

3.10 Tracing, Testing, and Replay for Semaphores and Locks

Outline:

- A technique for detecting violations of mutual exclusion.
- Tracing and replaying executions during debugging.
- Deadlock detection
- Reachability testing

3.10.1 Non-Deterministic Testing with the Lockset Algorithm

Recall from Chapter 1 that a data race is a failure to correctly implement critical sections for non-atomic shared variable accesses.

A concurrent program can be tested for data races:

- monitor shared variable accesses and make sure that each variable has been properly locked before it is accessed.
- execute the program several times with the same test input in order to increase our chances of finding data races.

This type of testing is called *non-deterministic testing*.

Non-deterministic testing of a concurrent program CP involves the following steps:

1. Select a set of inputs for CP.
2. For each selected input X, execute CP with X many times and examine the result of each execution.

The purpose of non-deterministic testing is to exercise as many behaviors as possible.

- Unfortunately, experiments have shown that programs tend to exhibit the same behavior from execution to execution.
- Also, “probe effect” (see Section 1.7) may make it impossible for some failures to be observed.

To increase the likelihood of exercising different behaviors:

- change the scheduling algorithm used by the operating system, e.g., change the value of the time quantum that is used for round-robin scheduling.
- insert Sleep(*t*) statements into the program with the sleep amount *t* randomly chosen. Executing a Sleep statement forces a context switch and thus indirectly affects thread scheduling.

We have implemented the second technique as an execution option for programs that use the *binarySemaphore*, *countingSemaphore*, and *mutexLock* classes in our synchronization library. When this option is specified, Sleep statements are executed at the beginning of methods *P()*, *V()*, *lock()* and *unlock()*.

To detect data races, we combine non-deterministic testing with the lockset algorithm:

- checks that all shared variables follow a consistent locking discipline in which every shared variable is protected by a lock.
- for each variable, we determine if there is some lock that is always held whenever the variable is accessed.

For shared variable *v*, let the set *CandidateLocks(v)* be those locks that have protected *v* during the execution so far. Thus, a lock *l* is in *CandidateLocks(v)* if, during the execution so far, every thread that has accessed *v* was holding *l* at the moment of access.

CandidateLocks(v) is computed as follows:

- When a new variable *v* is initialized, its candidate set is considered to hold all possible locks.
- When *v* is accessed on a read or write operation by T, *CandidateLocks(v)* is refined. The new value of *CandidateLocks(v)* is the intersection of *CandidateLocks(v)* and the set of locks held by thread T.

Based on this refinement algorithm:

- if some lock l consistently protects v , it will remain in $CandidateLocks(v)$ as $CandidateLocks(v)$ is refined.
- if $CandidateLocks(v)$ becomes empty, it indicates that there is no lock that consistently protects v .

Lockset Algorithm:

```
// Let  $LocksHeld(T)$  denote the set of locks currently held by thread  $T$ 
For each shared variable  $v$ , initialize  $CandidateLocks(v)$  to the set of all locks.
On each read or write access to  $v$  by thread  $T$ :
     $CandidateLocks(v) = CandidateLocks(v) \cap LocksHeld(T)$ ;
    if ( $CandidateLocks(v) == \{\}$ )
        issue a warning;
```

In Fig. 3.30, Thread1's access of shared variable s is protected first by $mutex1$ then by $mutex2$. This violation of mutual exclusion can be detected by the lockset algorithm:

- $CandidateLocks(s)$ is initialized to $\{mutex1, mutex2\}$ and is refined as s is accessed.
- When Thread1 locks $mutex1$, $LocksHeld(Thread1)$ becomes $\{mutex1\}$.
- When s is accessed in the first assignment statement, $CandidateLocks(s)$ becomes $mutex1$, which is the intersection of sets $CandidateLocks(s)$ and $LocksHeld(Thread1)$.
- When the second assignment statement is executed, Thread1 holds lock $mutex2$ and the only candidate lock of s is $mutex1$.
- After the intersection of $CandidateLocks(s)$ and $LocksHeld(Thread1)$, $CandidateLocks(s)$ becomes empty.

