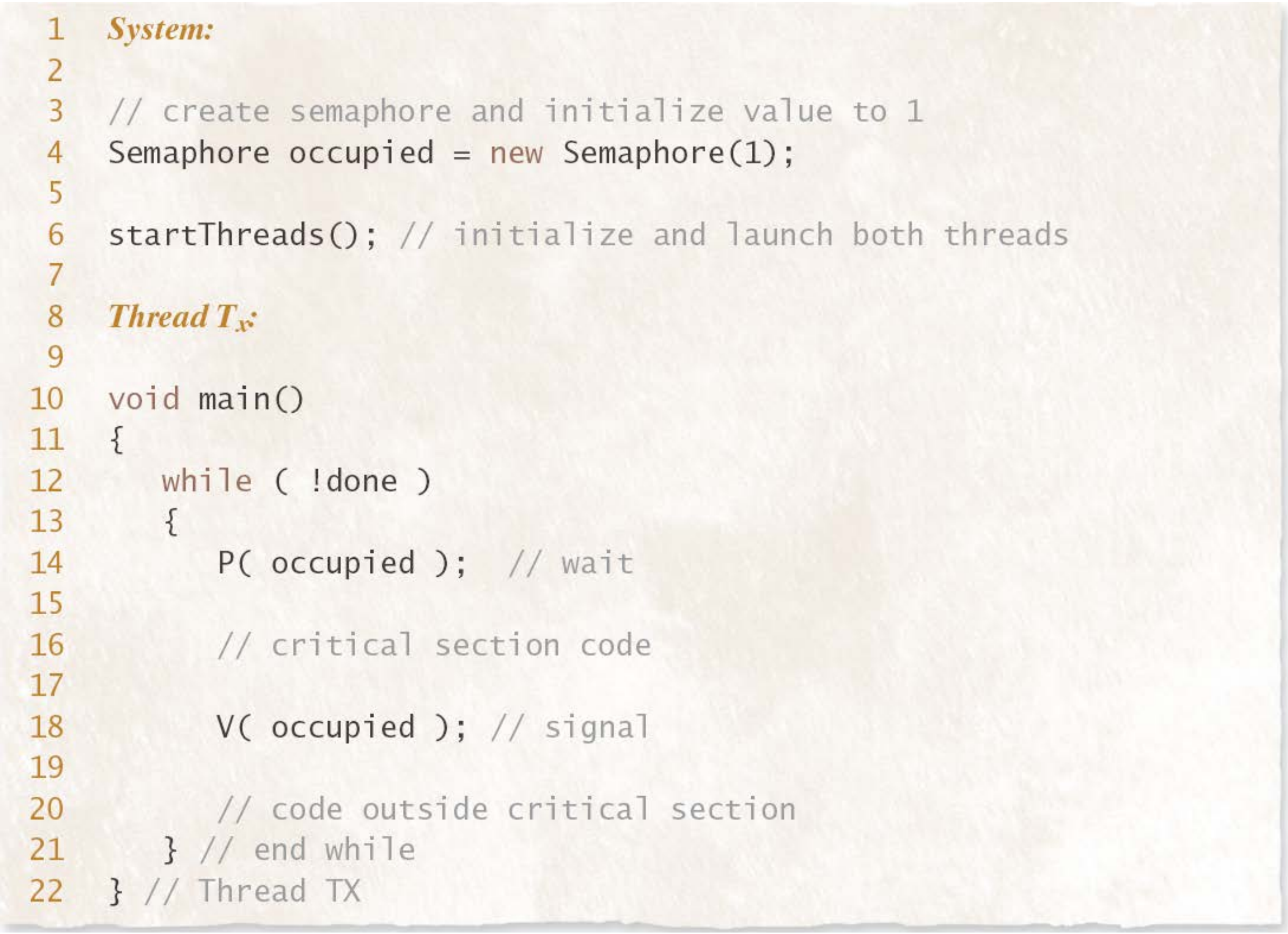
5.6 Semaphores

- Semaphores
  - Software construct that can be used to enforce mutual exclusion
  - Contains a protected variable
    - Can be accessed only via wait (P) and signal (V) commands
- A proper semaphore implementation requires that *P* and *V* be indivisible operations
- If several threads attempt a *P(S)* simultaneously, the implementation should guarantee that only one thread will be allowed to proceed
- The others will be kept waiting, but the implementation of *P* and *V* can guarantee that threads will not suffer indefinite postponement (maintain a FIFO queue)

5.6.1 Mutual Exclusion with Semaphores

Figure 5.15 Mutual exclusion with semaphores.

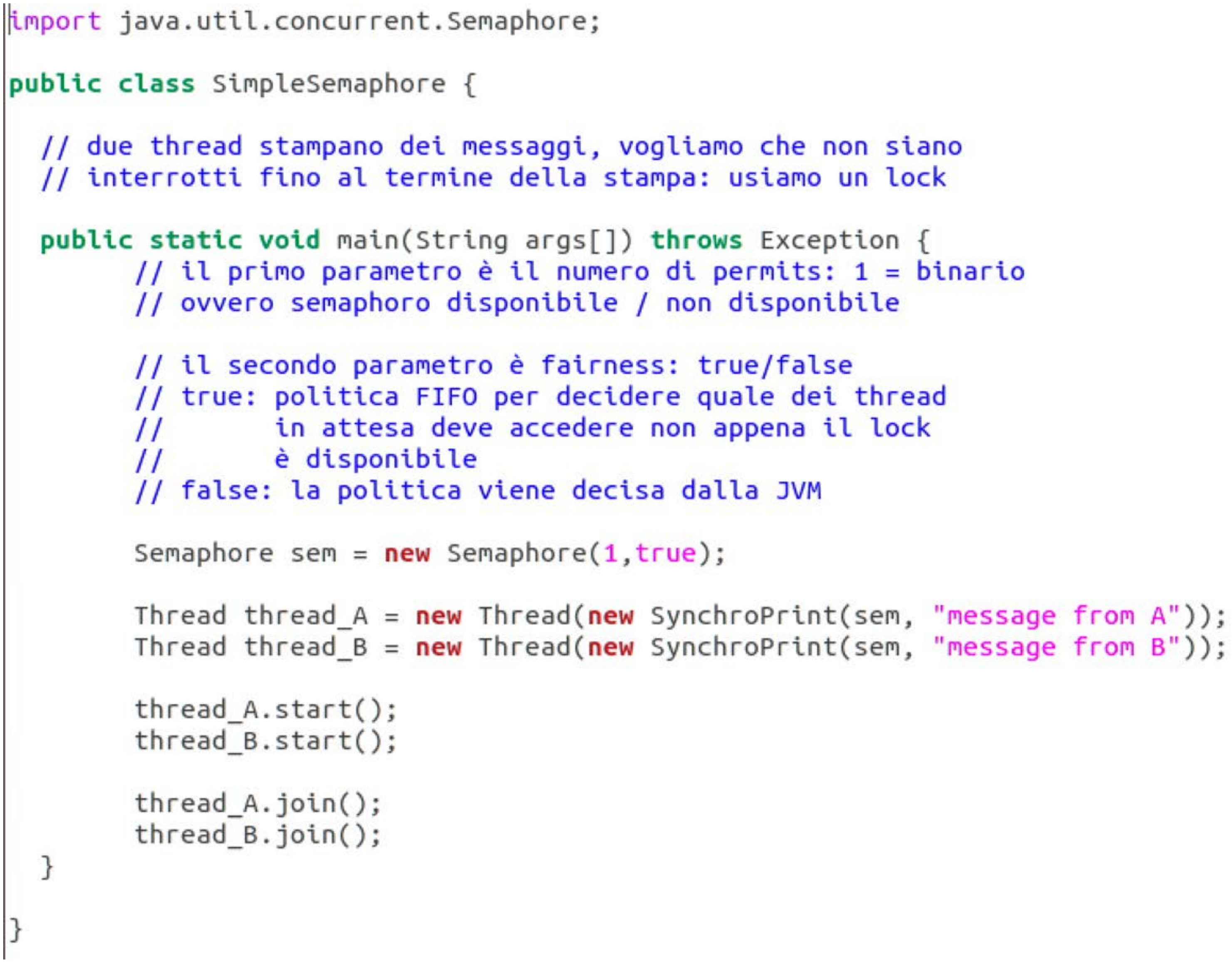


```
System:

// create semaphore and initialize value to 1
Semaphore occupied = new Semaphore(1);

startThreads(); // initialize and launch both threads

Thread Tx:

void main()
{
   while ( !done )
   {
      P( occupied );  // wait

      // critical section code

      V( occupied ); // signal

      // code outside critical section
   } // end while
} // Thread TX
```

5.6.2 Thread Synchronization with Semaphores

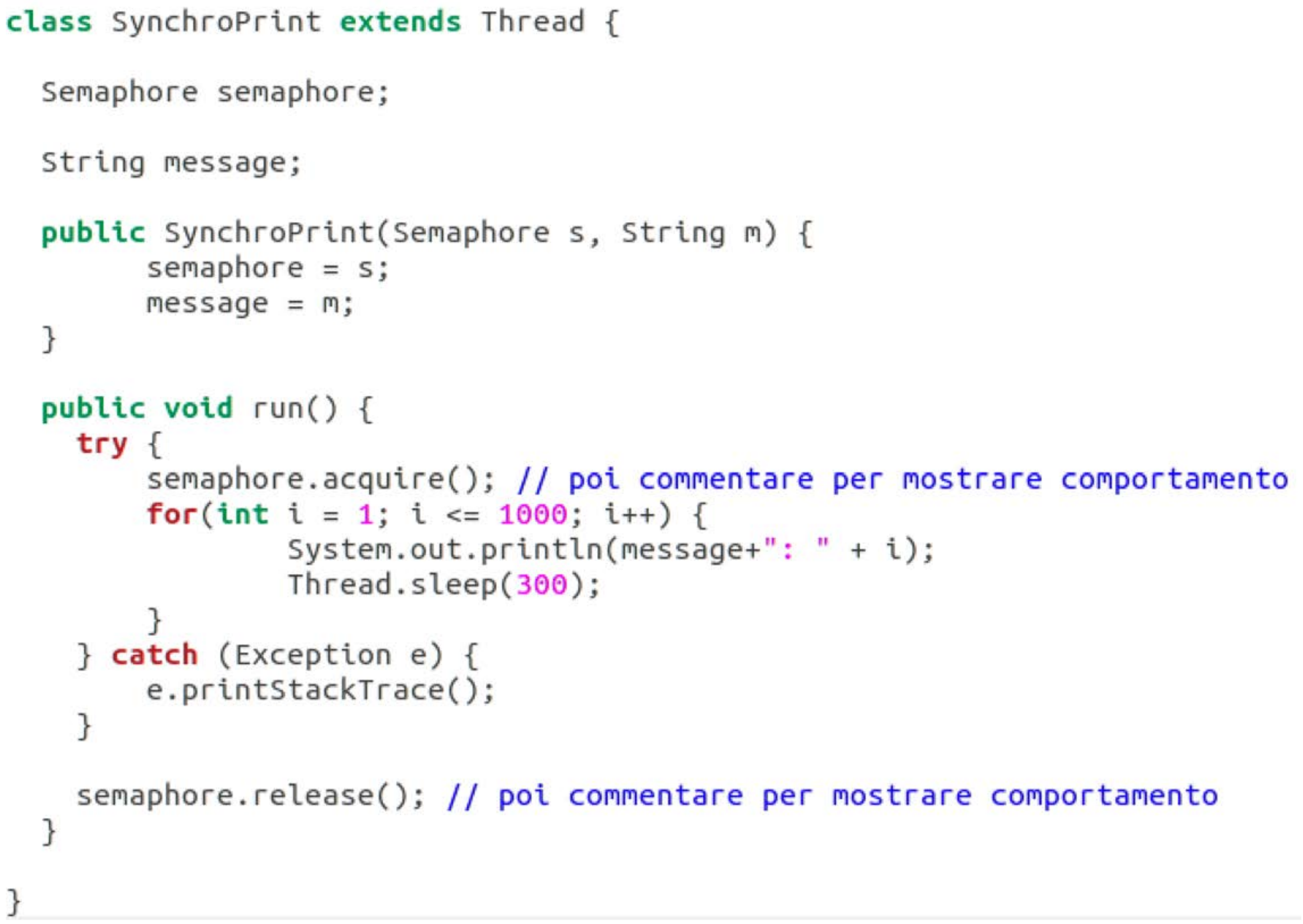
- Semaphores can be used to notify other threads that events have occurred
  - Producer-consumer relationship
    - Producer enters its critical section to produce value
    - Consumer is blocked until producer finishes
    - Consumer enters its critical section to read value
    - Producer cannot update value until it is consumed
  - Semaphores offer a clear, easy-to-implement solution to this problem

