```
protocolo de entrada con espera ocupada;
Sección Crítica (código signal);
protocolo de salida;
```

4.4.2. Implementación de semáforos como variables enteras no negativas

En este caso, la estructura que representa al semáforo será la misma, pero ahora se modifica la implementación de las operaciones:

```
type semaphore=record of
          valor:integer;
          L:lista de procesos;
          end;

procedure initial(var s:semaphore;s0:integer);
begin
   s.valor:=s0;
   inicializar(s.L);
end;

procedure wait(var s:semaphore);
begin
   if s.valor>0 then s.valor:=s.valor-1;
   else begin
      insertar proceso en s.L;
      bloquearlo;
   end;
end;

procedure signal(var s:semaphore);
begin
   if not vacia(L) then
   begin
      eliminar un proceso de s.L;
      desbloquearlo
   end
   else s.valor:=s.valor+1;
end;
```

4.5. Sincronización en Java

Java no tiene semáforos como primitivas de sincronización. Sin embargo, proporciona otras primitivas diferentes con las cuales sí que se puede implementar el comportamiento de los semáforos. De esta forma, cualquier problema que pueda ser solucionado con semáforos también podrá ser solucionado con las primitivas propias de Java.

En este apartado, mostraremos cómo solucionar los problemas de exclusión mutua y condición de sincronización en Java. Para ello veremos algunos métodos más del API de Java para tratar threads. Finalizaremos con los ejemplos del productor/consumidor, la comida de filósofos, el problema de los lectores y escritores y la simulación de semáforos generales y binarios haciendo uso de las primitivas de Java.

4.5.1. Exclusión mutua en Java

Por defecto en Java un objeto no está protegido. Esto quiere decir que cualquier número de hilos puede estar ejecutando código dentro del objeto. La exclusión mutua en Java se consigue mediante la palabra reservada `synchronized`. Esta palabra puede aplicarse tanto a métodos enteros como a bloques de código dentro de un método.

Synchronized como modificador de método

Un método en Java puede llevar el modificador `synchronized`. Todos los métodos que lleven ese modificador se ejecutarán en exclusión mutua. Cuando un método sincronizado se está ejecutando, se garantiza que ningún otro método sincronizado podrá ejecutarse. Sin embargo, cualquier número de métodos no sincronizados puede estar ejecutándose dentro del objeto. En la Figura 4.3 puede observarse cómo sólo puede haber un thread ejecutando el método 2 ó 4 (métodos sincronizados), mientras que puede haber cualquier número de hilos ejecutando el resto de métodos. A continuación de la figura se muestra el código para la misma. Aquellos threads que quieran ejecutar un método sincronizado, mientras otro thread está dentro de un método sincronizado,

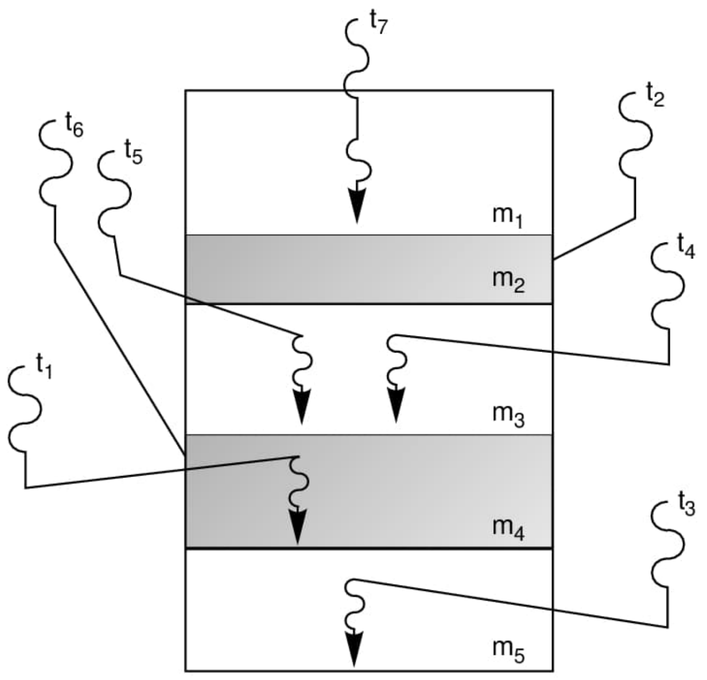

Figura 4.3. Métodos sincronizados y no sincronizados.

tendrán que esperar a que este último abandone la exclusión mutua. La razón natural de abandonar la exclusión mutua es la finalización de la ejecución del método, aunque pueden existir otras como veremos posteriormente.

```
public class Sincronizados {
   public void m1() {
   }
   public synchronized void m2() {
   }
   public void m3() {
   }
   public synchronized void m4() {
   }
   public void m5() {
   }
}
```

Es como si el objeto tuviese un cerrojo. Cuando un método sincronizado está ejecutándose el cerrojo está echado y ningún otro método sincronizado del objeto puede ejecutarse. Cuando se libera el cerrojo, entonces alguno de los threads que estaba esperando por la apertura del cerrojo podrá entrar a ejecutarse (t_2 y t_6 en la Figura 4.3). Recordemos que éste era uno de los estados en los que se podía encontrar un thread según se vio en el Capítulo 2.