The lockset algorithm has detected that no lock consistently protects shared variable s .

We have implemented the lockset algorithm in the *mutexLock* class and in the C++ *sharedVariable* class template that was presented in Chapter 2:

- We assume that threads can access a *sharedVariable* only after it is initialized. Thus, no refinement is performed during initialization, i.e., in the constructor of *sharedVariable*.
- Read-only variables can be accessed without locking. This means that warnings are issued only after a variable has been initialized and has been accessed by at least one write operation.
- The refinement algorithm can be turned off to eliminate false alarms. For example, a bounded buffer is shared by *Producer* and *Consumer* threads but no locks are needed. Even so, when this program is executed a warning will be issued by the lockset algorithm.

As a non-deterministic testing technique, the lockset algorithm cannot prove that a program is free from data races.

3.10.2 Simple SYN-sequences for Semaphores and Locks

We can characterize an execution of a concurrent program as a sequence of synchronization events on synchronization objects.

A sequence of synchronization events is called a SYN-sequence.

There are several ways to define a SYN-sequence and the definition of a SYN-sequence has an affect on the design of a replay solution, and vice versa.

Let CP be a concurrent program that uses shared variables, semaphores, and locks.

The result of executing CP with a given input depends on the (unpredictable) order in which the shared variables, semaphores, and locks in CP are accessed.

- The semaphores are accessed using *P* and *V* operations
- The locks are accessed using *lock* and *unlock* operations
- The shared variables are accessed using *read* and *write* operations.

⇒ The synchronization objects in CP are its shared variables, semaphores, and locks. The synchronization events in CP are executions of *read/write*, *P/V*, and *lock/unlock* operations on these objects.

A SYN-sequence for a shared variable *v* is a sequence of *read* and *write* operations on *v*. ReadWrite-sequences for shared variables were defined in Chapter 2.

A SYN-sequence for a *binarySemaphore* or *countingSemaphore* *s* is a sequence of events of the following types:

- completion of a *P* operation
- completion of a *V* operation
- start of a *P* operation that is never completed due to a deadlock or an exception
- start of a *V* operation that is never completed due to a deadlock or an exception

We refer to such a sequence as a PV-sequence of *s*. An event in a PV-sequence is denoted by the identifier (ID) of the thread that executed the *P* or *V* operation.

The order in which threads complete their *P* and *V* operations is not necessarily the same as the order in which they call *P* and *V* or even the same as the order in which the *P* and *V* operations start.

For the operations that are completed, it is their order of completion that must be replayed, since this order determines the result of the execution.

We will also replay the starts of operations that do not complete, so that the same events, exceptions, and deadlocks will occur during replay.

A SYN-sequence for a *mutexLock* *l* is a sequence of events of the following types:

- completion of a *lock* operation
- completion of an *unlock* operation
- start of a *lock* operation that is never completed due to a deadlock or an exception
- start of an *unlock* operation that is never completed due to a deadlock or an exception

We refer to such a sequence as a LockUnlock-sequence of *l*. An event in a LockUnlock-sequence is denoted by the identifier (ID) of the thread that executed the *lock* or *unlock* operation.

Example: Consider the simple program in Listing 3.31. The final value of shared variable x is either 1 or 2.

```
binarySemaphore mutex(1);
Thread1      Thread2
  mutex.P(); mutex.P();
x = 1;       x = 2;
mutex.V();   mutex.P(); // error: should be mutex.V();
```

Listing 3.31 Illustrating ReadWrite-sequences and PV-sequences

A possible ReadWrite-sequence of shared variable x is:

(1, 0, 0), (2, 1, 0). // from Ch. 2, the format is (thread ID, version number, total readers)

This denotes that x was first accessed by Thread 1 and then by Thread 2.

