T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	

Figure 17.3 Schedule 2—a serial schedule in which T_2 is followed by T_1.

These schedules are **serial**: Each serial schedule consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule. Recalling a well-known formula from combinatorics, we note that, for a set of n transactions, there exist n factorial ($n!$) different valid serial schedules.

When the database system executes several transactions concurrently, the corresponding schedule no longer needs to be serial. If two transactions are running concurrently, the operating system may execute one transaction for a little while, then perform a context switch, execute the second transaction for some time, and then switch back to the first transaction for some time, and so on. With multiple transactions, the CPU time is shared among all the transactions.

Several execution sequences are possible, since the various instructions from both transactions may now be interleaved. In general, it is not possible to predict exactly how many instructions of a transaction will be executed before the CPU switches to another transaction.[1]

Returning to our previous example, suppose that the two transactions are executed concurrently. One possible schedule appears in Figure 17.4. After this execution takes place, we arrive at the same state as the one in which the transactions are executed serially in the order T_1 followed by T_2. The sum $A + B$ is indeed preserved.

[1]The number of possible schedules for a set of n transactions is very large. There are $n!$ different serial schedules. Considering all the possible ways that steps of transactions might be interleaved, the total number of possible schedules is much larger than $n!$.