Simple Semaphore in JAVA



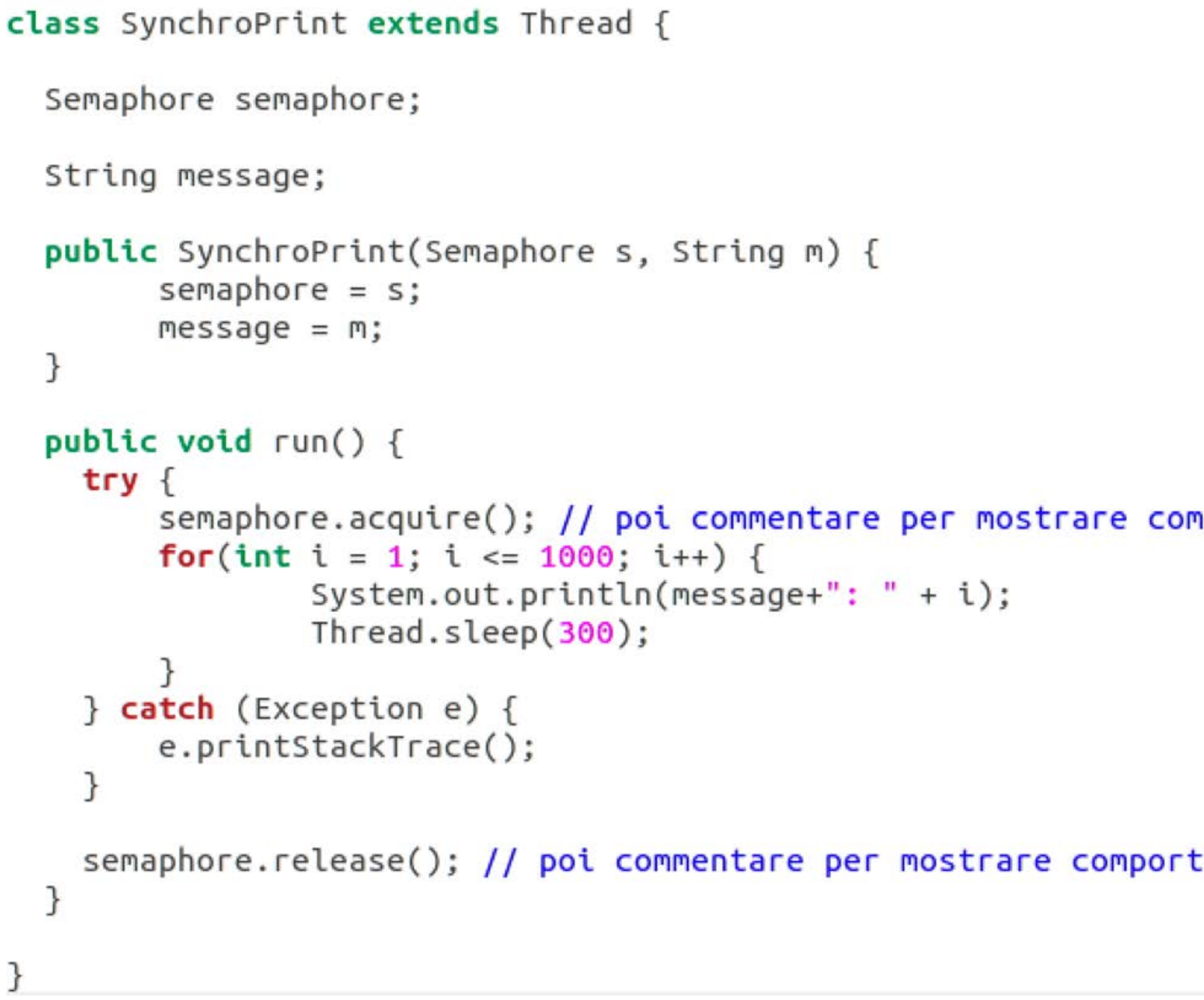
```
import java.util.concurrent.Semaphore;

public class SimpleSemaphore {

  // due thread stampano dei messaggi, vogliamo che non siano
  // interrotti fino al termine della stampa: usiamo un lock

  public static void main(String args[]) throws Exception {
        // il primo parametro è il numero di permits: 1 = binario
        // ovvero semaphoro disponibile / non disponibile

        // il secondo parametro è fairness: true/false
        // true: politica FIFO per decidere quale dei thread
        //       in attesa deve accedere non appena il lock
        //       è disponibile
        // false: la politica viene decisa dalla JVM

        Semaphore sem = new Semaphore(1,true);

        Thread thread_A = new Thread(new SynchroPrint(sem, "message from A"));
        Thread thread_B = new Thread(new SynchroPrint(sem, "message from B"));

        thread_A.start();
        thread_B.start();

        thread_A.join();
        thread_B.join();
  }

}
```

Simple Semaphore in JAVA

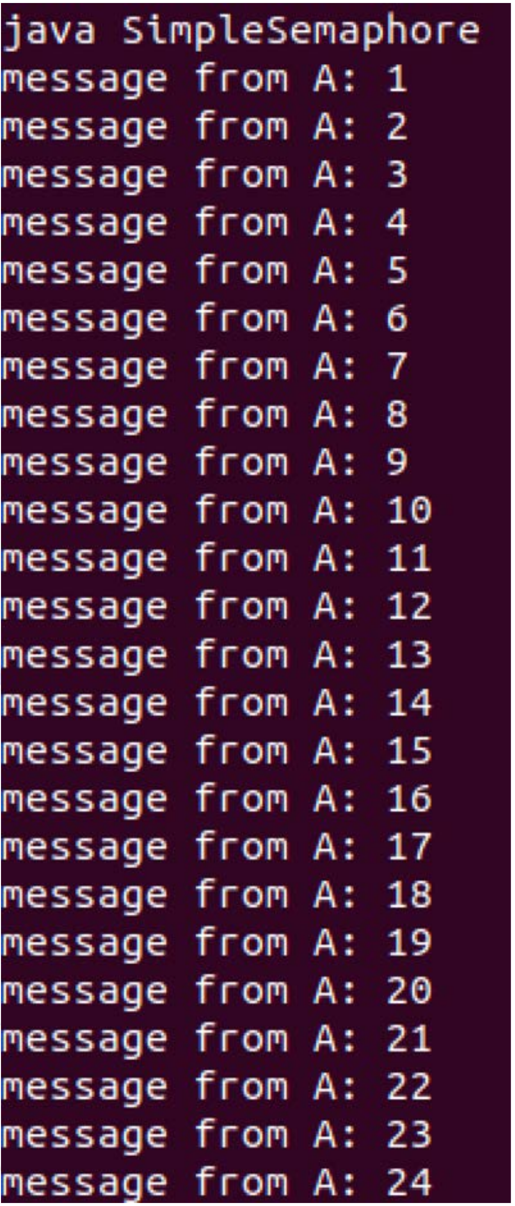


```
class SynchroPrint extends Thread {

  Semaphore semaphore;

  String message;

  public SynchroPrint(Semaphore s, String m) {
        semaphore = s;
        message = m;
  }

  public void run() {
    try {
        semaphore.acquire(); // poi commentare per mostrare comportamento
        for(int i = 1; i <= 1000; i++) {
                System.out.println(message+": " + i);
                Thread.sleep(300);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }

    semaphore.release(); // poi commentare per mostrare comportamento
  }

}
```

Simple Semaphore in JAVA



```java
class SynchroPrint extends Thread {

  Semaphore semaphore;

  String message;

  public SynchroPrint(Semaphore s, String m) {
        semaphore = s;
        message = m;
  }

  public void run() {
    try {
        semaphore.acquire(); // poi commentare per mostrare com
        for(int i = 1; i <= 1000; i++) {
                System.out.println(message+": " + i);
                Thread.sleep(300);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }

    semaphore.release(); // poi commentare per mostrare comport
  }

}
```



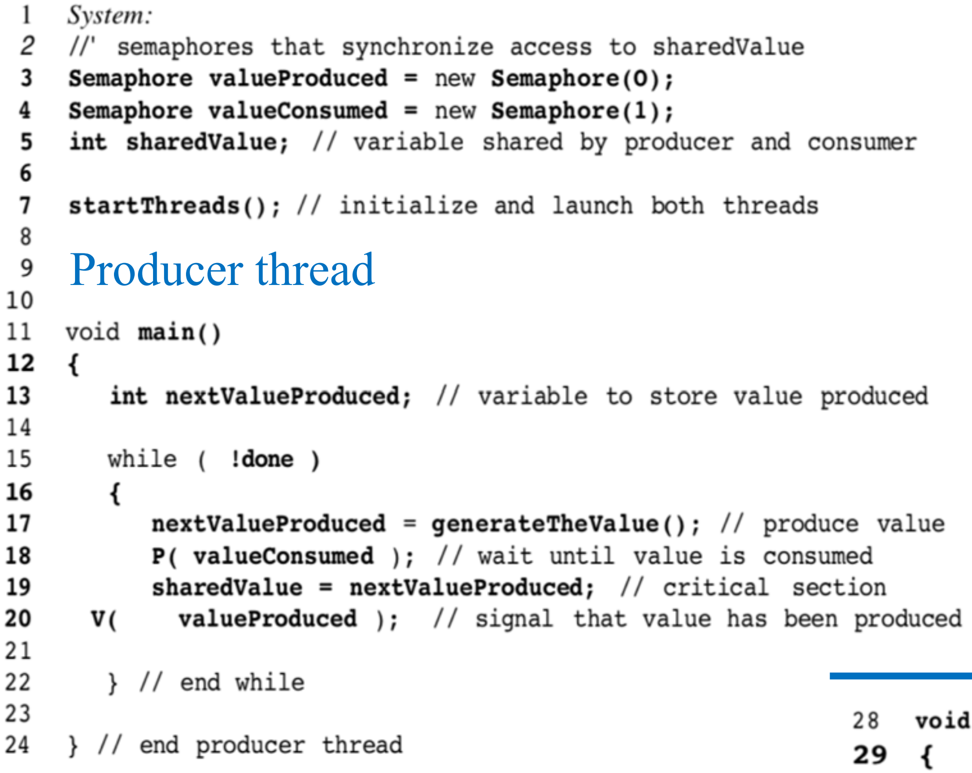
```
java SimpleSemaphore
message from A: 1
message from A: 2
message from A: 3
message from A: 4
message from A: 5
message from A: 6
message from A: 7
message from A: 8
message from A: 9
message from A: 10
message from A: 11
message from A: 12
message from A: 13
message from A: 14
message from A: 15
message from A: 16
message from A: 17
message from A: 18
message from A: 19
message from A: 20
message from A: 21
message from A: 22
message from A: 23
message from A: 24
```

Semaforo: Produttore-Consumatore

- Producer enters its critical section to produce value
- Consumer is blocked until producer finishes
- Consumer enters its critical section to read value
- Producer cannot update value until it is consumed

- Si utilizzano due semafori, uno per il produttore ed uno per il consumatore
- Ciascun semaforo protegge l’accesso alla risorsa condivisa (buffer) da parte di ciascun thread (sarebbe sbagliato implementarlo con un singolo semaforo)
- Il semaforo del Produttore protegge nel caso in cui il buffer sia pieno
- Il semaforo del Consumatore protegge nel caso in cui il buffer sia vuoto