Since Thread1 accessed x first, the PV-sequence for *mutex* must be:

1, 1, 2, 2

indicating that Thread1 performed its P and V operations before Thread2. The second P operation in Thread2 is an error. This P operation will start but not complete and should be a V operation instead.

A SYN-sequence for concurrent program CP is a collection of ReadWrite-sequences, PV-sequences, and LockUnlock-sequences. There is one sequence for each shared variable, semaphore, and lock in the program.

A SYN-sequence for the program in Listing 3.31 contains a ReadWrite-sequence for x and a PV-sequence for *mutex*:

((ReadWrite-sequence of x : (1,0,0), (2,1,0); PV-sequence of *mutex*: (1, 1, 2, 2)).

This is a partial ordering of the events, i.e., the events on a single object are (totally) ordered, but the order of events among different objects is not specified.

A SYN-sequence can also be a single totally-ordered sequence of synchronization events over all the synchronization objects. A totally-ordered sequence of events that is consistent with the above partially-ordered sequence is:

1, (1, 0, 0), 1, 2, (2, 1, 0), 2.

In general, there may be two or more totally-ordered sequences that are consistent with a given partial ordering since two concurrent events can appear in the total-ordering in either order.

The definition of a SYN-sequence is intended to capture what it means for one execution to replay another. Suppose that when the program above is executed:

- Thread2 executes *mutex.P()* and blocks because Thread1 is in its critical section.
- During replay of this execution, assume Thread2 executes its first *mutex.P()* operation without blocking, because Thread1 has already executed its *mutex.P()* and *mutex.V()* operations.
- These two executions are not identical, but are they close enough?

In both executions:

- the sequence of *completed P()* and *V()* operations is the same
- the final value of x is 2.

Thus, we consider the second execution to replay the first.

3.10.3 Tracing and Replaying Simple PV-sequences and LockUnlock-sequences

3.10.3.1 Modifying Methods P() and V().

Tracing: the identifier of the thread that completes a call to method *s.P()* or *s.V()* is recorded and saved to a trace file for *s*.

Replay: we assume that each semaphore has a permit, called a PV-permit.

- A thread must hold a semaphore's PV-permit before it executes a *P()* or *V()* operation on that semaphore.
- The order in which threads receive a semaphore's PV-permit is based on the PV-sequence that is being replayed.
- A thread requests and releases a semaphore's PV-permit by calling methods *requestPermit()* and *releasePermit()*.

```
void P() {
    if (replayMode)
        control.requestPermit(ID);
    // code to lock this semaphore appears here
    if (replayMode)
        control.releasePermit();
    /* rest of body of P() */
    if (traceMode)
        control.traceCompleteP(ID);
    // code to unlock this semaphore appears here
}
```

```
public final void V() {
    if (replayMode)
        control.requestPermit(ID);
    // code to lock this semaphore appears here
    if (replayMode)
        control.releasePermit();
    /* rest of body of V() */
    if (traceMode)
        control.traceCompleteV(ID);
    // code to unlock this semaphore appears here
}
```

3.10.3.2 Modifying Methods lock() and unlock().

The implementations of methods *lock()* and *unlock()* in class *mutexLock* are modified just like methods *P()* and *V()*.

- Class *mutexLock* contains calls to *requestPermit()* and *releasePermit()* before and after, respectively, the lock operation in *mutexLock*.
- Calls to *traceCompleteLock()* and *traceCompleteUnlock()* appear at the end of their respective critical sections.

Deadlocks and exceptions:

- When deadlock producing operations are replayed, the calling threads will not be blocked inside the body of the operation; rather, they will be blocked forever on the call to *requestPermit()* before the operation.
- Events involving exceptions that occur during the execution of a *P*, *V*, *lock*, or *unlock* operation are not replayed. But the trace would indicate that the execution of these operations would raise an error, which is probably enough help to debug the program.

3.10.3.3 Class control.

Each semaphore and lock is associated with a control object.

- Replay mode: the control object inputs the simple SYN-sequence of the semaphore or lock and handles the calls to *requestPermit()* and *releasePermit()*.
- Trace mode: the control object collects the synchronization events that occur and records them in a trace file.