Synchronized aplicado a un bloque de código

A veces no interesa que todo el método sea sincronizado, sino una parte de él. En ese caso `synchronized` se puede usar sobre un bloque de código en vez de sobre un método. Para utilizarlo sobre un bloque de código, `synchronized` necesita hacer referencia a un objeto. En el ejemplo siguiente se aplica `synchronized` sobre el mismo objeto sobre el que se está invocando el método `m1()`. Esto hace que ese trozo de código se ejecute en exclusión mutua con cualquier otro bloque de código sincronizado del mismo objeto, incluido los métodos que lleven el modificador `synchronized`.

```
public class Sincronizados2 {
   public void m1() {
      // cualquier código. Será no sincronizado
      synchronized (this) {
         // sólo esta parte es sincronizada
      }
      // cualquier código. Será no sincronizado
   }
}
```

Synchronized aplicado a otro objeto

También es posible aplicar `synchronized` sobre otro objeto distinto a aquel en el que se está utilizando. En el ejemplo siguiente, antes de ejecutar el código que está dentro de `synchronized` hemos de adquirir el cerrojo sobre el objeto `otroObjeto`. Para adquirirlo, ningún otro thread debe tenerlo en ese momento.

```
public class Sincronizados2 {
   Object otroObjeto = new Object ();
   public void m1() {
      // cualquier código. Será no sincronizado
      synchronized (otroObjeto) {
         // sólo esta parte es sincronizada, pero con el cerrojo de
         // otroObjeto
      }
      // cualquier código. Será no sincronizado
   }
}
```

Esta variante suele ser útil cuando se quiere realizar más de una operación de forma atómica sobre un objeto. Por ejemplo, consideremos que queremos hacer un reintegro de una cuenta bancaria y antes queremos consultar el saldo para ver si la operación es factible. Ambas operaciones deben ejecutarse sin que se intercale ninguna otra. Primero adquirimos el cerrojo sobre el objeto `cuentaBancaria` y luego realizamos todas las operaciones que queramos de forma segura.

```
synchronized (cuentaBancaria) {
   if (cuentaBancaria.haySaldo (cantidad))
      cuentaBancaria.reintegro (cantidad);
}
```

Synchronized sobre métodos static

De la misma forma que existe un cerrojo al nivel de objeto, también existe al nivel de clase. `Synchronized` puede ser usado sobre métodos `static`, de forma que se puede proteger el acceso de varios threads a variables `static`. También puede ser usado dentro del código de la siguiente manera:

```
synchronized (nombreClase.class) {...}
```

No obstante, a lo largo de este libro no haremos uso de `synchronized` sobre clases.

4.5.2. El caso de las variables volatile

En Java, las operaciones de asignación sobre variables de tipos primitivos menores de 32 bits se realizan de forma atómica. Para estos casos podemos omitir `synchronized`, eliminando la penalización en el rendimiento que implica su uso. Sin embargo, hay que tener cuidado en situaciones como las que muestra el código siguiente:

```
class Ejemplo {
   private boolean valor = false;
   public void cambiarValor () {
      valor = true;
   }

   public synchronized void esperaPorValorCierto () {
      valor = false;
      // otro thread llama al método valor()
      if (valor) {
         // Puede que este código no se ejecute, aunque valor sea true.
      }
   }
}
```

Esto es así porque la MVJ puede hacer algunas optimizaciones. Como en el método `esperaPorValorCierto()` se hace `valor = false`, el compilador puede suponer que la condición del `if` nunca se va a dar y considerarlo como código muerto que nunca se ejecutará.

La palabra reservada **volatile** sobre un identificador le dice al compilador que otro thread puede cambiar su valor y que no haga optimizaciones de código. En nuestro caso, declararíamos la variable valor como volatile.

4.5.3. Condición de sincronización en Java

La utilización de `synchronized` proporciona un mecanismo de seguridad e integridad. Sin embargo, hay que tener en cuenta también la sincronización y comunicación entre los distintos hilos.

Para implementar las esperas por ocurrencias de eventos y notificaciones de los mismos, Java proporciona tres métodos: `wait()`, `notify()` y `notifyAll()`:

- *wait()*: le indica al hilo en curso que abandone la exclusión mutua sobre el objeto (el cerrojo) y se vaya al estado *espera por wait* hasta que otro hilo lo despierte mediante los métodos `notify()` o `notifyAll()`. El lugar donde se queda esperando el hilo recibe el nombre de *conjunto de espera* o *wait set.*
- *notify()*: un hilo arbitrario del conjunto de espera es seleccionado por el planificador para pasar al estado listo.
- *notifyAll()*: todos los hilos del conjunto de espera son puestos en el estado listo.

- Estos métodos están definidos en la clase `Object` y, por tanto, son heredados por todas las clases de forma implícita.

Antes de que el conjunto de espera de un objeto pueda ser manipulado vía cualquiera de estos tres métodos, el hilo activo debe obtener el cerrojo para ese objeto. Así pues, todas las invocaciones de `wait()`, `notify()` o `notifyAll()` deben ocurrir dentro de un bloque sincronizado.

Implementación de guardas

Hay una forma más o menos estándar que los programadores usan para implementar los hilos que deben esperar porque algo suceda y que recibe el nombre de guarda booleana. En el siguiente trozo de código, si la condición no es cierta, el hilo pasará a formar parte del conjunto de espera, abandonando el cerrojo adquirido sobre el objeto. Cuando le despierten, volverá a evaluar la condición. El encargado de despertarle es el hilo que hace que la variable `condicion` obtenga el valor true mediante el método `hacerCondicionVerdadera ()`.

```
synchronized void hacerCuandoCondicion () {
   while (!condicion)
      try {
         wait();
      }
      catch (InterruptedException e) {
         // código para manejar la excepción
      }
      // código a ejecutar cuando es cierta la condición.
}

synchronized void hacerCondicionVerdadera () {
   condicion = true;
   notify(); // o notifyAll()
}
```

En el código anterior hay que tener en cuenta que:

- Todo el código del método `hacerCuandoCondición ()` se ejecuta dentro de `synchronized`. Si no, después de la sentencia `while` no habría garantía de que la condición continuara siendo cierta.
- Cuando el hilo se suspende por el `wait()`, de forma atómica se hace la suspensión del hilo y la liberación del cerrojo. De lo contrario, podría producirse un `notify()` después de liberar el bloqueo pero antes de que se suspendiera el hilo. El `notify()` no tendría efecto sobre el hilo y se perdería su efectividad.
- La prueba de la condición ha de estar siempre en un bucle. Que se haya despertado un hilo no quiere decir que se vaya a satisfacer la condición necesariamente;

por tanto, hay que volverla a evaluar. No se garantiza que el hilo despertado vaya a ejecutarse inmediatamente, sino que pueden intercalarse otros hilos. Habrá ocasiones en las que la condición no tenga por qué estar en un bucle, pero en la mayoría de los casos sí, y el hecho de ponerlo lo único que implica es una pequeña penalización en cuanto a rendimiento.