5.6.2 Thread Synchronization with Semaphores

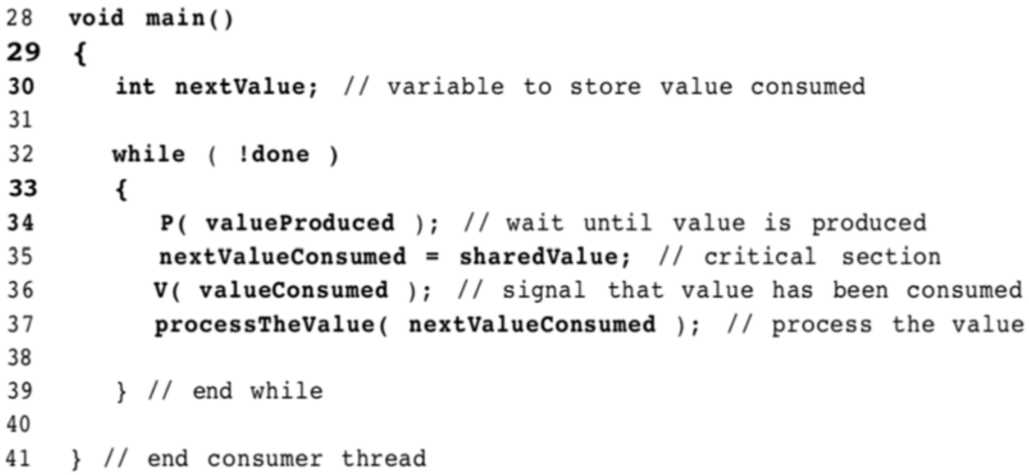


```
System:
//' semaphores that synchronize access to sharedValue
Semaphore valueProduced = new Semaphore(0);
Semaphore valueConsumed = new Semaphore(1);
int sharedValue; // variable shared by producer and consumer

startThreads(); // initialize and launch both threads

```

Producer thread

```

void main()
{
   int nextValueProduced; // variable to store value produced

   while ( !done )
   {
      nextValueProduced = generateTheValue(); // produce value
      P( valueConsumed ); // wait until value is consumed
      sharedValue = nextValueProduced; // critical section
  V(    valueProduced );  // signal that value has been produced

   } // end while

} // end producer thread
```

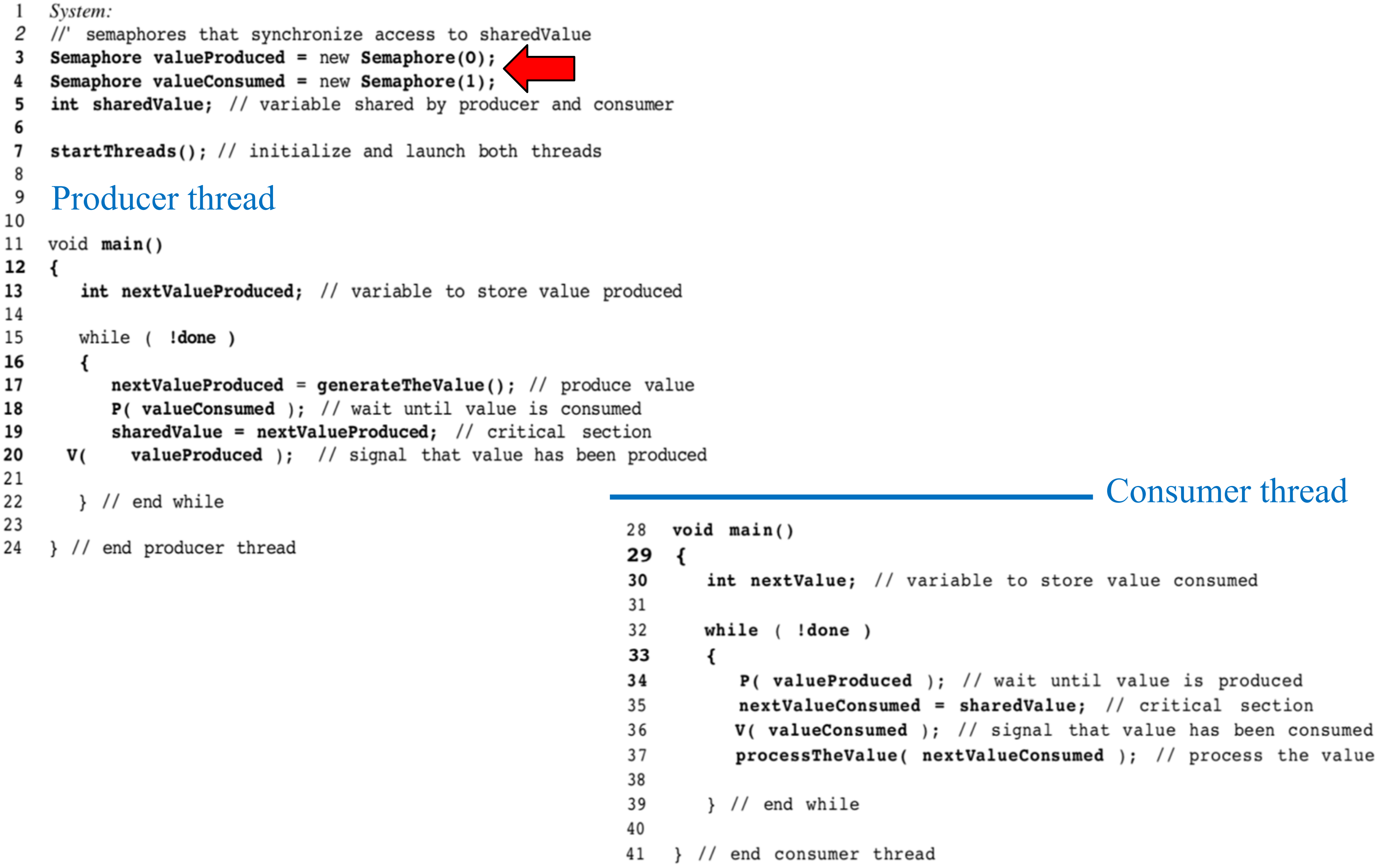
Consumer thread



```
void main()
{
   int nextValue; // variable to store value consumed

   while ( !done )
   {
      P( valueProduced ); // wait until value is produced
      nextValueConsumed = sharedValue; // critical section
     V( valueConsumed ); // signal that value has been consumed
     processTheValue( nextValueConsumed ); // process the value

   } // end while

} // end consumer thread
```

Figure 5.16 Producer/consumer relationship implemented with semaphores

5.6.2 Thread Synchronization with Semaphores



```
System:
//' semaphores that synchronize access to sharedValue
Semaphore valueProduced = new Semaphore(0);
Semaphore valueConsumed = new Semaphore(1);
int sharedValue; // variable shared by producer and consumer

startThreads(); // initialize and launch both threads
```

Producer thread

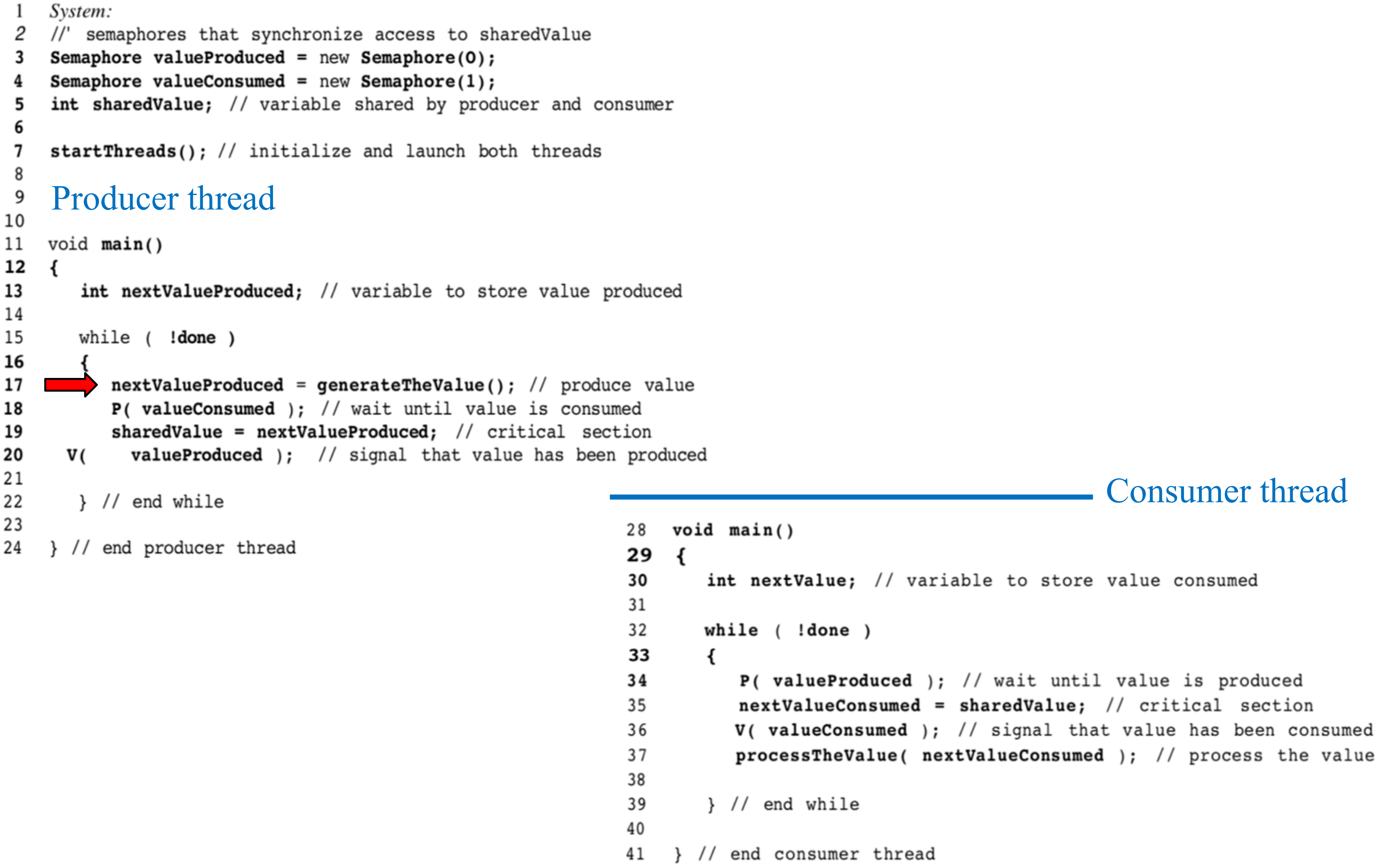
```
void main()
{
   int nextValueProduced; // variable to store value produced

   while ( !done )
   {
      nextValueProduced = generateTheValue(); // produce value
      P( valueConsumed ); // wait until value is consumed
      sharedValue = nextValueProduced; // critical section
  V(    valueProduced );  // signal that value has been produced

   } // end while

} // end producer thread
```

Consumer thread

```
void main()
{
   int nextValue; // variable to store value consumed

   while ( !done )
   {
      P( valueProduced ); // wait until value is produced
      nextValueConsumed = sharedValue; // critical section
     V( valueConsumed ); // signal that value has been consumed
     processTheValue( nextValueConsumed ); // process the value

   } // end while

} // end consumer thread
```

Figure 5.16 Producer/consumer relationship implemented with semaphores

5.6.2 Thread Synchronization with Semaphores



```
System:
//' semaphores that synchronize access to sharedValue
Semaphore valueProduced = new Semaphore(0);
Semaphore valueConsumed = new Semaphore(1);
int sharedValue; // variable shared by producer and consumer

startThreads(); // initialize and launch both threads
```