When a thread calls *requestPermit()* or one of the *trace* methods, it passes its identifier (*ID*). (Class *TDThread* in Chapter 1 handles the creation of unique thread *IDs*.)

A C++ *control* class is shown in Listing 3.32.

When the control object for a semaphore *s* is created, it reads the simple PV-sequence for *s* into vector *SYNsequence*.

Method *requestPPermit()*: Threads that try to execute an operation out of turn are delayed in method *requestPPermit()* on a separate semaphore in the *Threads* array.

A thread uses its *ID* to determine which semaphore in the *Threads* array to wait on.

Method *releasePPermit()*: increments *index* to allow the next operation in the *SYNsequence* to occur. If the thread that is to perform the next entry is blocked in *requestPPermit()*, it is awakened.

```
class control {
public:
    control() {
        /* input integer IDs into SYNsequence; initialize arrays threads and
        hasRequested */
    }
    void requestPermit(int ID) {
        mutex.lock();
        if (ID != SYNsequence[index]) { // thread ID should execute next event?
            hasRequested[ID] = true; // No; set flag to remember ID's request
            mutex.unlock();
            threads[ID].P(); // wait for permission
            hasRequested[ID] = false; // reset flag and exit requestPermit
        }
        else mutex.unlock(); // Yes; exit requestPermit.
    }
    void releasePermit() {
        mutex.lock();
        ++index;
        if (index < SYNsequence.size()) { // Are there more events to replay?
            // Has the next thread already requested permission?
            if (hasRequested[SYNsequence[index]])
                threads[SYNsequence[index]].V(); // Yes; wake it up.
        }
        mutex.unlock();
    }
    void traceCompleteP(int ID) { ... } // record integer ID
    void traceCompleteV(int ID) { ... } // record integer ID
private:
    // PV-sequence or LockUnlock-sequence; a sequence of integer IDs
    vector SYNsequence;
    binarySemaphore* threads; // all semaphores are initialized to 0
    // hasRequested[i] is true if Thread i is delayed in requestPermit(); init to false
    bool* hasRequested;
    int index = 0; // SYNsequence[index] is ID of next thread to execute an event
    mutexLock mutex; // note: no tracing or replay is performed for this lock
}
```

Listing 3.32 C++ Class *control* for replaying PV-sequences and LockUnlock-sequences.

To illustrate the operation of the controller, consider a corrected version of the simple program in Listing 3.31.

<u>Thread1</u>	<u>Thread2</u>
mutex.P();	mutex.P();
x = 1;	x = 2;
mutex.V();	mutex.V();

Assume that the PV-sequence traced during an execution of this program is: 1, 1, 2, 2.

Suppose that Thread2 tries to execute *mutex.P()* first and thus calls *requestPermit(2)* before Thread1 calls *requestPermit(1)*:

- Since the value of *index* is 0 and the value of *SYNsequence[index]* is 1 not 2, Thread2 blocks itself in *requestPermit()* by executing *Threads[2].P()*.
- When Thread1 eventually calls *requestPermit(1)*, it will be allowed to exit *requestPermit()* and execute its *mutex.P()* operation.
- Thread1 will then call *releasePermit()*. Method *releasePermit()* increments *index* to 1 and checks whether the thread that is to execute the next *P/V* operation already has already called *requestPermit()*.
- The next thread is *SYNsequence[1]*, which is 1. Thread1 has not called *requestPermit()* for the next operation, so nothing further happens in *releasePermit()*.
- Eventually Thread1 calls *requestPermit(1)* to request permission to execute its *mutex.V()* operation. Thread1 receives permission, executes *mutex.V()* and calls *releasePermit()*.
- Method *releasePermit()* increments *index* to 2 and finds that the thread to execute the next *P/V* operation is Thread2.
- Thread 2, having already called *requestPermit()*, is still blocked on its call to *Threads[2].P()*. This is indicated by the value of *hasRequested[2]*, which is *true*. Thus, *releasePermit()* calls *Threads[2].V()*.
- This allows Thread2 to exit *requestPermit()* and perform its *mutex.P()* operation.
- Thread 2 will eventually request and receive permission for its *mutex.V()* operation, completing the replay.

