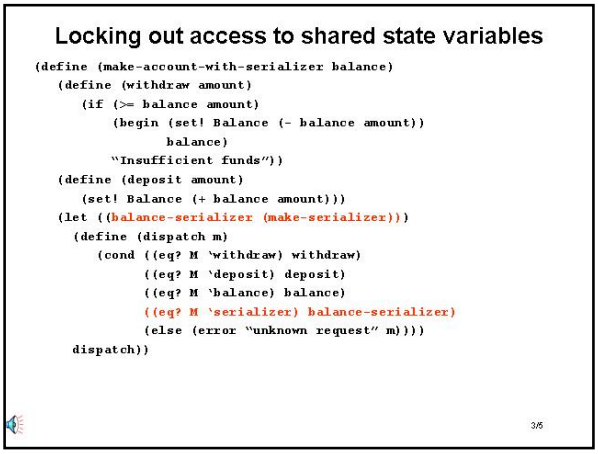
Even with account deposits and withdrawals serialized for individual accounts (as in the `make-account` procedure shown above), `exchange` can still produce incorrect results. For example, Peter might compute the difference in the balances for a1 and a2, but then Paul might change the balance in a1 before Peter is able to complete the exchange.

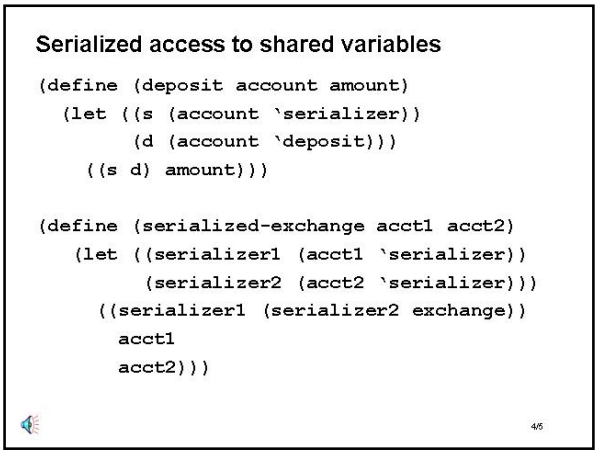
In more specific detail, the stages of the computation of the two exchanges are shown. We know by serialization that steps 2 and 5 must each complete in entirety before the other can begin. However, it is still possible for the sequence of steps shown on the right to take place, resulting in an incorrect behavior (the total amount of money is preserved but we intended to simply swap amounts between the accounts, meaning there should be 100, 200 and 300 dollars in the accounts).

Slide 32.3.3

For correct behavior, we must arrange for the `exchange` procedure to **lock out** any other concurrent accesses to the accounts during the entire time of the exchange.

One way we can accomplish this is by using both accounts' serializers to serialize the entire `exchange` procedure. To do this, we will arrange for access to an account's serializer. Note that we are deliberately breaking the modularity of the bank-account object by exposing the serializer. This version of `make-account` is identical to the original version, except that a serializer is provided to protect the balance variable, and the serializer is exported via message passing.



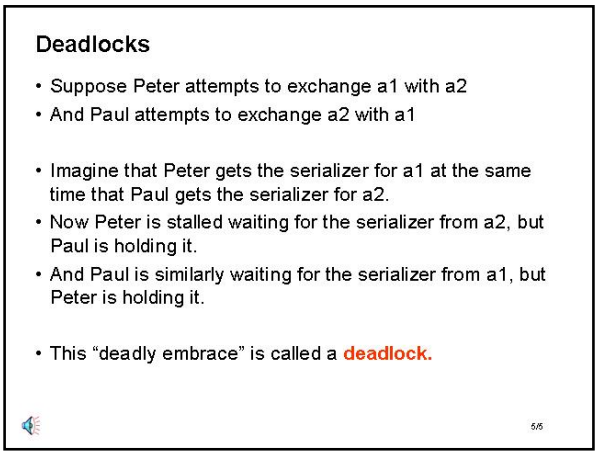


Slide 32.3.4

We can use this to do serialized deposits and withdrawals. However, unlike our earlier serialized account, it is now the responsibility of each user of bank-account objects to explicitly manage the serialization. For example, we might create a deposit procedure that gets access to the balance through the serializer, thus ensuring that it only gains access if no one else is using it. Exporting the serializer in this way gives us enough flexibility to implement a serialized exchange program. We simply serialize the original `exchange` procedure with the serializers for both accounts, as shown.