Producer thread

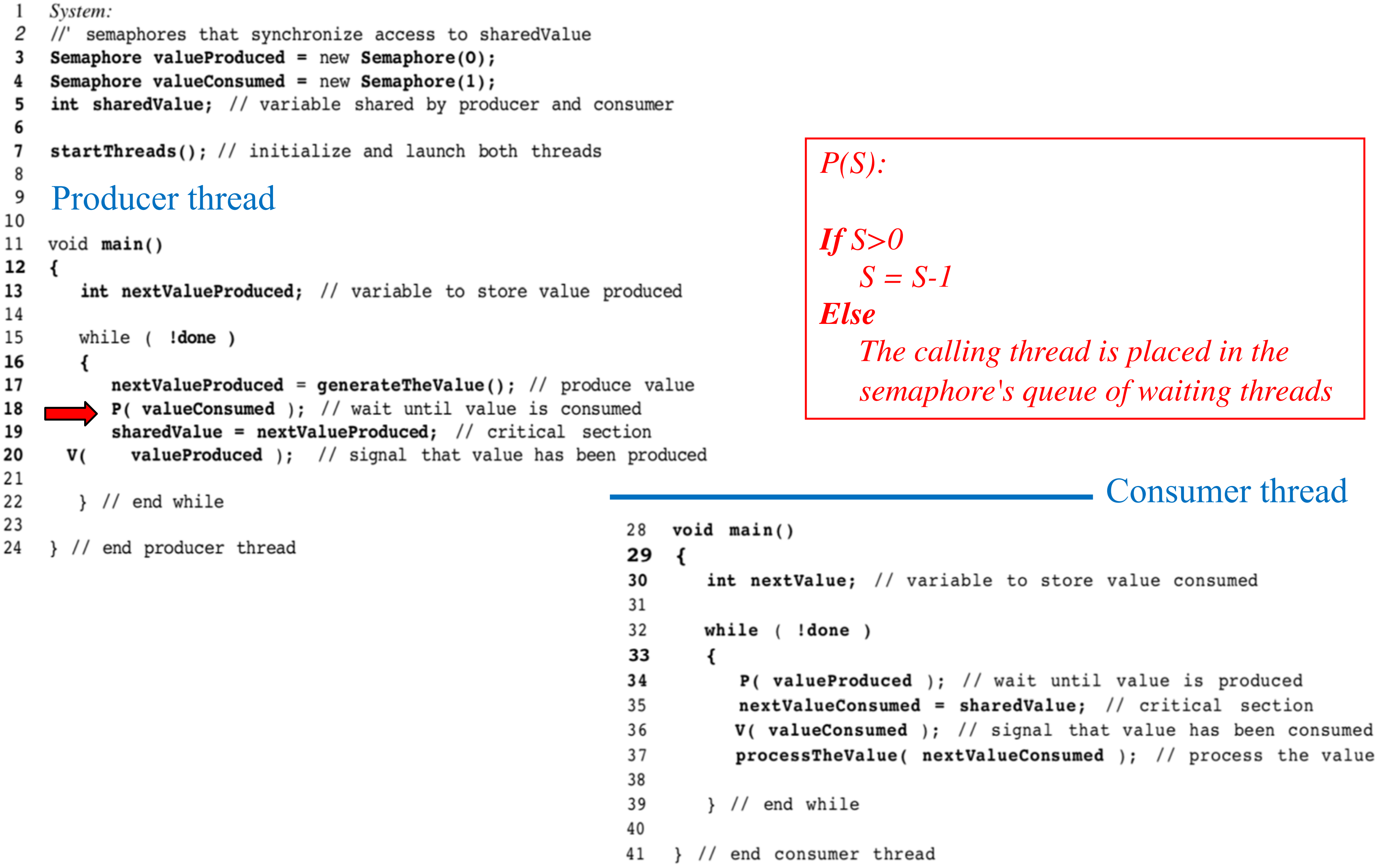
```
void main()
{
   int nextValueProduced; // variable to store value produced

   while ( !done )
   {
      nextValueProduced = generateTheValue(); // produce value
      P( valueConsumed ); // wait until value is consumed
      sharedValue = nextValueProduced; // critical section
  V(   valueProduced );  // signal that value has been produced

   } // end while

} // end producer thread
```

Consumer thread

```
void main()
{
   int nextValue; // variable to store value consumed

   while ( !done )
   {
      P( valueProduced ); // wait until value is produced
      nextValueConsumed = sharedValue; // critical section
     V( valueConsumed ); // signal that value has been consumed
     processTheValue( nextValueConsumed ); // process the value

   } // end while

} // end consumer thread
```

Figure 5.16 Producer/consumer relationship implemented with semaphores

5.6.2 Thread Synchronization with Semaphores



```
System:
//' semaphores that synchronize access to sharedValue
Semaphore valueProduced = new Semaphore(0);
Semaphore valueConsumed = new Semaphore(1);
int sharedValue; // variable shared by producer and consumer

startThreads(); // initialize and launch both threads
```

Producer thread

```
void main()
{
   int nextValueProduced; // variable to store value produced

   while ( !done )
   {
      nextValueProduced = generateTheValue(); // produce value
      P( valueConsumed ); // wait until value is consumed
      sharedValue = nextValueProduced; // critical section
 V(   valueProduced ); // signal that value has been produced

   } // end while

} // end producer thread
```

P(S):

***If** S>0*
    S = S-1
Else
    The calling thread is placed in the semaphore's queue of waiting threads

Consumer thread

```
void main()
{
   int nextValue; // variable to store value consumed

   while ( !done )
   {
      P( valueProduced ); // wait until value is produced
      nextValueConsumed = sharedValue; // critical section
     V( valueConsumed ); // signal that value has been consumed
     processTheValue( nextValueConsumed ); // process the value

   } // end while

} // end consumer thread
```

Figure 5.16 Producer/consumer relationship implemented with semaphores

5.6.2 Thread Synchronization with Semaphores

```
System:
//' semaphores that synchronize access to sharedValue
Semaphore valueProduced = new Semaphore(0);
Semaphore valueConsumed = new Semaphore(1);  → 0
int sharedValue; // variable shared by producer and consumer

startThreads(); // initialize and launch both threads
```

Producer thread

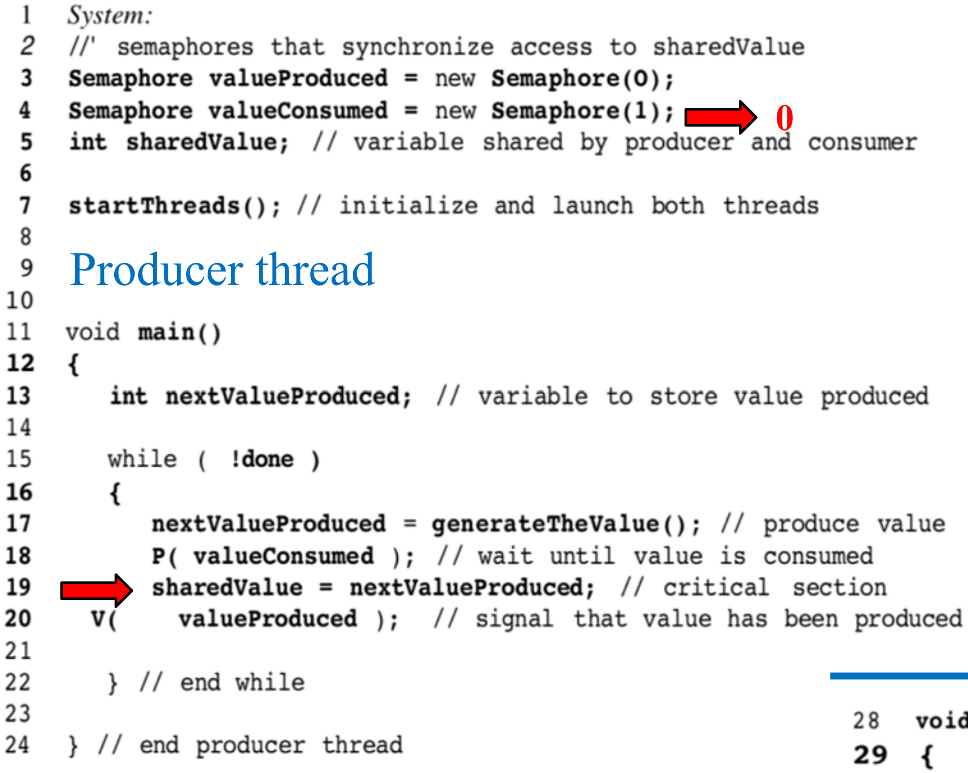
```
void main()
{
   int nextValueProduced; // variable to store value produced

   while ( !done )
   {
      nextValueProduced = generateTheValue(); // produce value
→     P( valueConsumed ); // wait until value is consumed
      sharedValue = nextValueProduced; // critical section
  V(    valueProduced );   // signal that value has been produced

   } // end while

} // end producer thread
```

Consumer thread

```
void main()
{
   int nextValue; // variable to store value consumed

   while ( !done )
   {
      P( valueProduced ); // wait until value is produced
      nextValueConsumed = sharedValue; // critical section
     V( valueConsumed ); // signal that value has been consumed
     processTheValue( nextValueConsumed ); // process the value

   } // end while

} // end consumer thread
```

Figure 5.16 Producer/consumer relationship implemented with semaphores

5.6.2 Thread Synchronization with Semaphores



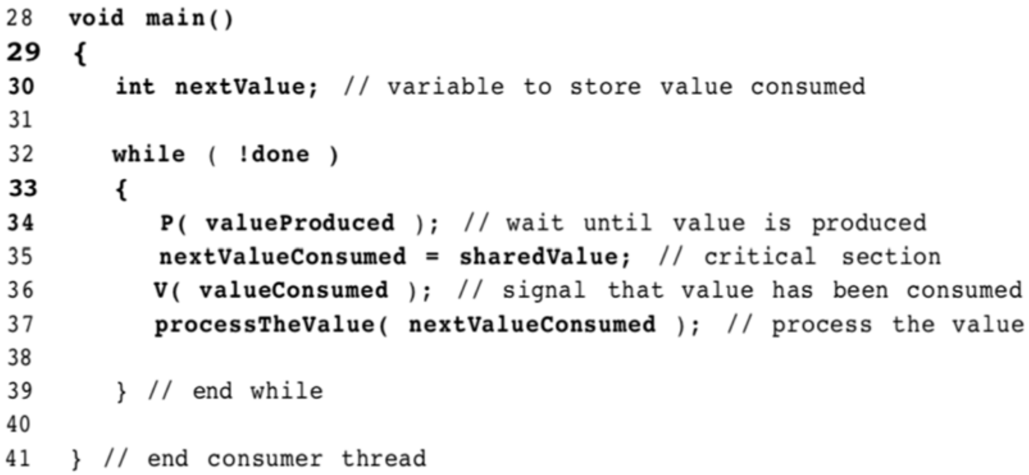
```
System:
//' semaphores that synchronize access to sharedValue
Semaphore valueProduced = new Semaphore(0);
Semaphore valueConsumed = new Semaphore(1);
int sharedValue; // variable shared by producer and consumer

startThreads(); // initialize and launch both threads
```

0

Producer thread

```
void main()
{
   int nextValueProduced; // variable to store value produced

   while ( !done )
   {
      nextValueProduced = generateTheValue(); // produce value
      P( valueConsumed ); // wait until value is consumed
      sharedValue = nextValueProduced; // critical section
  V(   valueProduced ); // signal that value has been produced

   } // end while

} // end producer thread
```

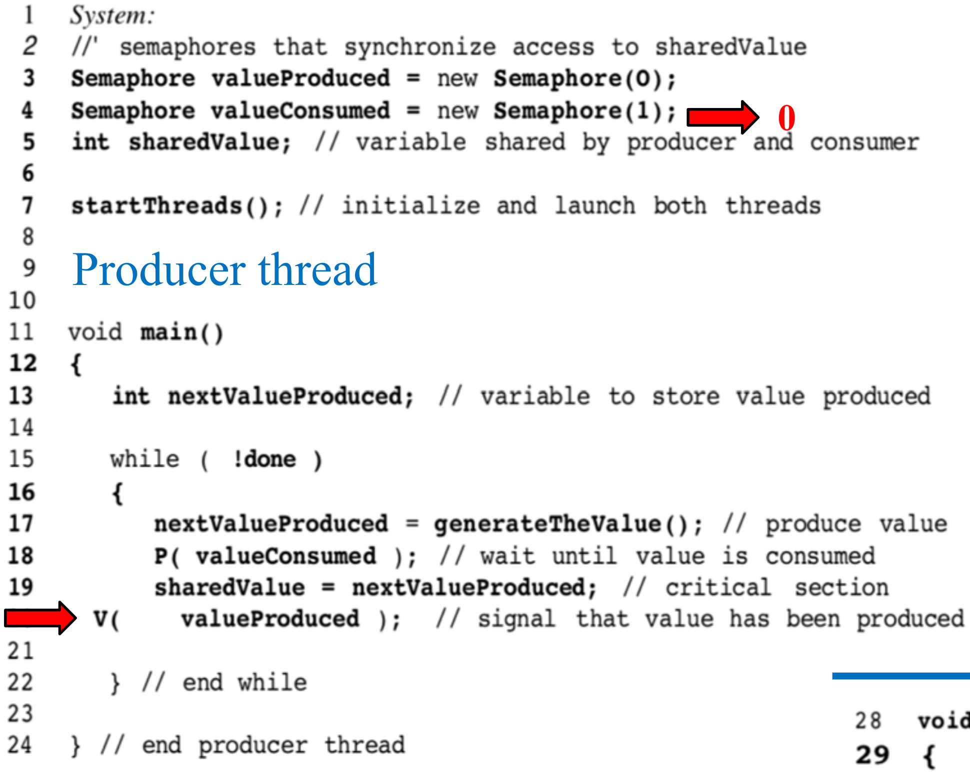
Consumer thread



```
void main()
{
   int nextValue; // variable to store value consumed

   while ( !done )
   {
      P( valueProduced ); // wait until value is produced
      nextValueConsumed = sharedValue; // critical section
     V( valueConsumed ); // signal that value has been consumed
     processTheValue( nextValueConsumed ); // process the value

   } // end while

} // end consumer thread
```

