

T_1	T_2
<code>read(A)</code> <code>A := A - 50</code> <code>write(A)</code>	<code>read(A)</code> <code>temp := A * 0.1</code> <code>A := A - temp</code> <code>write(A)</code>
<code>read(B)</code> <code>B := B + 50</code> <code>write(B)</code> <code>commit</code>	<code>read(B)</code> <code>B := B + temp</code> <code>write(B)</code> <code>commit</code>

Figure 17.4 Schedule 3—a concurrent schedule equivalent to schedule 1.

Not all concurrent executions result in a correct state. To illustrate, consider the schedule of Figure 17.5. After the execution of this schedule, we arrive at a state where the final values of accounts A and B are \$950 and \$2100, respectively. This final state is an *inconsistent state*, since we have gained \$50 in the process of the concurrent execution. Indeed, the sum $A + B$ is not preserved by the execution of the two transactions.

If control of concurrent execution is left entirely to the operating system, many possible schedules, including ones that leave the database in an inconsistent state, such as the one just described, are possible. It is the job of the database system to ensure that any schedule that is executed will leave the database in a consistent state. The **concurrency-control** component of the database system carries out this task.

We can ensure consistency of the database under concurrent execution by making sure that any schedule that is executed has the same effect as a schedule that could have occurred without any concurrent execution. That is, the schedule should, in some sense, be equivalent to a serial schedule. Such schedules are called **serializable** schedules.

17.6 Serializability

Before we can consider how the concurrency-control component of the database system can ensure serializability, we consider how to determine when a schedule is serializable. Certainly, serial schedules are serializable, but if steps of multiple transactions are interleaved, it is harder to determine whether a schedule is serializable. Since trans-