- Los métodos `notify()` o `notifyAll()` serán invocados por métodos que harán cambiar la condición por la que están esperando otros hilos. El hilo que ejecuta cualquiera de los dos métodos sigue su ejecución hasta terminar el método o quedarse bloqueado porque ejecute un `wait()` posteriormente.
- Un hilo despertado por `notify()` o `notifyAll()` tendrá que volver a luchar por conseguir el cerrojo sobre el objeto. Nada asegura que lo adquiera por delante de otros hilos que no estaban en el conjunto de espera pero que quieren obtenerlo.

Así pues vamos a tener dos conjuntos de hilos esperando: aquellos que han ejecutado el método `wait()` y aquellos otros que están esperando por obtener el cerrojo. A este último conjunto pertenecerán tanto aquellos hilos que quieran entrar al objeto a ejecutar código sincronizado como aquellos otros que han sido despertados y necesitan adquirir de nuevo el cerrojo del objeto. Este hecho puede observarse en la Figura 4.4.

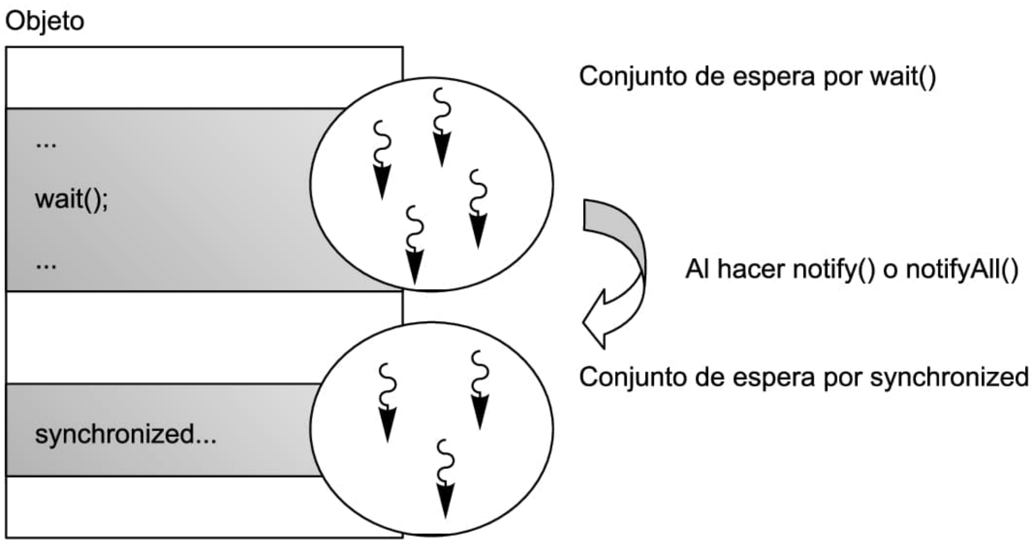
Objeto
...
wait();
...
synchronized...
Conjunto de espera por wait()
Al hacer notify() o notifyAll()
Conjunto de espera por synchronized

Figura 4.4. Conjuntos de espera por wait() y synchronized.

4.5.4. Ejemplos

Como ejemplo veamos el problema del productor/consumidor, el de los filósofos y el de lectores/escritores con preferencia a lectores.

Ejemplo del productor/consumidor

En la clase `Buffer` puede observarse cómo se utiliza una guarda booleana para asegurarnos de que no se toma un elemento cuando el buffer está vacío y que no se pone un elemento cuando el buffer está lleno. Al final de los métodos `coger` y `poner` se despierta a los posibles hilos dormidos en el conjunto de espera. Hay que realizar un `notifyAll()` en vez de `notify()`. Si despertamos sólo un hilo, no podremos asegurar que ese hilo vaya a satisfacer su condición. Se podría entonces dar el caso de tener hilos en espera que sí satisfacen su condición, pero, sin embargo, no los hemos despertado. Por otra parte, el hecho de usar `notifyAll()` puede implicar que sean muchos los hilos que se despierten simplemente para volver al estado en espera pues no van a satisfacer su condición. Esto implica una penalización en cuanto a rendimiento. Como veremos en el siguiente capítulo este problema puede solventarse haciendo uso de las denominadas *variables condición*. En este caso, el buffer es implementado como una pila.

Clase buffer

```
public class Buffer { // no es un Thread
   private int cima, capacidad, vector[];

   Buffer (int i) {
      cima = 0;
      capacidad = i;
      vector = new int[i];
   }

   synchronized public int extraer ( ) {
      while (cima == 0)
         try {
            wait();
         } catch (InterruptedException e){;}
      notifyAll();
      return vector[--cima];
   }

   synchronized public void insertar (int elem) {
      while (cima==capacidad-1)
        try {
         wait();
        } catch (InterruptedException e){;}
      vector[cima]=elem;
      cima++;
      notifyAll();
   }
}
```

Clase Consumidor

```
public class Consumidor extends Thread {
   int elem;
   Buffer buffer;
   Consumidor (Buffer b, int i) {
      buffer = b;
   }

   public void run ( ) {
      try {
         elem = buffer.extraer();
      } catch (Exception e) {}
      return;
   }
}
```

Clase Productor

```
public class Productor extends Thread {
   Buffer buffer;
   int elem;

   Productor (Buffer b, int i) {
      elem=i;
      buffer = b;
      System.out.println ("entra el productor "+i);
   }

   public void run ( ) {
      try {
         buffer.insertar(elem);
      } catch (Exception e) {}
      System.out.println ("he puesto el elemento "+elem);
      return;
   }
}
```