3.10.4 Deadlock Detection

In Chapter 2, we defined a deadlock as a situation in which one or more threads become blocked forever.

Let CP be a concurrent program containing threads that use semaphores and locks for synchronization. Assume there is an execution of CP that exercises a SYN-sequence S, and at the end of S, there exists a thread T that satisfies these conditions:

- T is blocked due to the execution of a *P()*, *V()*, or *lock()* statement.
- T will remain blocked forever, regardless of what the other threads will do.

Thread T is said to be deadlocked at the end of S, and CP is said to have a deadlock. A deadlock in CP is a *global deadlock* if every thread in CP is either blocked or completed; otherwise, it is a *local deadlock*.

In operating systems, processes request resources (e.g., printers and files) and enter a wait-state if the resources are held by other processes. If the requested resources can never become available then the processes can never leave their wait-state and a deadlock occurs.

The information about which process is waiting for a resource held by which other process can be represented by a “wait-for” graph:

- an edge in a wait-for graph from node P_i to P_j indicates that process P_i is waiting for process P_j to release a resource that P_i needs.
- a deadlock exists in the system if and only if the wait-for graph contains a cycle.
- periodically invoke an algorithm that searches for cycles in the wait-for graph.

Deadlock detection using wait-for graphs is not always applicable to concurrent programs. A thread blocked in a *P()* operation, for example, does not know which of the other threads can unblock it, and thus the wait-for relation among the threads is unknown.

Another approach: Assume that a deadlock occurs if all of the threads in a program are permanently blocked. Assume also that all of the threads are expected to terminate.

To detect deadlocks, maintain a count of the threads that have not completed their *run()* methods, and a count of the blocked threads, and compare the two counts:

- The *numThreads* counter is incremented when a thread starts its *run()* method and decremented when a thread completes its *run()* method.
- The *blockedThreads* counter is incremented when a thread blocks in a *P()*, *V()*, or *lock()* operation, and decremented when a thread is unblocked during a *V()* or *unlock()* operation. The *blockedThreads* counter should also be maintained in other blocking methods such as *join()*.
- If the *numThreads* and *blockedThreads* counters are ever equal (and non-zero), then all of the threads are blocked and we assume that a deadlock has occurred.

This approach can be used to detect global deadlocks, but not local deadlocks, since it requires all non-deadlocked threads to be completed.

This approach is implemented in the synchronization classes, but the implementation is incomplete:

- the *blockedThreads* counter is not modified inside the *join()* method.
- the *numThreads* counter is not modified when the *main* thread starts and completes, simply because there is no convenient and transparent way to do so.
- Since the *main* thread is not included in *numThreads* and the status of the *main* thread is unknown, it is possible for the *numThreads* and *blockedThreads* counters to become temporarily equal (indicating that a deadlock has occurred) and then for more threads to be created and start running. Thus, it is not known for sure that a deadlock has occurred when the counters are equal.

To handle the uncertainty about deadlock, the *numThreads* and *blockedThreads* counters are compared only after no synchronization events have been generated for some period of time, indicating that all the threads are blocked or completed, or that running threads are stuck in infinite loops, i.e., livelocked.

Status information is displayed when a deadlock is detected. The events leading to a deadlock can be traced and replayed using the methods described earlier:

Deadlock detected:

- philosopher3 blocked on operation P() of semaphore chopstick[4]
- philosopher2 blocked on operation P() of semaphore chopstick[3]
- philosopher4 blocked on operation P() of semaphore chopstick[0]
- philosopher1 blocked on operation P() of semaphore chopstick[2]
- philosopher0 blocked on operation P() of semaphore chopstick[1]

Some wait-for relations among threads can be captured by wait-for graphs:

- a thread blocked in a *lock()* operation, knows that the thread that owns the lock can unblock it. Thus, the wait-for relation among threads and locks is known.
- when one thread tries to lock two locks in one order, while another thread tries to lock them in reverse order, a deadlock may occur.
- such a deadlock will be indicated by a cycle in the wait-for graph.