Slide 32.3.5

Even with the serialization technique discussed above, account exchanging still has a problem. Imagine that Peter attempts to exchange a1 with a2 while Paul concurrently attempts to exchange a2 with a1. Suppose that Peter's process reaches the point where it has entered a serialized procedure protecting a1 and, just after that, Paul's process enters a serialized procedure protecting a2. Now Peter cannot proceed (to enter a serialized procedure protecting a2) until Paul exits the serialized procedure protecting a2. Similarly, Paul cannot proceed until Peter exits the serialized procedure protecting a1. 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.

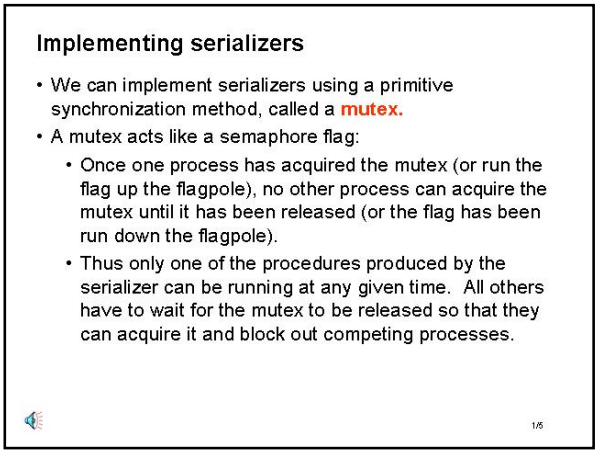


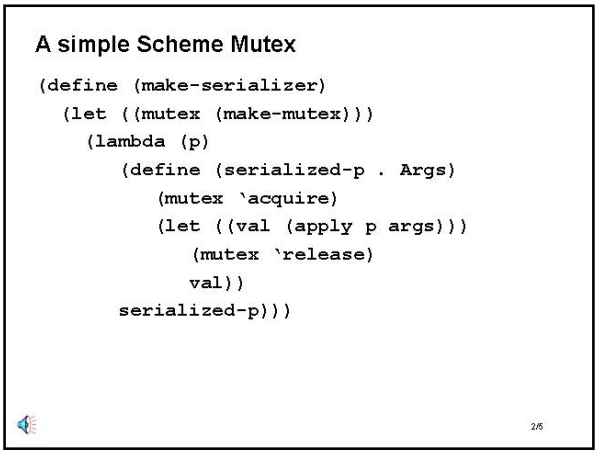
One way to avoid the deadlock in this situation is to give each account a unique identification number and rewrite `exchange` so that a process will always attempt to enter a procedure protecting the lowest-numbered account first. Although this method works well for the exchange problem, there are other situations that require more sophisticated deadlock-avoidance techniques, or where deadlock cannot be avoided at all.

6.001 Notes: Section 32.4

Slide 32.4.1

Finally, how do we actually implement serializers? The most common method is to implement serializers in terms of a more primitive synchronization mechanism called a **mutex**. A mutex is an object that supports two operations: the mutex can be acquired, and 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.