Clase con el programa principal

```
public class ProductorConsumidor {
   static Buffer buf = new Buffer(3);

   public static void main (String[] args ) {
      for (int i=1;i<=5;i++)
         new Productor (buf,i).start();
      for (int j=1;j<=5;j++)
         new Consumidor (buf,j).start();
      System.out.println ("Fin del hilo main");
   }
}
```

Ejemplo de los filósofos

Clase Palillo

```
public class Palillo {

   boolean libre;

   Palillo () {

      libre = true;
   }

   synchronized public void coger (int quien) {
      while (!libre) {// el otro filósofo ha cogido este palillo
         try {wait();}
         catch (Exception e) {}
      }
      libre = false;
   }

   synchronized public void soltar () {
      libre = true;
      notifyAll ();
   }
}
```

Clase Contador

Esta clase evita que se produzca un interbloqueo, impidiendo que 4 filósofos traten de comer al mismo tiempo.

```
public class Contador {
   int cont;
   int tope;

   Contador (int _tope) {
      cont = 0;
      tope = _tope;
   }

   public void inc () {
      while (cont == tope) {
         try {wait();} // bloqueado por ser el quinto filósofo
         catch (Exception e) {}
      }
      cont ++;
   }
   synchronized public void dec () {
      cont --;
      notifyAll();
   }

   synchronized public int valor () {
      return cont;
   }
}
```

Clase Filósofo

```
public class Filosofo extends Thread {
   int quienSoy = 0;
   Palillo palDer, palIzq;
   Contador cont;
   int numeroOperaciones = 10;

   public Filosofo (int _quienSoy, Contador _cont,
                    Palillo _palDer, Palillo _palIzq) {
      quienSoy = _quienSoy;
      palDer = _palDer;
      palIzq = _palIzq;
      cont = _cont;
   }

   public void run () {
      for (int i=0;i<numeroOperaciones;i++) {
         System.out.println ("Filósofo "+quienSoy+" pensando");
         cont.inc();
         palDer.coger(quienSoy);
         palIzq.coger(quienSoy);
         System.out.println ("Filósofo "+quienSoy+" comiendo");
         palDer.soltar();
         palIzq.soltar();
         cont.dec();
      }
   }
}
```

Clase con el programa principal

El programa lanza los cinco filósofos y luego espera por su terminación con el método `join()`.

```
public class MainFilosofos {
   Filosofo f[] = new Filosofo[5];
   Palillo palillos[] = new Palillo[5];
   Contador contador;
   int numFil = 5;

   public MainFilosofos () {
      contador = new Contador (numFil-1);

      for (int i=0;i<numFil;i++) {
         palillos[i] = new Palillo();
      }

      for (int i=0;i<numFil;i++) {
         f[i] = new Filosofo(i, contador, palillos[i],
         palillos[(i+1)%numFil]); f[i].start();
      }

      for (int i=0;i<numFil;i++) {
         try {
            f[i].join(); // espera al resto de filósofos
         } catch (Exception e) {}
      }
   }

   public static void main (String args[]) {
      new MainFilosofos();
   }
}
```

Ejemplo de lectores y escritores

Clase Escritor

```
public class Escritor extends Thread {
   int miId;
   Recurso recurso;

   public Escritor (Recurso _recurso, int _miId) {
      miId = _miId;
      recurso = _recurso;
   }

   public void run () {
      int i= 0;
      while (i<10) {
            System.out.println ("Escritor "+ miId + " quiere escribir");
            recurso.escribir ();
            System.out.println ("Escritor "+ miId + " ha terminado");
            i++;
        }
    }
}
```

Clase Lector

```
public class Lector extends Thread {
    int miId;
    Recurso recurso;

    public Lector (Recurso _recurso, int _miId) {
        miId = _miId;
        recurso = _recurso;
    }

    public void run () {
    int i = 0;
        while (i<10) {
            System.out.println ("Lector "+ miId + " quiere leer");
            recurso.leer ();
            System.out.println ("Lector "+ miId + " ha terminado");
            i++;
        }
    }
}
```

Clase Recurso

Esta clase implementa el recurso al que se quiere acceder con la política de lectores y escritores. Podemos observar cómo hay partes que están sincronizadas y otras partes que no lo están para permitir un acceso concurrente de lectores.

```
public class Recurso {

    int numLectores = 0;
    boolean hayEscritor = false;

    public Recurso () {
    }

    public void leer () {
      //protocolo de entrada
      synchronized (this) {
         while (hayEscritor)
            try {
               wait();
            }
            catch (Exception e) {}
         numLectores++;
      }

      // leyendo. Sin sincronizar para permitir concurrencia.

      // protocolo de salida
      synchronized (this) {
         numLectores--;
         if (numLectores ==0) notifyAll();
      }
   }

   synchronized public void escribir () {

      // protocolo de entrada
      synchronized (this) {
         while (hayEscritor || (numLectores > 0))
            try {
                 wait();
            }
            catch (Exception e) {e.printStackTrace();}
         hayEscritor = true;
      }

      // escribiendo. Sin sincronizar, pero sólo habrá un escritor.

      // protocolo de salida
      synchronized (this) {
         hayEscritor = false;
         notifyAll();
      }
   }
}
```

Clase con el programa principal

```
public class LyE {

   public static void main (String args[]) {
      Recurso recurso = new Recurso();
      for (int i=0;i<3;i++)
         new Lector (recurso, i).start();

      for (int i=0;i<3;i++)
         new Escritor (recurso, i).start();
   }
}
```

4.5.5. Implementación de semáforos con las primitivas de Java

Veremos a continuación cómo podemos implementar un semáforo binario y un semáforo general usando las primitivas propias de Java. Con esto quedará demostrado que las primitivas de Java tienen como mínimo el mismo poder de expresividad que los semáforos y que cualquier problema que podamos resolver con semáforos también podremos resolverlo con Java.