A deadlock detection utility has been incorporated into the Java HotSpot VM:

- This utility is invoked by typing Ctrl+ (for Linux or the Solaris Operating Environment) or Ctrl-Pause/Break (for Microsoft Windows) on the command line while an application is running.
- If the application is deadlocked because two or more threads are involved in a cycle to acquire locks, then the list of threads involved in the deadlock is displayed.
- This utility will not find deadlocks involving one or more threads that are permanently blocked in a monitor (see Chapter 4) waiting for a notification that never comes, which is similar to the situation where a thread is blocked in a *P()* operation waiting for a *V()* operation that will never come.

3.10.5 Reachability Testing for Semaphores and Locks

Non-deterministic testing is easy to carry out, but it can be very inefficient. It is possible that some behaviors will be exercised many times, while others will not be exercised at all.

For Solution 1 to the dining philosophers problem with five philosophers:

- no deadlock was detected during 100 executions.
- after inserting random delays, deadlocks were detected in eighteen out of one hundred executions.
- for ten philosophers, deadlocks were detected in only four out of one hundred executions.

As the number of threads increases and thus the total number of possible behaviors increases, it apparently becomes harder to detect deadlocks with non-deterministic testing.

Furthermore, the instrumentation added to perform tracing and deadlock detection may create a probe effect that prevents some failures from being observed.

Reachability testing allows us to examine all the behaviors of a program, or at least as many different behaviors as is practical, in a “systematic” manner.

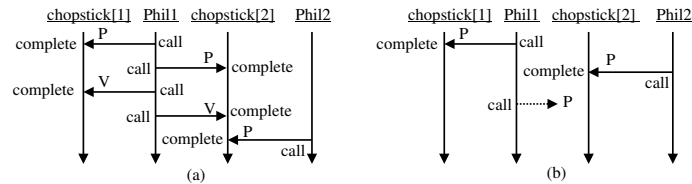
By systematic, we mean that a given SYN-sequence is exercised only once and it is possible to know when all the SYN-sequences have been exercised.

Reachability testing combines non-deterministic testing and program replay.

During reachability testing, the SYN-sequence exercised by a non-deterministic execution is traced as usual.

- The execution trace captures the behavior that actually happened.
- The trace is also carefully defined so that it captures alternate behaviors that could have happened, but didn't.
- These alternate behaviors are called "race variants" of the trace. Replaying a race variant ensures that a different behavior is observed during the next test execution.

Fig. 3.33a shows a portion of an execution trace for the dining philosophers program. Philosopher₁ and Philosopher₂ race to pick up chopstick₂, with Philosopher₁ completing its *P()* operation on chopstick[2] before Philosopher₂ can complete its *P()* operation. (Philosopher₁ and Philosopher₂ share chopstick[2], which lies between them.)



In general, there is a race between calls to *P()* or *V()* operations on the same semaphore if these calls could be completed in a different order during another execution (with the same input).

Fig. 3.33b shows a race variant of the execution in Fig. 3.33a.

- In this race variant, Philosopher₂ wins the race to chopstick[2] and picks it up before Philosopher₁ can grab it.
- The dashed arrow indicates that Philosopher₁'s *P()* operation on chopstick[2] was called but not completed in the race variant. This call will be completed in an execution that replays the variant.

In general, a race variant represents the beginning portion of a SYN-sequence.

Reachability testing uses replay to make sure that the events in the race variant are exercised, and then lets the execution continue non-deterministically so that a complete sequence can be traced.

There may be many complete SYN-sequences that have a given race variant at the beginning.