Figure 5.16 Producer/consumer relationship implemented with semaphores

5.6.2 Thread Synchronization with Semaphores

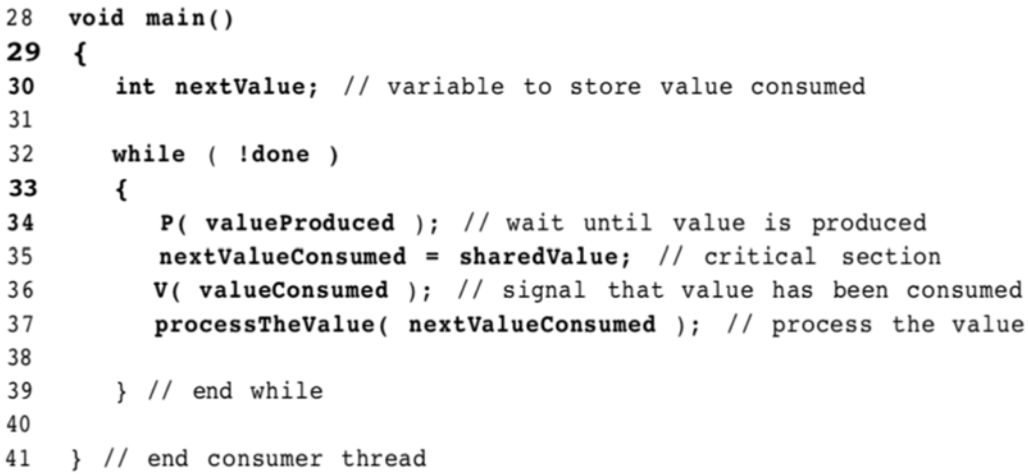


```
System:
//' semaphores that synchronize access to sharedValue
Semaphore valueProduced = new Semaphore(0);
Semaphore valueConsumed = new Semaphore(1); → 0
int sharedValue; // variable shared by producer and consumer

startThreads(); // initialize and launch both threads

Producer thread

void main()
{
   int nextValueProduced; // variable to store value produced

   while ( !done )
   {
      nextValueProduced = generateTheValue(); // produce value
      P( valueConsumed ); // wait until value is consumed
      sharedValue = nextValueProduced; // critical section
→  V(   valueProduced ); // signal that value has been produced

   } // end while

} // end producer thread
```

V(S):

If *any threads are waiting on S*
Resume the "next" waiting thread in the semaphore's queue
Else
S = S+1

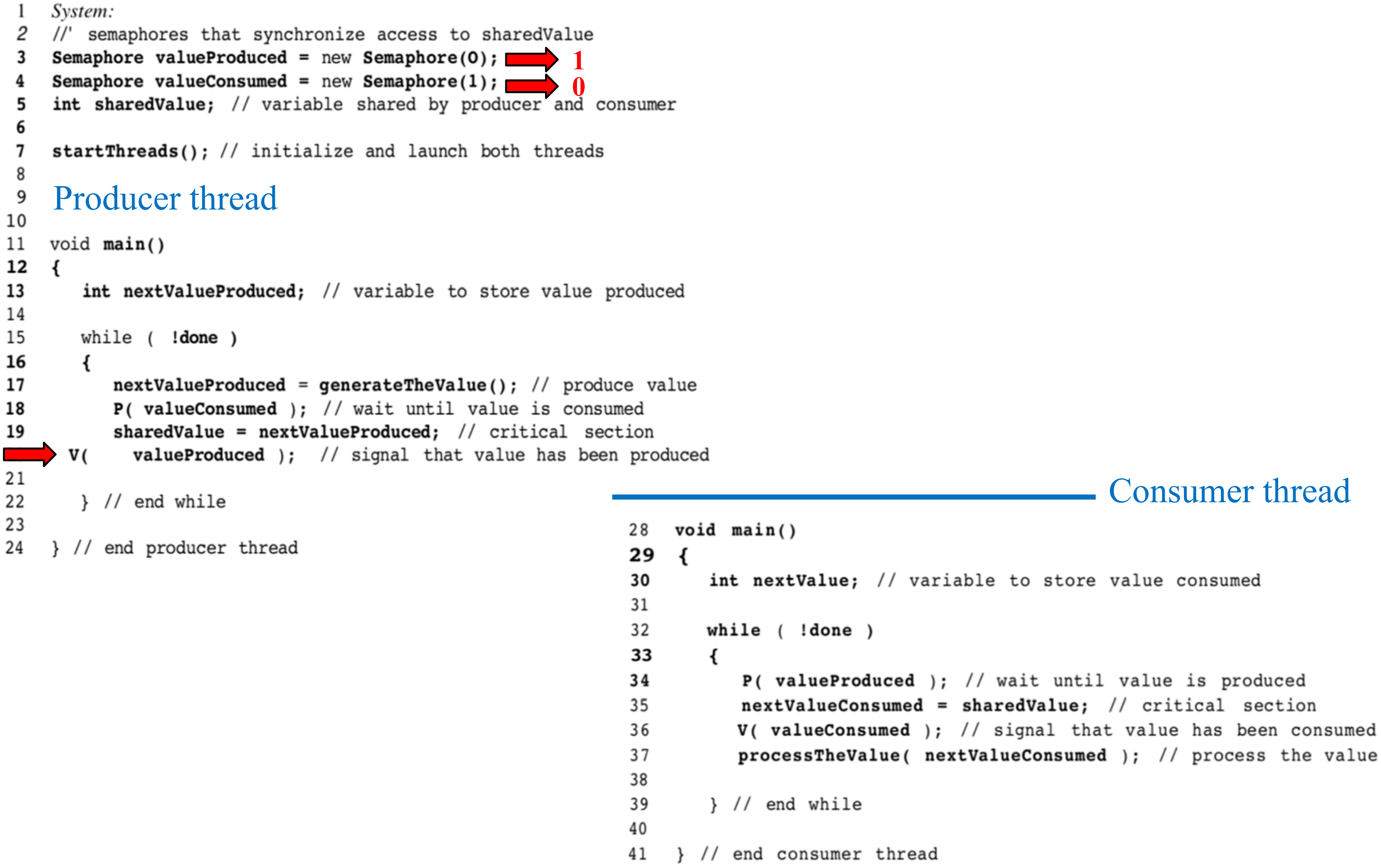
Consumer thread



```
void main()
{
   int nextValue; // variable to store value consumed

   while ( !done )
   {
      P( valueProduced ); // wait until value is produced
      nextValueConsumed = sharedValue; // critical section
     V( valueConsumed ); // signal that value has been consumed
     processTheValue( nextValueConsumed ); // process the value

   } // end while

} // end consumer thread
```

Figure 5.16 Producer/consumer relationship implemented with semaphores

5.6.2 Thread Synchronization with Semaphores



System:

```
//' semaphores that synchronize access to sharedValue
Semaphore valueProduced = new Semaphore(0);
Semaphore valueConsumed = new Semaphore(1);
int sharedValue; // variable shared by producer and consumer

startThreads(); // initialize and launch both threads
```

Producer thread

```
void main()
{
   int nextValueProduced; // variable to store value produced

   while ( !done )
   {
      nextValueProduced = generateTheValue(); // produce value
      P( valueConsumed ); // wait until value is consumed
      sharedValue = nextValueProduced; // critical section
V(    valueProduced ); // signal that value has been produced

   } // end while

} // end producer thread
```

Consumer thread

```
void main()
{
   int nextValue; // variable to store value consumed

   while ( !done )
   {
      P( valueProduced ); // wait until value is produced
      nextValueConsumed = sharedValue; // critical section
     V( valueConsumed ); // signal that value has been consumed
     processTheValue( nextValueConsumed ); // process the value

   } // end while

} // end consumer thread
```

Figure 5.16 Producer/consumer relationship implemented with semaphores

5.6.2 Thread Synchronization with Semaphores

```
System:
//'  semaphores that synchronize access to sharedValue
Semaphore valueProduced = new Semaphore(0); ➡ 1
Semaphore valueConsumed = new Semaphore(1); ➡ 0
int sharedValue; // variable shared by producer and consumer

startThreads(); // initialize and launch both threads
```

Producer thread

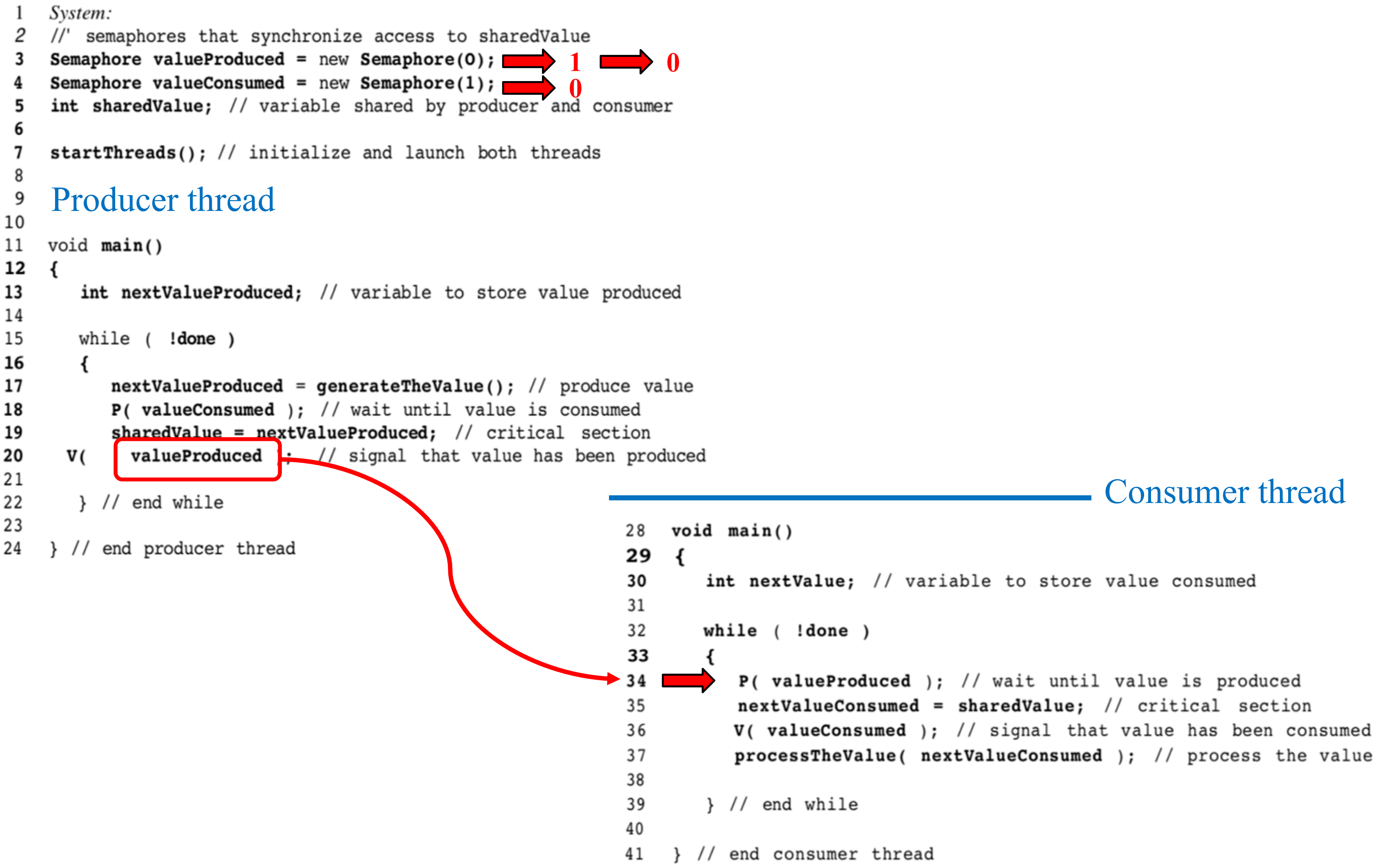
```
void main()
{
    int nextValueProduced; // variable to store value produced

    while ( !done )
    {
        nextValueProduced = generateTheValue(); // produce value
        P( valueConsumed ); // wait until value is consumed
        sharedValue = nextValueProduced; // critical section
      V(   valueProduced  ); // signal that value has been produced

    } // end while

} // end producer thread
```