La ventaja de Java con respecto a los semáforos es que sus primitivas son más fáciles de utilizar y no están dispersas por el código de los objetos cliente, sino centradas en los objetos que juegan el papel de servidores con lo que el código se hace más fácilmente modificable. No obstante, si uno se siente más cómodo podría utilizar el paquete `Semáforo` que se implementa a continuación para sincronizar sus programas en Java. Utilizamos nombres en mayúsculas para `WAIT()` y `SIGNAL()` para evitar confusiones con el `wait()` propio de Java.

Implementación del semáforo binario

```
package Semaforo;

public class SemaforoBinario {
   protected int contador = 0;

   public SemaforoBinario (int valorInicial) {
      contador = valorInicial;
   }

   synchronized public void WAIT () {
      while (contador == 0)
         try {
            wait();
         }
         catch (Exception e) {}
      contador--;
   }

   synchronized public void SIGNAL () {
      contador = 1;
      notify(); /* aquí con despertar a uno es suficiente, pues todos
                   esperan por la misma condición */
   }
}
```

Implementación del semáforo general

Heredamos de la clase anterior, redefiniendo el método SIGNAL() pues ahora el contador puede pasar de uno.

```
package Semaforo;

public class SemaforoGeneral extends SemaforoBinario {

public SemaforoGeneral (int valorInicial) {
      super (valorInicial);
   }
   synchronized public void SIGNAL () {
      contador++;
      notify();
   }
}
```

Puede observarse que estas implementaciones no nos garantizan una política FIFO para la gestión de los procesos bloqueados. Queda como ejercicio para el lector diseñar ambos tipos de semáforos donde se siga una política FIFO. Como idea se sugiere la creación de un vector donde cada celda es un hilo. Cada vez que un hilo se bloquea, se añade al final de este vector. Un hilo se desbloqueará sólo cuando sea el primero de este vector. Entonces también se eliminará el hilo del vector.

El vector se declararía como:

```
Vector bloqueados = new Vector(50);
// 50 sería el número máximo de procesos bloqueados.
```

Cada vez que un hilo se bloquea dentro de la operación `WAIT()` haría algo como:

```
try {
   bloqueados.addElement (Thread.currentThread());
   do {
      wait();
      /* la condición de salida es que este thread sea el que más tiempo
      lleva esperando en el semáforo
      */
      condSalida = bloqueados.firstElement().equals
      (Thread.currentThread());

      // si este thread no puede despertarse, se despierta a otro
      if (!condSalida) notify ();
   }
   while (!condSalida);
   condSalida = false;
   bloqueados.removeElement (Thread.currentThread());
} catch (Exception e) {}
```

4.6. Inconvenientes del mecanismo de los semáforos

Una vez que finalizamos el análisis del mecanismo de semáforo para implementar la sincronización que surge entre procesos concurrentes podemos decir que, si bien es un mecanismo con un enorme poder expresivo, encontramos en su uso los siguientes inconvenientes:

1. Es un mecanismo de bajo nivel, no estructurado, que fácilmente nos puede llevar a errores transitorios. La ejecución de cada sección crítica debe comenzar con un `wait` y terminar con un `signal` sobre el mismo semáforo. La omisión de un `wait` o `signal`, o realizar un `wait` sobre un semáforo y un `signal` sobre otro accidentalmente puede tener efectos no deseables, como no asegurar una ejecución en exclusión mutua de secciones críticas.
2. No se puede restringir el tipo de operaciones realizadas sobre los recursos.
3. Cuando se usan semáforos, un programador puede olvidar el incluir en las secciones críticas todas las sentencias que hagan referencia a los objetos compartidos.
4. Tanto la exclusión mutua como la condición de sincronización se implementan usando el mismo par de primitivas. Esto hace difícil el identificar el propósito de un `wait` o `signal` sin mirar las otras operaciones realizadas sobre el mismo semáforo. Como la exclusión mutua y la condición de sincronización son conceptos distintos, deben tener distintas notaciones.
5. Los programas que utilizan semáforos son difíciles de mantener, ya que el código de sincronización está repartido entre los distintos procesos, con lo que cualquier modificación implica la revisión de todos los procesos.

PROBLEMAS RESUELTOS

1. Supuesto que tenemos varios terminales que comparten una impresora, programar el acceso a la impresora usando semáforos.

Solución: Usaremos el semáforo `s`, que inicialmente valdrá 1.

```
process terminal(i:integer);
begin
```

Documentation

The Java™ Tutorials

Trail: Essential Java Classes
Lesson: Concurrency

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Dev.java for updated tutorials taking advantage of the latest releases.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Synchronization

Threads communicate primarily by sharing access to fields and the objects reference fields refer to. This form of communication is extremely efficient, but makes two kinds of errors possible: *thread interference* and *memory consistency errors*. The tool needed to prevent these errors is *synchronization*.

However, synchronization can introduce *thread contention*, which occurs when two or more threads try to access the same resource simultaneously *and* cause the Java runtime to execute one or more threads more slowly, or even suspend their execution. Starvation and livelock are forms of thread contention. See the section Liveness for more information.

This section covers the following topics:

- Thread Interference describes how errors are introduced when multiple threads access shared data.
- Memory Consistency Errors describes errors that result from inconsistent views of shared memory.
- Synchronized Methods describes a simple idiom that can effectively prevent thread interference and memory consistency errors.
- Implicit Locks and Synchronization describes a more general synchronization idiom, and describes how synchronization is based on implicit locks.
- Atomic Access talks about the general idea of operations that can't be interfered with by other threads.