- One of these sequences will be captured in the non-deterministic portion of the execution.
- The complete traced sequence can then be analyzed to derive more race variants, which can be used to generate more traces, and so on.

When replay is applied to the race variant in Fig 3-33b:

- Philosopher₁ and Philosopher₂ will both pick up their left chopsticks, which is part of the deadlock scenario in which all the philosophers hold their left chopstick and are waiting for their right.
- While the complete deadlock scenario may not occur immediately, reachability testing ensures that every possible PV-sequence of the dining philosophers program will eventually be exercised and thus that the deadlock will eventually be detected.

Table 3.2 shows the results of applying reachability testing to three Java programs.

- BB is the bounded buffer program from Listing 3.6, with multiple producers and consumers.
- DP1 and DP4 are solutions 1 and 4 of the dining philosophers problem in Listing 3.7 and Listing 3.8, respectively. (Recall that program DP1 has a deadlock.)
- RW is the readers and writers program from Listing 3.11.

The threads in these programs deposit, withdraw, read, write, or eat one time

The second column shows the configuration of each program.

1. For BB, it indicates the number of producers (P), the number of consumers (C), and the number of slots (S) in the buffer.
2. For RW, it indicates the number of readers (R) and the number of writers (W).
3. For DP, it indicates the number of philosophers (P).

The third column shows the number of PV-sequences generated during reachability testing. (The total reachability testing time for the DP4 program with 5 philosophers is 15 minutes on a 1.6GHz PC with 512 MB of RAM.)

Program	Configuration	#Seqs.	Program	Configuration	#Seqs.	Program	Configuration	#Seqs.
BB	1P+1C+2S	1	RW	2R+2W	608	DP1	5P/10P	31/1023
BB	2P+2C+2S	132	RW	2R+3W	12816	DP4	4P	720
BB	3P+3C+2S	9252	RW	3R+2W	21744	DP4	5P	22300

Table 3.2 Reachability testing for the example programs in Section 3.5.

Observations about Table 3.2:

1. There is only one possible sequence for program BB with one producer and one consumer. There are six $P()$ and $V()$ operations exercised during an execution of BB, why aren't there more sequences?
 - (a) Reachability testing exercises all of the partially ordered PV-sequences of BB, not all of the totally ordered sequences. That is, in a given execution trace, if events E1 and E2 are concurrent and E1 appears before E2, then no race variant is generated to cover the case where the order of E1 and E2 is reversed.
 - (b) Reachability testing considers only sequences of *completed* $P()$ and $V()$ operations. While there are many different orders in which the producer and consumer threads can *start* their $P()$ and $V()$ operations, there is only one order in which they can *complete* these operations.

2. Program DP1 with five philosophers has a total of 31 sequences, one of which results in a deadlock. For DP1 with 10 philosophers, there is one deadlock sequence out of 1023 possible sequences.
 3. The number of sequences exercised during reachability testing grows quickly as the number of threads increases:
 - a. It may be impractical to exercise all of the sequences, let alone to manually inspect the output of all the test executions.
 - b. Reachability testing does not have to be applied exhaustively; rather, it can stop when a selected coverage criterion is satisfied.
 4. Exercising every PV-sequence may be more effort than is really required.
 - a. Some race variants are created simply by reversing the order of two $V()$ operations. But such variants are not very interesting since the order of the $V()$ operations has no effect on the rest of the execution. Ignoring these and similar types of variants:
 - The number of sequences for BB with 3P + 3C drops from 9252 to 36.
 - The number of sequences for BB with 2P + 2C, drops from 132 to 4.
 - b. Reachability testing with three Producers will exercise all the possible orders in which the three Producers can deposit their items: (Producer₁, Producer₂, Producer₃), (Producer₂, Producer₁, Producer₃), (Producer₃, Producer₂, Producer₁), etc. But the Producers all execute the same code and their behavior is independent of their IDs, so this batch of variants is not very interesting. If you use the symmetry of the threads to ignore these "duplicate" variants,
 - The number of sequences for BB with 3P + 3C drops from 36 to 4.
 - The number of sequences for BB with 2P + 2C drops from 4 to 2 (or perhaps even lower (see Exercise 7.5)).
- These numbers are in line with the numbers we would likely come up with if we tried to generate sequences manually.