Consumer thread

```
void main()
{
    int nextValue; // variable to store value consumed

    while ( !done )
    {
        P( valueProduced ); // wait until value is produced
        nextValueConsumed = sharedValue; // critical section
       V( valueConsumed ); // signal that value has been consumed
       processTheValue( nextValueConsumed ); // process the value

    } // end while

} // end consumer thread
```

Figure 5.16 Producer/consumer relationship implemented with semaphores

5.6.2 Thread Synchronization with Semaphores



```
System:
//' semaphores that synchronize access to sharedValue
Semaphore valueProduced = new Semaphore(0);
Semaphore valueConsumed = new Semaphore(1);
int sharedValue; // variable shared by producer and consumer

startThreads(); // initialize and launch both threads
```

Producer thread

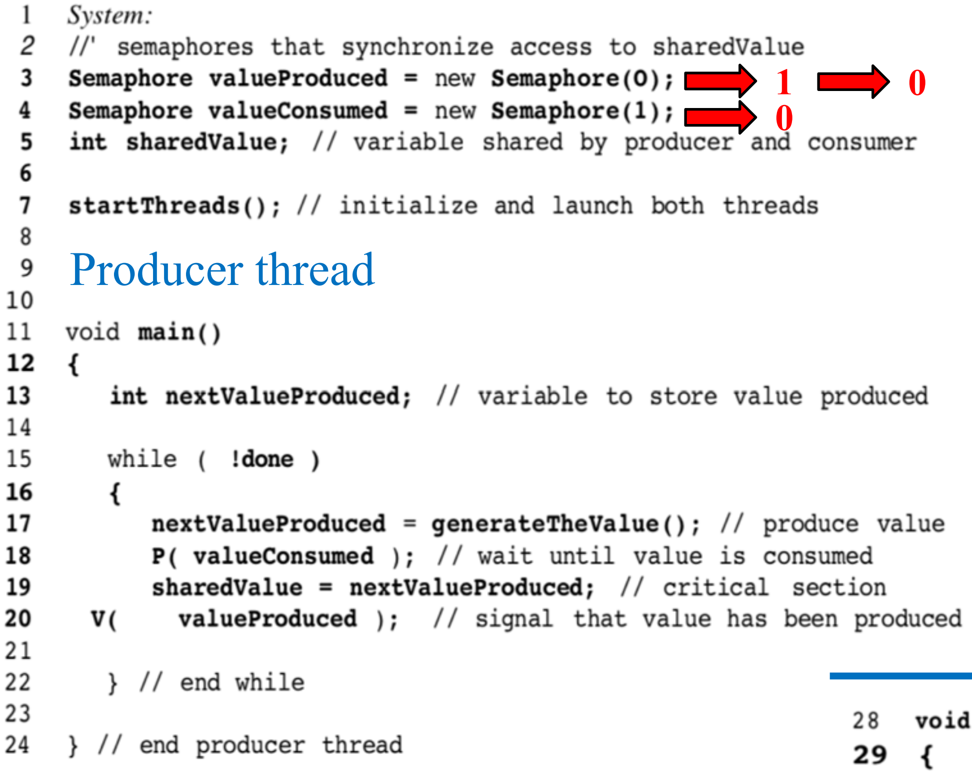
```
void main()
{
   int nextValueProduced; // variable to store value produced

   while ( !done )
   {
      nextValueProduced = generateTheValue(); // produce value
      P( valueConsumed ); // wait until value is consumed
      sharedValue = nextValueProduced; // critical section
  V(   valueProduced ); // signal that value has been produced

   } // end while

} // end producer thread
```

Consumer thread

```
void main()
{
   int nextValue; // variable to store value consumed

   while ( !done )
   {
      P( valueProduced ); // wait until value is produced
      nextValueConsumed = sharedValue; // critical section
     V( valueConsumed ); // signal that value has been consumed
     processTheValue( nextValueConsumed ); // process the value

   } // end while

} // end consumer thread
```

Figure 5.16 Producer/consumer relationship implemented with semaphores

5.6.2 Thread Synchronization with Semaphores

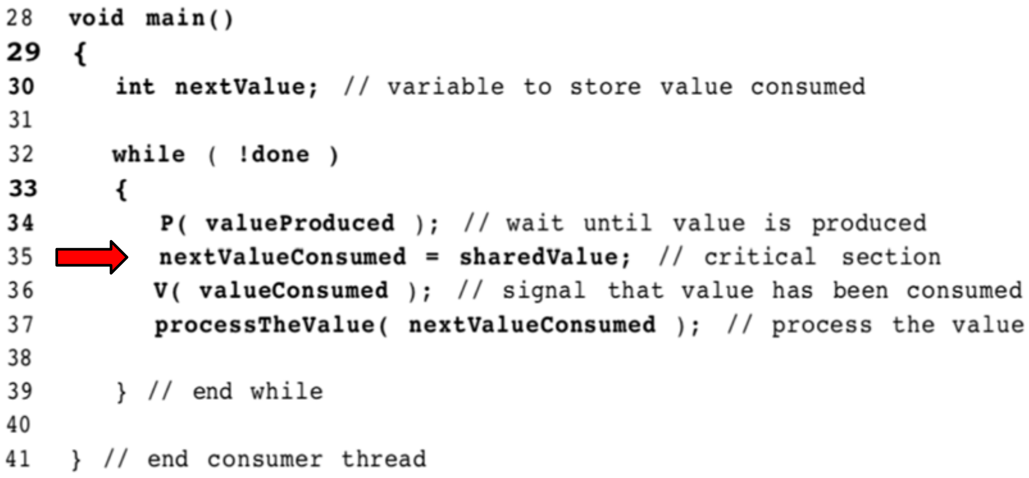


```
System:
//' semaphores that synchronize access to sharedValue
Semaphore valueProduced = new Semaphore(0);  → 1 → 0
Semaphore valueConsumed = new Semaphore(1);  → 0
int sharedValue; // variable shared by producer and consumer

startThreads(); // initialize and launch both threads
```

Producer thread

```
void main()
{
   int nextValueProduced; // variable to store value produced

   while ( !done )
   {
      nextValueProduced = generateTheValue(); // produce value
      P( valueConsumed ); // wait until value is consumed
      sharedValue = nextValueProduced; // critical section
  V(   valueProduced );  // signal that value has been produced

   } // end while

} // end producer thread
```

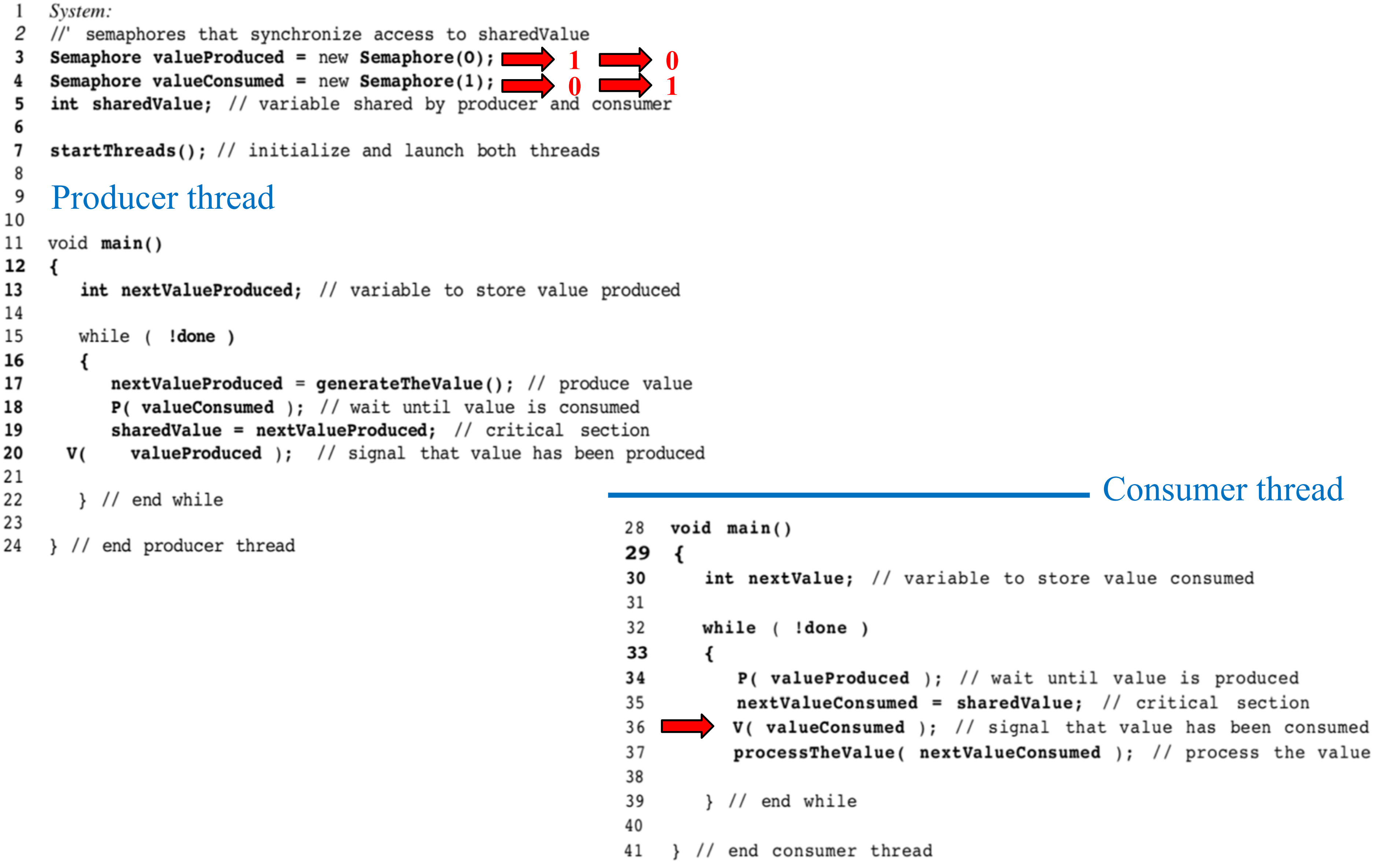
Consumer thread



```
void main()
{
   int nextValue; // variable to store value consumed

   while ( !done )
   {
      P( valueProduced ); // wait until value is produced
  →   nextValueConsumed = sharedValue; // critical section
     V( valueConsumed ); // signal that value has been consumed
     processTheValue( nextValueConsumed ); // process the value

   } // end while

} // end consumer thread
```

Figure 5.16 Producer/consumer relationship implemented with semaphores

5.6.2 Thread Synchronization with Semaphores



```
System:
//' semaphores that synchronize access to sharedValue
Semaphore valueProduced = new Semaphore(0); → 1 → 0
Semaphore valueConsumed = new Semaphore(1); → 0 → 1
int sharedValue; // variable shared by producer and consumer

startThreads(); // initialize and launch both threads
```