Previous page: The SimpleThreads Example
Next page: Thread Interference

Documentation

The Java™ Tutorials

Trail: Essential Java Classes
Lesson: Concurrency
Section: Synchronization

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Dev.java for updated tutorials taking advantage of the latest releases.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Thread Interference

Consider a simple class called `Counter`

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }

}
```

`Counter` is designed so that each invocation of `increment` will add 1 to `c`, and each invocation of `decrement` will subtract 1 from `c`. However, if a `Counter` object is referenced from multiple threads, interference between threads may prevent this from happening as expected.

Interference happens when two operations, running in different threads, but acting on the same data, *interleave*. This means that the two operations consist of multiple steps, and the sequences of steps overlap.

It might not seem possible for operations on instances of `Counter` to interleave, since both operations on `c` are single, simple statements. However, even simple statements can translate to multiple steps by the virtual machine. We won't examine the specific steps the virtual machine takes — it is enough to know that the single expression `c++` can be decomposed into three steps:

1. Retrieve the current value of `c`.
2. Increment the retrieved value by 1.
3. Store the incremented value back in `c`.

The expression `c--` can be decomposed the same way, except that the second step decrements instead of increments.

Suppose Thread A invokes `increment` at about the same time Thread B invokes `decrement`. If the initial value of `c` is `0`, their interleaved actions might follow this sequence:

1. Thread A: Retrieve c.
2. Thread B: Retrieve c.
3. Thread A: Increment retrieved value; result is 1.
4. Thread B: Decrement retrieved value; result is -1.
5. Thread A: Store result in c; c is now 1.
6. Thread B: Store result in c; c is now -1.

Thread A's result is lost, overwritten by Thread B. This particular interleaving is only one possibility. Under different circumstances it might be Thread B's result that gets lost, or there could be no error at all. Because they are unpredictable, thread interference bugs can be difficult to detect and fix.

The Java™ Tutorials

Trail: Essential Java Classes
Lesson: Concurrency
Section: Synchronization

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Dev.java for updated tutorials taking advantage of the latest releases.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Memory Consistency Errors

Memory consistency errors occur when different threads have inconsistent views of what should be the same data. The causes of memory consistency errors are complex and beyond the scope of this tutorial. Fortunately, the programmer does not need a detailed understanding of these causes. All that is needed is a strategy for avoiding them.

The key to avoiding memory consistency errors is understanding the *happens-before* relationship. This relationship is simply a guarantee that memory writes by one specific statement are visible to another specific statement. To see this, consider the following example. Suppose a simple `int` field is defined and initialized:

```
int counter = 0;
```

The `counter` field is shared between two threads, A and B. Suppose thread A increments `counter`:

```
counter++;
```

Then, shortly afterwards, thread B prints out `counter`:

```
System.out.println(counter);
```

If the two statements had been executed in the same thread, it would be safe to assume that the value printed out would be "1". But if the two statements are executed in separate threads, the value printed out might well be "0", because there's no guarantee that thread A's change to `counter` will be visible to thread B — unless the programmer has established a happens-before relationship between these two statements.

There are several actions that create happens-before relationships. One of them is synchronization, as we will see in the following sections.

We've already seen two actions that create happens-before relationships.

- When a statement invokes `Thread.start`, every statement that has a happens-before relationship with that statement also has a happens-before relationship with every statement executed by the new thread. The effects of the code that led up to the creation of the new thread are visible to the new thread.
- When a thread terminates and causes a `Thread.join` in another thread to return, then all the statements executed by the terminated thread have a happens-before relationship with all the statements following the successful join. The effects of the code in the thread are now visible to the thread that performed the join.

For a list of actions that create happens-before relationships, refer to the Summary page of the `java.util.concurrent` package..

Previous page: Thread Interference
Next page: Synchronized Methods

Documentation

The Java™ Tutorials

Trail: Essential Java Classes
Lesson: Concurrency
Section: Synchronization

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Dev.java for updated tutorials taking advantage of the latest releases.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Synchronized Methods

The Java programming language provides two basic synchronization idioms: *synchronized methods* and *synchronized statements*. The more complex of the two, synchronized statements, are described in the next section. This section is about synchronized methods.

To make a method synchronized, simply add the `synchronized` keyword to its declaration:

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

If `count` is an instance of `SynchronizedCounter`, then making these methods synchronized has two effects:

- First, it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.
- Second, when a synchronized method exits, it automatically establishes a happens-before relationship with *any subsequent invocation* of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

Note that constructors cannot be synchronized — using the `synchronized` keyword with a constructor is a syntax error. Synchronizing constructors doesn't make sense, because only the thread that creates an object should have access to it while it is being constructed.

Warning: When constructing an object that will be shared between threads, be very careful that a reference to the object does not "leak" prematurely. For example, suppose you want to maintain a `List` called `instances` containing every instance of class. You might be tempted to add the following line to your constructor:

```
instances.add(this);
```

But then other threads can use `instances` to access the object before construction of the object is complete.

Synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors: if an object is visible to more than one thread, all reads or writes to that object's variables are done through `synchronized` methods. (An important exception: `final` fields, which cannot be modified after the object is constructed, can be safely read through non-synchronized methods, once the object is constructed) This strategy is effective, but can present problems with liveness, as we'll see later in this lesson.

About Oracle | Contact Us | Legal Notices | Terms of Use | Your Privacy Rights

Previous page: Memory Consistency Errors
Next page: Intrinsic Locks and Synchronization

Preferencias sobre cookies | Ad Choices

The Java™ Tutorials

Trail: Essential Java Classes
Lesson: Concurrency
Section: Synchronization

Intrinsic Locks and Synchronization

Synchronization is built around an internal entity known as the *intrinsic lock* or *monitor lock*. (The API specification often refers to this entity simply as a "monitor.") Intrinsic locks play a role in both aspects of synchronization: enforcing exclusive access to an object's state and establishing happens-before relationships that are essential to visibility.

Every object has an intrinsic lock associated with it. By convention, a thread that needs exclusive and consistent access to an object's fields has to *acquire* the object's intrinsic lock before accessing them, and then *release* the intrinsic lock when it's done with them. A thread is said to *own* the intrinsic lock between the time it has acquired the lock and released the lock. As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.

When a thread releases an intrinsic lock, a happens-before relationship is established between that action and any subsequent acquisition of the same lock.

Locks In Synchronized Methods

When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns. The lock release occurs even if the return was caused by an uncaught exception.

You might wonder what happens when a static synchronized method is invoked, since a static method is associated with a class, not an object. In this case, the thread acquires the intrinsic lock for the `Class` object associated with the class. Thus access to class's static fields is controlled by a lock that's distinct from the lock for any instance of the class.

Synchronized Statements

Another way to create synchronized code is with *synchronized statements*. Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

In this example, the `addName` method needs to synchronize changes to `lastName` and `nameCount`, but also needs to avoid synchronizing invocations of other objects' methods. (Invoking other objects' methods from synchronized code can create problems that are described in the section on Liveness.) Without synchronized statements, there would have to be a separate, unsynchronized method for the sole purpose of invoking `nameList.add`.

Synchronized statements are also useful for improving concurrency with fine-grained synchronization. Suppose, for example, class `MsLunch` has two instance fields, `c1` and `c2`, that are never used together. All updates of these fields must be synchronized, but there's no reason to prevent an update of c1 from being interleaved with an update of c2 — and doing so reduces concurrency by creating unnecessary blocking. Instead of using synchronized methods or otherwise using the lock associated with `this`, we create two objects solely to provide locks.

```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1() {
        synchronized(lock1) {
            c1++;
        }
    }

    public void inc2() {
        synchronized(lock2) {
            c2++;
        }
    }
}
```

Use this idiom with extreme care. You must be absolutely sure that it really is safe to interleave access of the affected fields.

Reentrant Synchronization

Recall that a thread cannot acquire a lock owned by another thread. But a thread *can* acquire a lock that it already owns. Allowing a thread to acquire the same lock more than once enables *reentrant synchronization*. This describes a situation where synchronized code, directly or indirectly, invokes a method that also contains synchronized code, and both sets of code use the same lock. Without reentrant synchronization, synchronized code would have to take many additional precautions to avoid having a thread cause itself to block.

Previous page: Synchronized Methods
Next page: Atomic Access

The Java™ Tutorials

Trail: Essential Java Classes
Lesson: Concurrency
Section: Synchronization

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Dev.java for updated tutorials taking advantage of the latest releases.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Atomic Access

In programming, an *atomic* action is one that effectively happens all at once. An atomic action cannot stop in the middle: it either happens completely, or it doesn't happen at all. No side effects of an atomic action are visible until the action is complete.

We have already seen that an increment expression, such as `c++`, does not describe an atomic action. Even very simple expressions can define complex actions that can decompose into other actions. However, there are actions you can specify that are atomic:

- Reads and writes are atomic for reference variables and for most primitive variables (all types except `long` and `double`).
- Reads and writes are atomic for *all* variables declared `volatile` (*including* `long` and `double` variables).

Atomic actions cannot be interleaved, so they can be used without fear of thread interference. However, this does not eliminate all need to synchronize atomic actions, because memory consistency errors are still possible. Using `volatile` variables reduces the risk of memory consistency errors, because any write to a `volatile` variable establishes a happens-before relationship with subsequent reads of that same variable. This means that changes to a `volatile` variable are always visible to other threads. What's more, it also means that when a thread reads a `volatile` variable, it sees not just the latest change to the `volatile`, but also the side effects of the code that led up the change.

Using simple atomic variable access is more efficient than accessing these variables through synchronized code, but requires more care by the programmer to avoid memory consistency errors. Whether the extra effort is worthwhile depends on the size and complexity of the application.

Some of the classes in the `java.util.concurrent` package provide atomic methods that do not rely on synchronization. We'll discuss them in the section on High Level Concurrency Objects.

Previous page: Intrinsic Locks and Synchronization
Next page: Liveness

```
public String toString()
```

Returns a string representation of the object. Satisfying this method's contract implies a non-`null` result must be returned.

API Note:

In general, the `toString` method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method. The string output is not necessarily stable over time or across JVM invocations.

Implementation Requirements:

The `toString` method for class `Object` returns a string consisting of the name of the class of which the object is an instance, the at-sign character `` `@' ``, and the unsigned hexadecimal representation of the hash code of the object. In other words, this method returns a string equal to the value of:

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

The `Objects.toIdentityString` method returns the string for an object equal to the string that would be returned if neither the `toString` nor `hashCode` methods were overridden by the object's class.

Returns:

a string representation of the object

notify

```
public final void notify()
```

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. A thread waits on an object's monitor by calling one of the `wait` methods.

The awakened thread will not be able to proceed until the current thread relinquishes the lock on this object. The awakened thread will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, the awakened thread enjoys no reliable privilege or disadvantage in being the next thread to lock this object.

This method should only be called by a thread that is the owner of this object's monitor. A thread becomes the owner of the object's monitor in one of three ways:

- By executing a synchronized instance method of that object.
- By executing the body of a `synchronized` statement that synchronizes on the object.
- For objects of type `Class,` by executing a static synchronized method of that class.

Only one thread at a time can own an object's monitor.

Throws:

`IllegalMonitorStateException` - if the current thread is not the owner of this object's monitor.

See Also:

`notifyAll()`, `wait()`

notifyAll

```
public final void notifyAll()
```

Wakes up all threads that are waiting on this object's monitor. A thread waits on an object's monitor by calling one of the `wait` methods.

The awakened threads will not be able to proceed until the current thread relinquishes the lock on this object. The awakened threads will compete in the usual manner with any other threads that might be actively competing to synchronize on this object; for example, the awakened threads enjoy no reliable privilege or disadvantage in being the next thread to lock this object.

This method should only be called by a thread that is the owner of this object's monitor. See the `notify` method for a description of the ways in which a thread can become the owner of a monitor.

Throws:

`IllegalMonitorStateException` - if the current thread is not the owner of this object's monitor.

See Also:

`notify()`, `wait()`

wait