3.10.6 Putting it all Together

3.10.6.1 Tracing and Replaying C++/Win32/Pthreads Programs.

Class *TDThread* ensures that threads receive unique thread IDs.

Tracing, replay, and analysis are controlled through the values of environment variables `MODE` and `RACEANALYSIS`. To set tracing *on* in Windows/DOS, execute:

```
set MODE=TRACE
// Unix: setenv MODE TRACE
```

before executing the program.

To execute random delays during `TRACE` mode, execute the command:

```
set RANDOMDELAY=ON
// Unix: setenv RANDOMDELAY ON
```

This will enable the execution of *Sleep* statements at the beginning of methods *P()*, *V()*, *lock()* and *unlock()*. The command “set `RANDOMDELAY=OFF`” will turn off random delays, and `OFF` is the default value.

To turn on deadlock detection during trace mode, execute:

```
set DEADLOCKDETECTION=ON
// Unix: setenv DEADLOCKDETECTION ON
```

The command “set `DEADLOCKDETECTION=OFF`” will turn off deadlock detection, and `OFF` is the default value.

Data race detection using the lockset algorithm is enabled by

```
set DATARACEDETECTION =ON
// Unix: setenv DATARACEDETECTION ON
```

The command “set `DATARACEDETECTION =OFF`” will turn off data race detection, and `OFF` is the default value.

Environment variable `CONTROLLERS` is used to determine the number of trace files that will be created. The value `SINGLE` causes one controller and one trace file to be created:

```
set CONTROLLERS=SINGLE
// Unix: setenv CONTROLLERS SINGLE
```

The trace file `semaphores-replay.txt` will contain a totally-ordered sequence of events for the semaphore and lock objects.

To create a separate controller for each object, which results in a separate trace file for each object’s `SYN`-sequence, use the value `MULTIPLE`:

```
set CONTROLLERS=MULTIPLE
// Unix: setenv CONTROLLERS MULTIPLE
```

The default value for the number of controllers is `SINGLE`.

An execution is replayed by setting the `MODE` to `REPLAY` and executing the program.

```
set MODE=REPLAY
// Unix: setenv MODE REPLAY
```

The value of `CONTROLLERS` must be the same during tracing and replay. No data race detection or random delays are performed during replay.

Reachability testing is performed by setting the `MODE` to `RT` and customizing the driver process in file *RTDriver.cpp*, which is part of the synchronization library. Directions for customizing the driver process are in the file.

3.10.6.2 Tracing and Replaying Java Programs.

Tracing and replay are controlled using a property named *mode*. An execution of Java program *Buffer* is traced using the command:

```
java -Dmode=trace -DrandomDelay=on -Dcontrollers=single Buffer
```

This executes program *Buffer* with random delays and creates a single controller and a trace file named *semaphores-replay.txt*.

File *semaphores-replay.txt* will contain a totally-ordered sequence of events for the semaphore and lock objects.

To create a separate controller for each synchronization object and a separate file for each object's SYN-sequence, set the value of the *controllers* property to *multiple*:

```
-Dcontrollers=multiple
```

The default value for property *controllers* is *single* and the default value for property *randomDelay* is *off*.

To turn on deadlock detection during trace mode, specify *-DdeadlockDetection=on*. The default value for property *deadlockDetection* is *off*

An execution is replayed by setting the *mode* property to *replay*:

```
java -Dmode=replay Buffer
```

The value of *CONTROLLERS* must be the same during tracing and replay. No random delays are performed during replay.

Reachability testing is performed on *Buffer* by setting the *mode* property to *rt* and executing a driver process named *RTDriver* that is part of the synchronization library:

```
java -Dmode=rt RTDriver Buffer // "Buffer" is a command line parameter
```