Producer thread

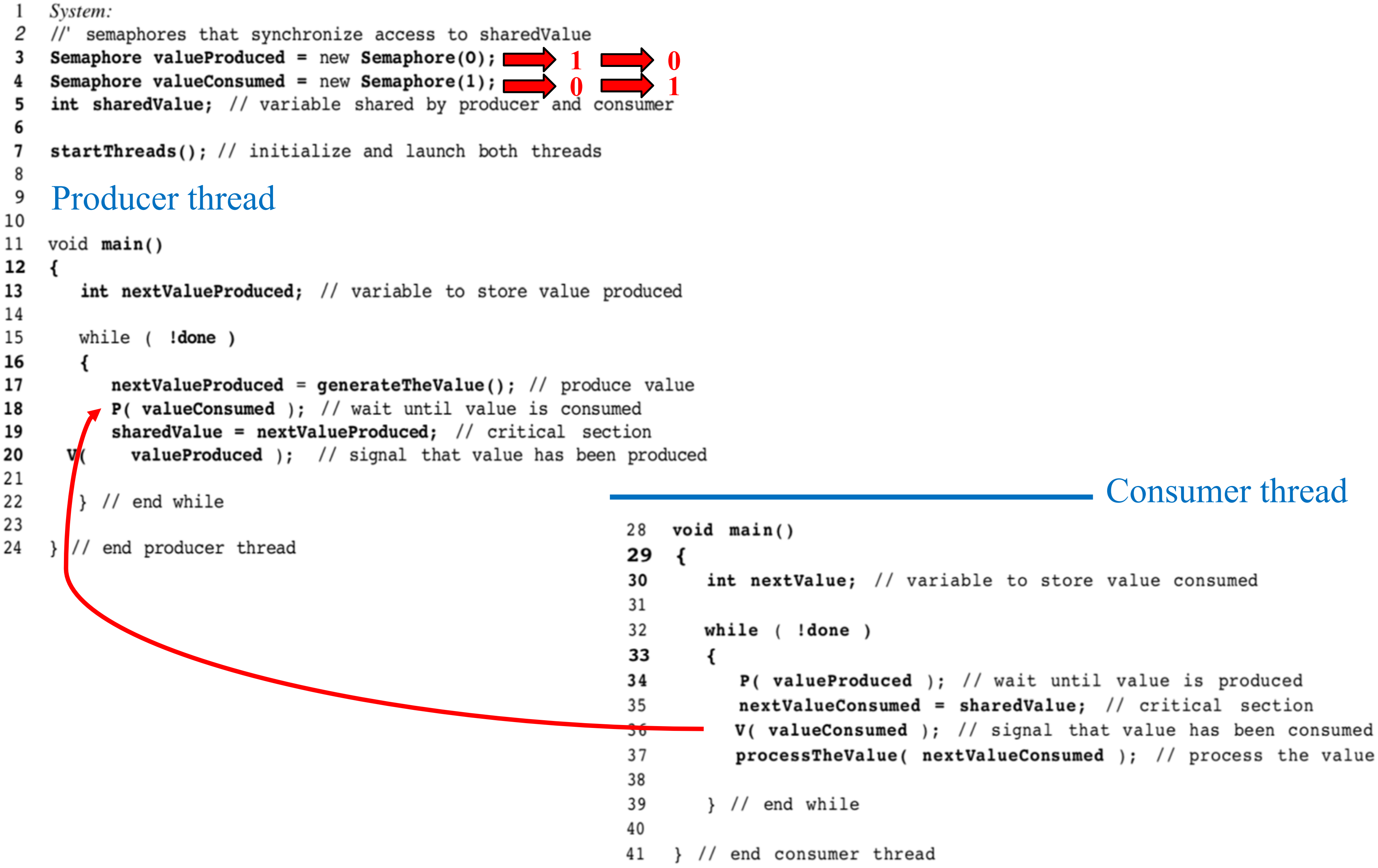
```
void main()
{
   int nextValueProduced; // variable to store value produced

   while ( !done )
   {
      nextValueProduced = generateTheValue(); // produce value
      P( valueConsumed ); // wait until value is consumed
      sharedValue = nextValueProduced; // critical section
 V(   valueProduced );  // signal that value has been produced

   } // end while

} // end producer thread
```

Consumer thread

```
void main()
{
   int nextValue; // variable to store value consumed

   while ( !done )
   {
      P( valueProduced ); // wait until value is produced
      nextValueConsumed = sharedValue; // critical section
  →   V( valueConsumed ); // signal that value has been consumed
      processTheValue( nextValueConsumed ); // process the value

   } // end while

} // end consumer thread
```

Figure 5.16 Producer/consumer relationship implemented with semaphores

5.6.2 Thread Synchronization with Semaphores



```
System:
//'  semaphores that synchronize access to sharedValue
Semaphore valueProduced = new Semaphore(0);
Semaphore valueConsumed = new Semaphore(1);
int sharedValue; // variable shared by producer and consumer

startThreads(); // initialize and launch both threads
```

Producer thread

```
void main()
{
   int nextValueProduced; // variable to store value produced

   while ( !done )
   {
      nextValueProduced = generateTheValue(); // produce value
      P( valueConsumed ); // wait until value is consumed
      sharedValue = nextValueProduced; // critical section
  V(    valueProduced );  // signal that value has been produced

   } // end while

} // end producer thread
```

Consumer thread

```
void main()
{
   int nextValue; // variable to store value consumed

   while ( !done )
   {
      P( valueProduced ); // wait until value is produced
      nextValueConsumed = sharedValue; // critical section
     V( valueConsumed ); // signal that value has been consumed
     processTheValue( nextValueConsumed ); // process the value

   } // end while

} // end consumer thread
```

Figure 5.16 Producer/consumer relationship implemented with semaphores

5.6.3 Counting Semaphores

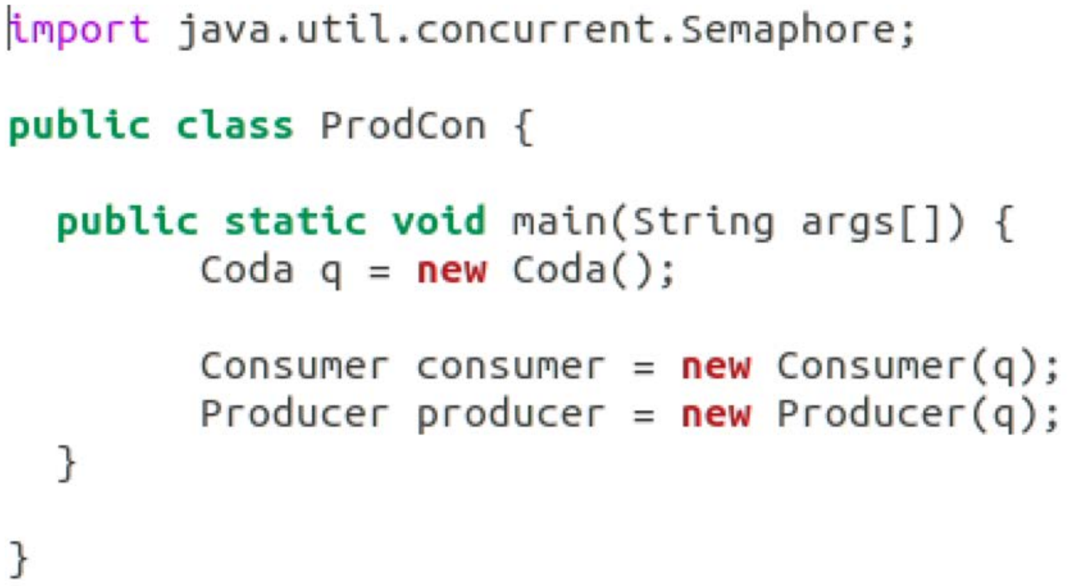
- Counting semaphores
  - Initialized with **values greater than one**
  - Can be used to control access to a pool of identical resources
    - Decrement the semaphore’s counter when taking resource from pool
    - Increment the semaphore’s counter when returning it to pool
    - If no resources are available, thread is blocked until a resource becomes available

5.6.4 Implementing Semaphores

- Semaphores can be implemented at application or kernel level
  - Application level: typically implemented by busy waiting
    - Inefficient
  - Kernel implementations can avoid busy waiting
    - Block waiting threads until they are ready
  - Kernel implementations can disable interrupts
    - Guarantee exclusive semaphore access
    - Must be careful to avoid poor performance and deadlock
    - Implementations for multiprocessor systems must use a more sophisticated approach

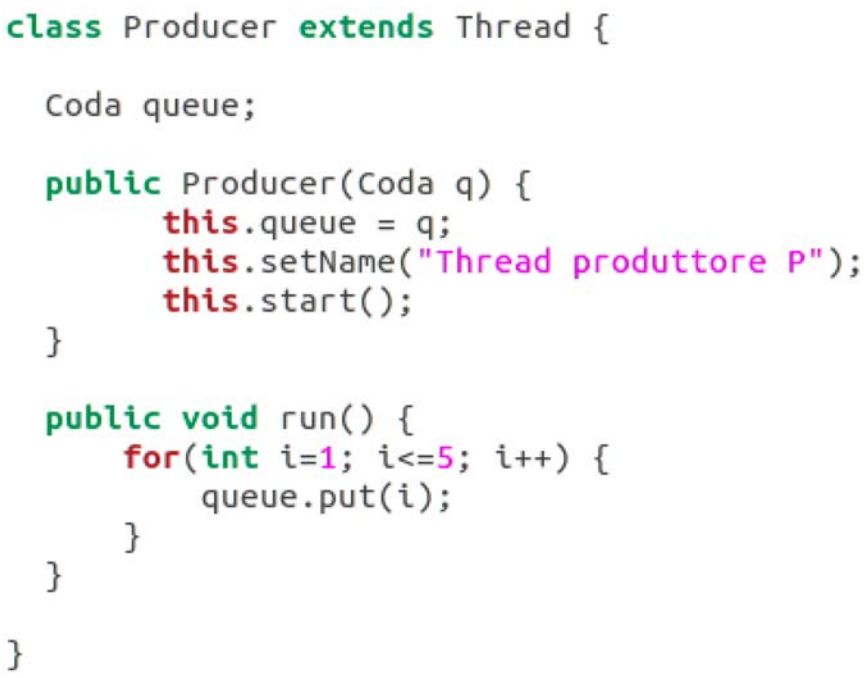
JAVA Semaphore: Producer - Consumer

- Si utilizzano due semafori, uno per il produttore ed uno per il consumatore
- Ciascun semaforo protegge l'accesso alla risorsa condivisa (buffer) da parte di ciascun thread (sarebbe sbagliato implementarlo con un singolo semaforo)
- Il semaforo del Produttore protegge nel caso in cui il buffer sia pieno
- Il semaforo del Consumatore protegge nel caso in cui il buffer sia vuoto

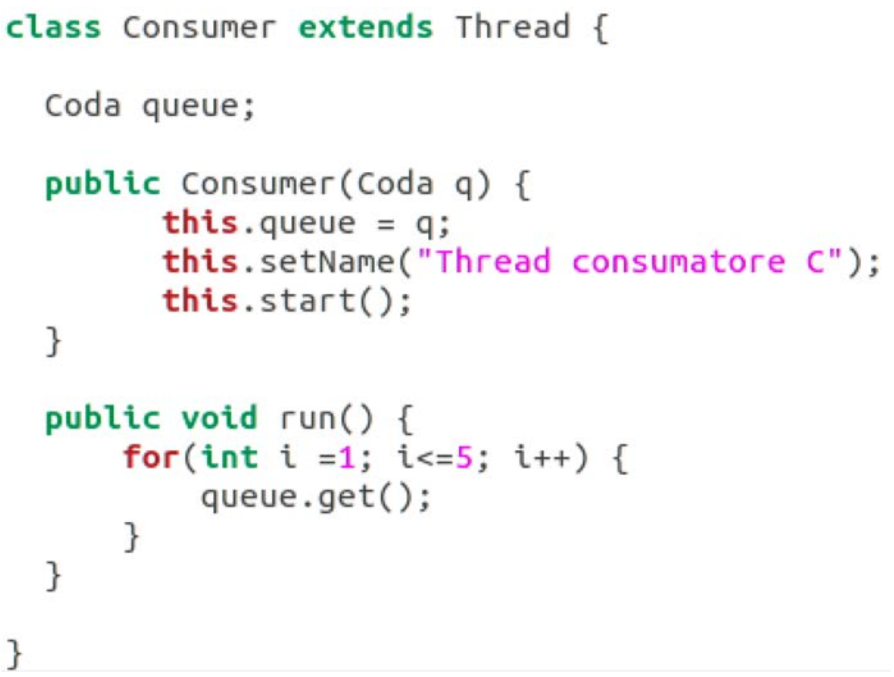


```java
import java.util.concurrent.Semaphore;

public class ProdCon {

  public static void main(String args[]) {
        Coda q = new Coda();

        Consumer consumer = new Consumer(q);
        Producer producer = new Producer(q);
  }

}
```