```
public final void wait()
                throws InterruptedException
```

Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*.

In all respects, this method behaves as if `wait(0L, 0)` had been called. See the specification of the `wait(long, int)` method for details.

Throws:

`IllegalMonitorStateException` - if the current thread is not the owner of the object's monitor

`InterruptedException` - if any thread interrupted the current thread before or while the current thread was waiting. The *interrupted status* of the current thread is cleared when this exception is thrown.

See Also:

`notify()`,
`notifyAll()`,
`wait(long)`,
`wait(long, int)`

wait

```
public final void wait(long timeoutMillis)
                throws InterruptedException
```

Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*, or until a certain amount of real time has elapsed.

In all respects, this method behaves as if `wait(timeoutMillis, 0)` had been called. See the specification of the `wait(long, int)` method for details.

Parameters:

`timeoutMillis` - the maximum time to wait, in milliseconds

Throws:

`IllegalArgumentException` - if `timeoutMillis` is negative

`IllegalMonitorStateException` - if the current thread is not the owner of the object's monitor

`InterruptedException` - if any thread interrupted the current thread before or while the current thread was waiting. The *interrupted status* of the current thread is cleared when this exception is thrown.

See Also:

`notify()`,
`notifyAll()`,
`wait()`,
`wait(long, int)`

wait

```
public final void wait(long timeoutMillis,
                       int nanos)
                throws InterruptedException
```

Causes the current thread to wait until it is awakened, typically by being *notified* or *interrupted*, or until a certain amount of real time has elapsed.

The current thread must own this object's monitor lock. See the `notify` method for a description of the ways in which a thread can become the owner of a monitor lock.

This method causes the current thread (referred to here as *T*) to place itself in the wait set for this object and then to relinquish any and all synchronization claims on this object. Note that only the locks on this object are relinquished; any other objects on which the current thread may be synchronized remain locked while the thread waits.

Thread *T* then becomes disabled for thread scheduling purposes and lies dormant until one of the following occurs:

- Some other thread invokes the `notify` method for this object and thread *T* happens to be arbitrarily chosen as the thread to be awakened.
- Some other thread invokes the `notifyAll` method for this object.
- Some other thread interrupts thread *T*.
- The specified amount of real time has elapsed, more or less. The amount of real time, in nanoseconds, is given by the expression `1000000 * timeoutMillis + nanos`. If `timeoutMillis` and `nanos` are both zero, then real time is not taken into consideration and the thread waits until awakened by one of the other causes.
- Thread *T* is awakened spuriously. (See below.)

The thread *T* is then removed from the wait set for this object and re-enabled for thread scheduling. It competes in the usual manner with other threads for the right to synchronize on the object; once it has regained control of the object, all its synchronization claims on the object are restored to the status quo ante - that is, to the situation as of the time that the `wait` method was invoked. Thread *T* then returns from the invocation of the `wait` method. Thus, on return from the `wait` method, the synchronization state of the object and of thread `T` is exactly as it was when the `wait` method was invoked.

A thread can wake up without being notified, interrupted, or timing out, a so-called *spurious wakeup*. While this will rarely occur in practice, applications must guard against it by testing for the condition that should have caused the thread to be awakened, and continuing to wait if the condition is not satisfied. See the example below.

For more information on this topic, see section 14.2, "Condition Queues," in Brian Goetz and others' *Java Concurrency in Practice* (Addison-Wesley, 2006) or Item 81 in Joshua Bloch's *Effective Java, Third Edition* (Addison-Wesley, 2018).

If the current thread is interrupted by any thread before or while it is waiting, then an `InterruptedException` is thrown. The *interrupted status* of the current thread is cleared when this exception is thrown. This exception is not thrown until the lock status of this object has been restored as described above.

API Note:

The recommended approach to waiting is to check the condition being awaited in a `while` loop around the call to `wait`, as shown in the example below. Among other things, this approach avoids problems that can be caused by spurious wakeups.

```
     synchronized (obj) {
         while ( <condition does not hold and timeout not exceeded> ) {
             long timeoutMillis = ... ; // recompute timeout values
             int nanos = ... ;
             obj.wait(timeoutMillis, nanos);
         }
         ... // Perform action appropriate to condition or timeout
     }
```

Parameters:

`timeoutMillis` - the maximum time to wait, in milliseconds

`nanos` - additional time, in nanoseconds, in the range 0-999999 inclusive

Throws:

`IllegalArgumentException` - if `timeoutMillis` is negative, or if the value of `nanos` is out of range

`IllegalMonitorStateException` - if the current thread is not the owner of the object's monitor

`InterruptedException` - if any thread interrupted the current thread before or while the current thread was waiting. The *interrupted status* of the current thread is cleared when this exception is thrown.

See Also:

`notify()`, `notifyAll()`, `wait()`, `wait(long)`

finalize

```
@Deprecated(since="9",
            forRemoval=true)
protected void finalize()
                 throws Throwable
```

Deprecated, for removal: This API element is subject to removal in a future version.
Finalization is deprecated and subject to removal in a future release. The use of finalization can lead to problems with security, performance, and reliability. See JEP 421 for discussion and alternatives.

Subclasses that override `finalize` to perform cleanup should use alternative cleanup mechanisms and remove the `finalize` method. Use `Cleaner` and `PhantomReference` as safer ways to release resources when an object becomes unreachable. Alternatively, add a `close` method to explicitly release resources, and implement `AutoCloseable` to enable use of the `try`-with-resources statement.

This method will remain in place until finalizers have been removed from most existing code.

Called by the garbage collector on an object when garbage collection determines that there are no more references to the object. A subclass overrides the `finalize` method to dispose of system resources or to perform other cleanup.

When running in a Java virtual machine in which finalization has been disabled or removed, the garbage collector will never call `finalize()`. In a Java virtual machine in which finalization is enabled, the garbage collector might call `finalize` only after