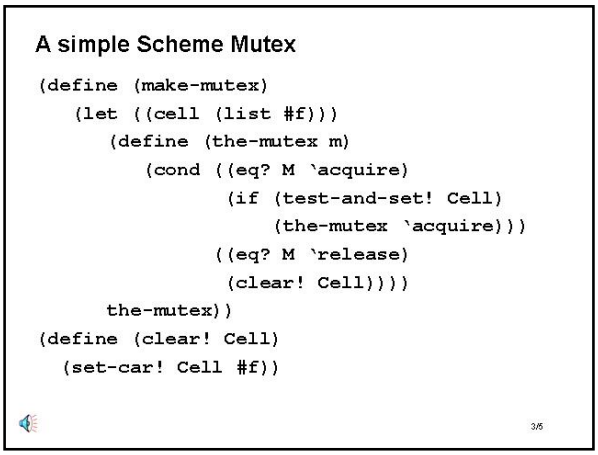


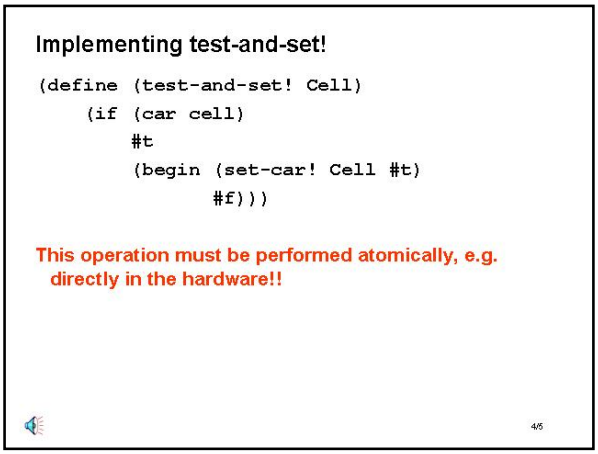
Slide 32.4.2

In our implementation, each serializer has an associated mutex. Given a procedure `p`, the serializer returns a procedure that acquires the mutex, runs `p`, and then releases the mutex. This ensures that only one of the procedures produced by the serializer can be running at once, which is precisely the serialization property that we need to guarantee.

Slide 32.4.3

The mutex is a mutable object (here we'll use a one-element list, which we'll refer to as a cell) that can hold the value true or false. When the value is false, the mutex is available to be acquired. When the value is true, the mutex is unavailable, and any process that attempts to acquire the mutex must wait.
Our mutex constructor `make-mutex` begins by initializing the cell contents to false. To acquire the mutex, we test the cell. If the mutex is available, we set the cell contents to true and proceed. Otherwise, we wait in a loop, attempting to acquire over and over again, until we find that the mutex is available. To release the mutex, we set the cell contents to false.





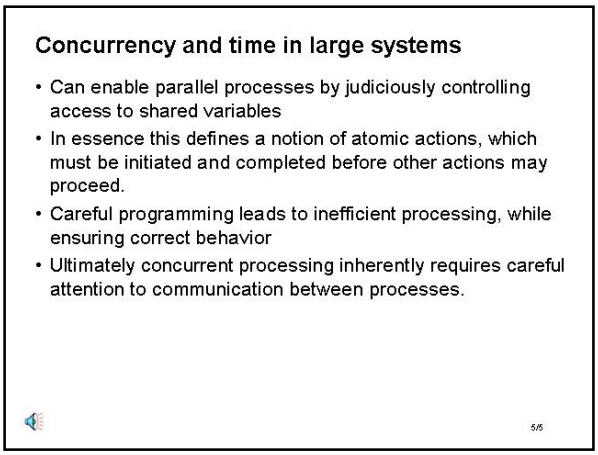
Slide 32.4.4

`Test-and-set!` tests the cell and returns the result of the test. In addition, if the test was false, `test-and-set!` sets the cell contents to true before returning false. We can express this behavior with the procedure shown.
However, this implementation of `test-and-set!` does not suffice as it stands. There is a crucial subtlety here, which is the essential place where concurrency control enters the system: The `test-and-set!` operation must be performed atomically. That is, 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. If we do not make this guarantee, then the mutex can fail.
The actual implementation of `test-and-set!` depends on the details of how our system runs concurrent

processes. For example, we might be executing concurrent processes on a sequential processor using a time-slicing mechanism that cycles through the processes, permitting each process to run for a short time before interrupting it and moving on to the next process. In that case, `test-and-set!` can work by disabling time slicing during the testing and setting.

Alternatively, multiprocessing computers provide instructions that support atomic operations directly in hardware.

Slide 32.4.5



We've seen how programming concurrent systems requires controlling the ordering of events when different processes access shared state, and we've seen how to achieve this control through judicious use of serializers. But the problems of concurrency lie deeper than this, because, from a fundamental point of view, it's not always clear what is meant by ``shared state.''

Mechanisms such as `test-and-set!` require processes to examine a global shared flag at arbitrary times. This is problematic and inefficient to implement in modern high-speed processors, where due to optimization techniques such as pipelining and cached memory, the contents of memory may not be in a consistent state at every instant. In contemporary multiprocessing systems, therefore, the serializer paradigm is being supplanted by new approaches to concurrency control.

The basic phenomenon here is that synchronizing different processes, establishing shared state, or imposing an order on events requires communicating among the processes. In essence, any notion of time in concurrency control must be intimately tied to communication.