JAVA Semaphore: Producer - Consumer

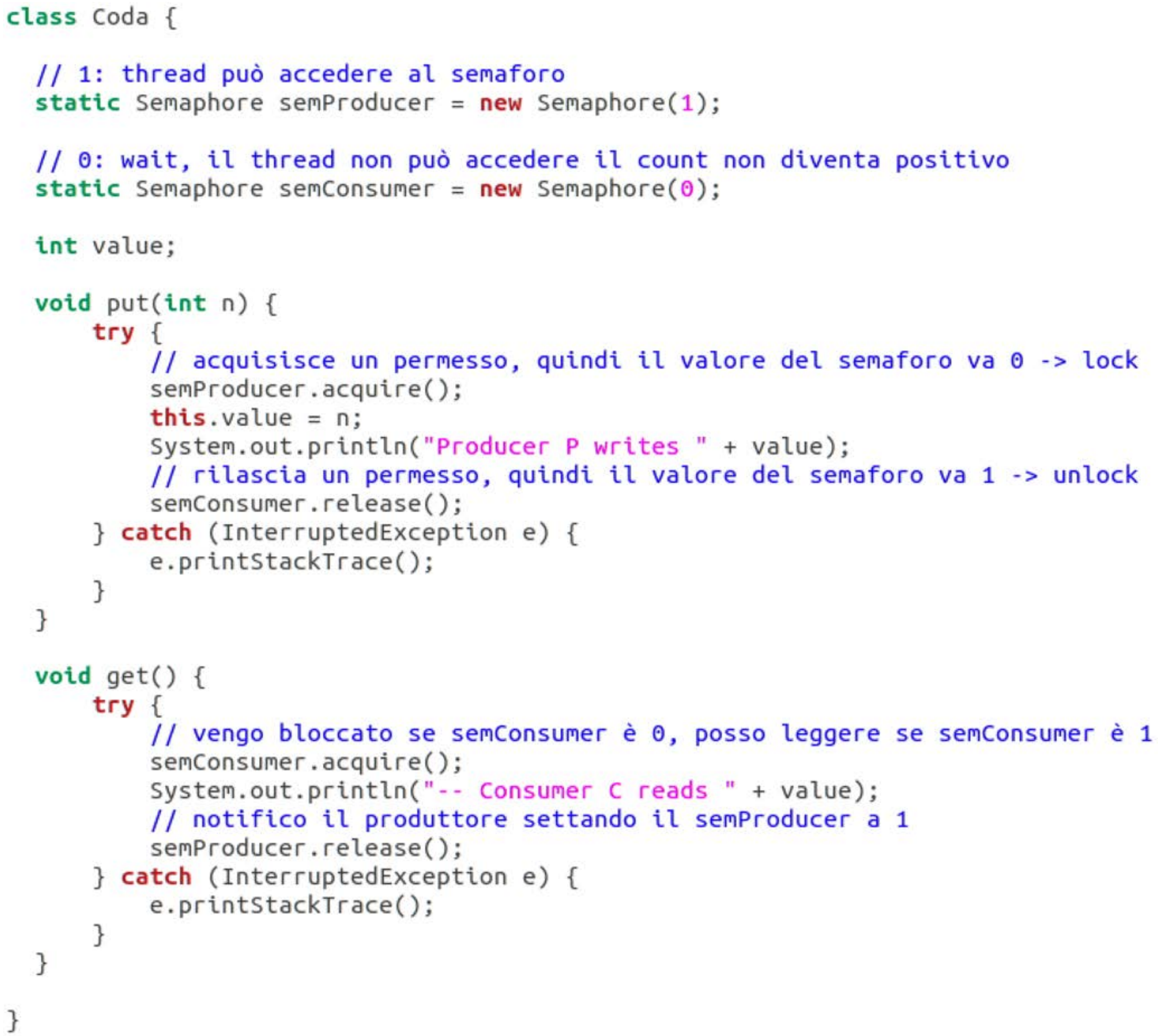


```
class Producer extends Thread {

  Coda queue;

  public Producer(Coda q) {
        this.queue = q;
        this.setName("Thread produttore P");
        this.start();
  }

  public void run() {
      for(int i=1; i<=5; i++) {
          queue.put(i);
      }
  }

}
```



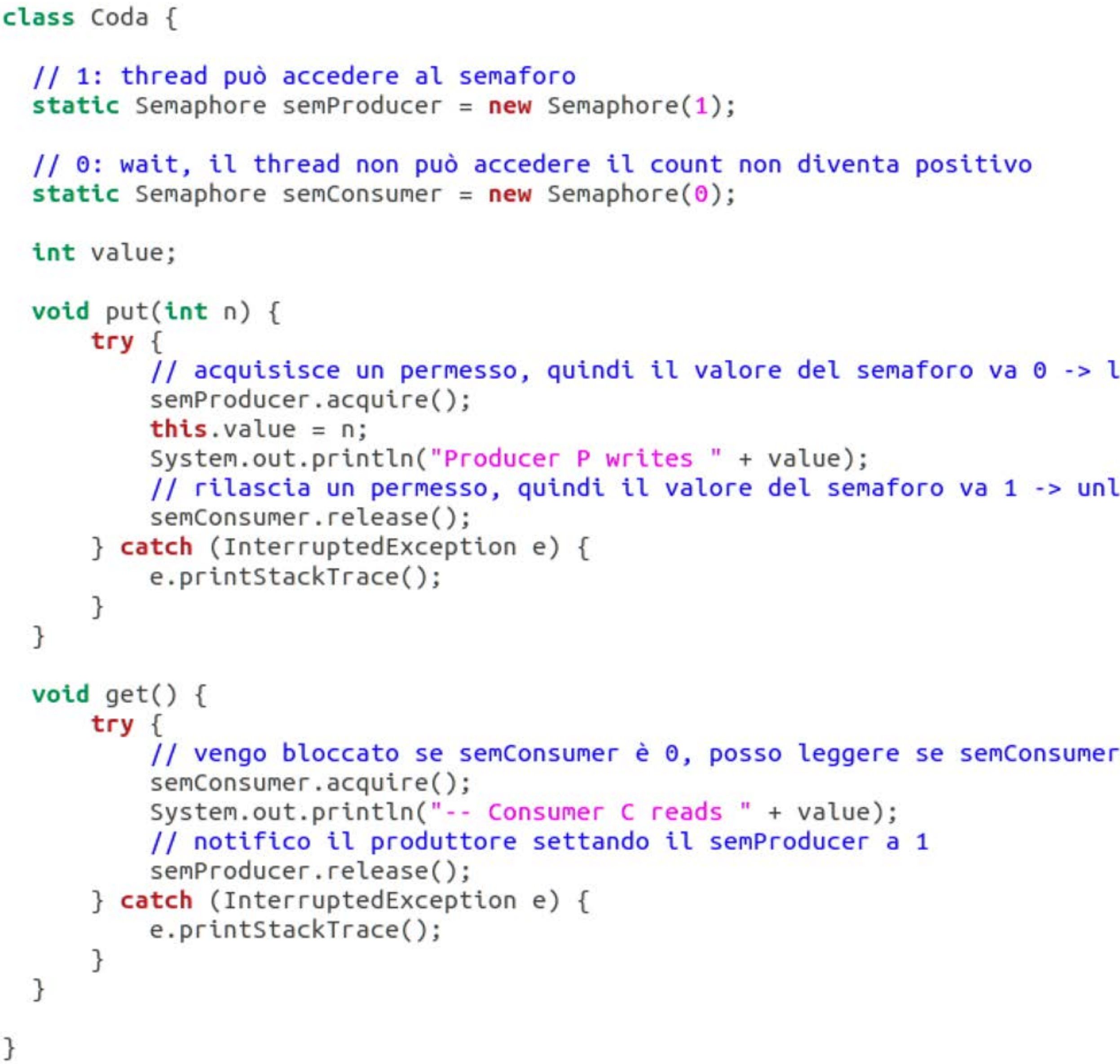
```
class Consumer extends Thread {

  Coda queue;

  public Consumer(Coda q) {
        this.queue = q;
        this.setName("Thread consumatore C");
        this.start();
  }

  public void run() {
      for(int i =1; i<=5; i++) {
          queue.get();
      }
  }

}
```

JAVA Semaphore: Producer - Consumer

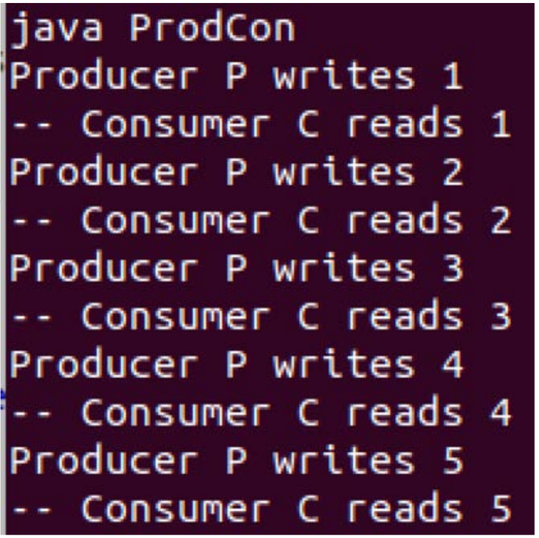


```
class Coda {

 // 1: thread può accedere al semaforo
 static Semaphore semProducer = new Semaphore(1);

 // 0: wait, il thread non può accedere il count non diventa positivo
 static Semaphore semConsumer = new Semaphore(0);

 int value;

 void put(int n) {
     try {
         // acquisisce un permesso, quindi il valore del semaforo va 0 -> lock
         semProducer.acquire();
         this.value = n;
         System.out.println("Producer P writes " + value);
         // rilascia un permesso, quindi il valore del semaforo va 1 -> unlock
         semConsumer.release();
     } catch (InterruptedException e) {
         e.printStackTrace();
     }
 }

 void get() {
     try {
         // vengo bloccato se semConsumer è 0, posso leggere se semConsumer è 1
         semConsumer.acquire();
         System.out.println("-- Consumer C reads " + value);
         // notifico il produttore settando il semProducer a 1
         semProducer.release();
     } catch (InterruptedException e) {
         e.printStackTrace();
     }
 }

}
```

JAVA Semaphore: Producer - Consumer



```
class Coda {

  // 1: thread può accedere al semaforo
  static Semaphore semProducer = new Semaphore(1);

  // 0: wait, il thread non può accedere il count non diventa positivo
  static Semaphore semConsumer = new Semaphore(0);

  int value;

  void put(int n) {
      try {
          // acquisisce un permesso, quindi il valore del semaforo va 0 -> lock
          semProducer.acquire();
          this.value = n;
          System.out.println("Producer P writes " + value);
          // rilascia un permesso, quindi il valore del semaforo va 1 -> unlock
          semConsumer.release();
      } catch (InterruptedException e) {
          e.printStackTrace();
      }
  }

  void get() {
      try {
          // vengo bloccato se semConsumer è 0, posso leggere se semConsumer
          semConsumer.acquire();
          System.out.println("-- Consumer C reads " + value);
          // notifico il produttore settando il semProducer a 1
          semProducer.release();
      } catch (InterruptedException e) {
          e.printStackTrace();
      }
  }

}
```



```
java ProdCon
Producer P writes 1
-- Consumer C reads 1
Producer P writes 2
-- Consumer C reads 2
Producer P writes 3
-- Consumer C reads 3
Producer P writes 4
-- Consumer C reads 4
Producer P writes 5
-- Consumer C reads 5
```