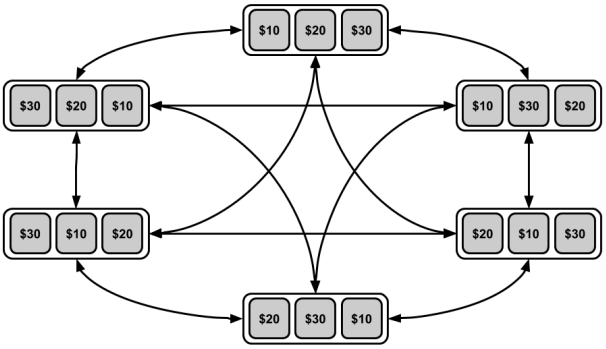

Figure 1: Possible states of the three bank accounts when the `exchange` procedure is applied in serial.

Exercise 3.43

Problem statement

Suppose that the balances in three accounts start out as $10, $20, and $30, and that multiple processes run, exchanging the balances in the accounts. Argue that if the processes are run sequentially, after any number of concurrent exchanges, the account balances should be $10, $20, and $30 in some order. Draw a timing diagram like the one in Figure 3.29 to show how this condition can be violated if the exchanges are implemented using the first version of the account-exchange program in this section. On the other hand, argue that even with this exchange program, the sum of the balances in the accounts will be preserved. Draw a timing diagram to show how even this condition would be violated if we did not serialize the transactions on individual accounts.

Solution

Running Processes Sequentially

```
(define (exchange account1 account2)
  (let ((difference (- (account1 'balance)
                       (account2 'balance))))
    ((account1 'withdraw) difference)
    ((account2 'deposit) difference)))
```

As the exchange procedure above swaps the balances of the two accounts one will always end up having a permutation of the initial balances. Figure 1 shows the possible states of the system. Observe that no matter how we follow the arrows trying to swap accounts we will always end up with a set of balances that are some permutation of $\{\$10, \$20, \$30\}$.

Running Processes Simultaneously

Running the processes simultaneously can result in a final account structure that is not a permutation of the initial accounts. The diagram in Figure 2 shows such a situation.

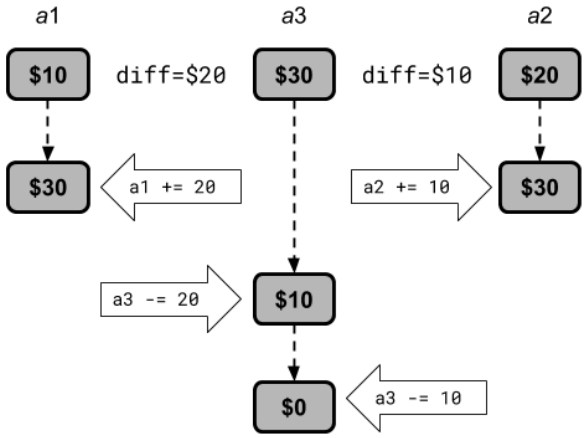

Figure 2: An example of two exchanges running at the same time which results in balances that are not permutations of $\{\$10, \$20, \$30\}$.

But does the sum of money in the accounts always stay the same? In the trivial case, where we run the exchanges serially – yes. As we've seen, regardless of the operations the end result is a permutation of the initial sums. In the other cases, we need to notice that the difference is calculated at the start of the exchange procedure, and one account will receive that money while another one will be depleted of it. At the end of the cycle, no money is actually gained or lost from the system. As operations are atomic, and withdraw operations on a single account are serialized no exchange operation will make money *disappear* from the overall system.

No serialization

Remember why we serialized the `withdraw` procedure in the first place — it accessed the internal value of balance. Similarly the `deposit` procedure called from `exchange` looks up the balance internally. Serialization makes sure that no other procedure can modify the balance of the system before the current one finishes modifying it.

```
(define (deposit amount)
    (set! balance (+ balance amount))
    balance)
```

If we were to run `(parallel-execute (exchange a1 a3) (exchange a2 a3))` we could end up with two simultaneous deposit calls to account $a3$ – which now would not be serialized! Imagine the last two arrows in Figure 2 now try to write the same information at the same time. Figure 3 shows how the final balance on account $a3$ could be set to \$20, leaving us with a total balance of \$80 – which doesn't preserve the system total of \$60!

In conclusion – running all the operations in sequence results in correct but slow behavior. Serializing operations on accounts preserves the amount of money in the system, but can leave an account in debt incorrectly. Not serializing any operation puts the whole system at risk as not even the total amount of money is preserved.

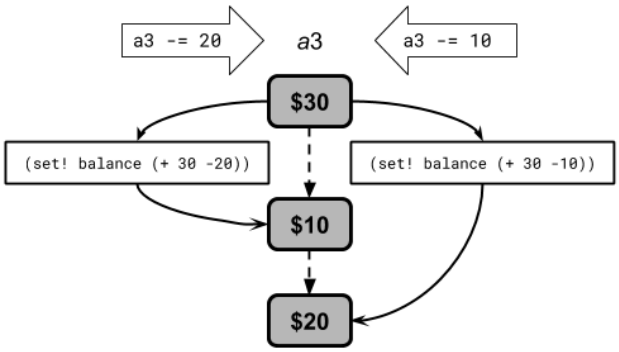

Figure 3: An example how two unserialized calls to deposit could result in a final balance on account $a3$ that doesn't preserve the amount of total money in the system. Both `deposit` procedures could read the initial balance of 30 and aim to make modify it based on that value. This ends up resulting in aberrant system behavior.