Capítulo 8

Programación Concurrente

Procesos e Hilos

Un proceso es todo aquello que el computador necesita para ejecutar una aplicación. Técnicamente, un proceso es una colección de:

- Espacio de memoria virtual.
- Código.
- Datos (como las variables globales y la memoria del montón).
- Recursos del sistema (como los descriptores de archivos y los manejadores o *handles*).

Un hilo es la instancia de ejecución dentro de un proceso, es código que está siendo ejecutado *serialmente* dentro de un proceso. Técnicamente, un hilo es una colección de:

- Estados de la pila.
- Información del contexto de ejecución (como el estado de los registros del CPU)

Un procesador ejecuta hilos, no procesos, por lo que una aplicación tiene al menos un proceso, y un proceso tiene al menos un hilo de ejecución. El primer hilo de un proceso, el que comienza su ejecución, se le llama *hilo primario*. A partir del hilo primario pueden crearse *hilos secundarios*, y a su vez crearse de éstos. Un hilo puede a su vez crear otros procesos. Cada hilo puede ejecutar secciones de código distintas, o múltiples hilos pueden ejecutar la misma sección de código. Los hilos de un proceso tienen pilas independientes, pero comparten los datos y recursos de su proceso.

A un sistema operativo capaz de "ejecutar" más de un hilo, pertenecientes al mismo o a un distinto proceso, se le llama *multitarea*. Es importante no perder de vista que el sistema operativo mismo es un conjunto de procesos con atribuciones especiales. El sistema operativo (S.O.) utiliza elementos del hardware del computador, como veremos más adelante, para asistirse en el control de la multitarea.

La aparente ejecución simultánea de más de un hilo es sólo un efecto del rápido paso de la ejecución de un hilo a otro, por turnos, por parte del computador. Un computador con un único procesador es capaz de ejecutar el código de un único hilo por vez. Dado que el hilo es la unidad de ejecución de las aplicaciones, es también la unidad de planificación del orden de ejecución de las mismas. El componente del sistema operativo encargado de determinar cuándo y qué tan frecuentemente un hilo debe ejecutarse se le llama *planificador* (scheduler). Cuando uno de los hilos de un proceso está en ejecución, se dice que su proceso está en ejecución. Sin embargo, vale insistir, el decir que un proceso se ejecuta significa que uno de sus hilos se está ejecutando. El computador ejecuta hilos, no procesos.

Para sacar a un hilo de ejecución e ingresar otro, se realiza un *cambio de contexto*. Este cambio se realiza mediante una interrupción de hardware, la que ejecuta una rutina instalada por el S.O. cuando se arrancó el computador (como es de suponerse, el planificador) y que realiza lo siguiente:

- Guarda la información del estado de ejecución del hilo saliente.
- Carga la información del estado de ejecución del hilo entrante.
- Finaliza la rutina de la interrupción, por lo que "continúa" la ejecución del hilo configurado, es decir, el hilo entrante.

Cuando el cambio de contexto es entre hilos de un mismo proceso, éste se realiza en forma rápida, dado que la información del contexto de ejecución relacionada con el proceso, como es el espacio de direccionamiento, no requiere ser modificada, todos los hilos de un proceso comparten el mismo espacio de direccionamiento. Cuando el cambio de contexto es entre hilos de distintos procesos, éste es más lento debido al número de cambios que es necesario realizar. Éste es el motivo por el cual los S.O. multitarea modernos separan los conceptos de proceso e hilo, dado que técnicamente un S.O. podría trabajar únicamente con procesos como unidad de ejecución. El uso de los hilos le permite a los programas que requieren realizar varias tareas en paralelo o concurrentemente, usar hilos en lugar de crear nuevos procesos, mejorando el desempeño general del computador.

Espacio de Memoria Virtual

Los procesos establecen un nivel de aislamiento entre una aplicación y otra, donde los hilos de cada uno son ejecutados como si sólo existiese un único proceso en el computador. Esto se consigue manejando por cada proceso un espacio de memoria virtual. Este espacio de memoria se obtiene con un manejo lógico de la memoria física del computador, de forma que cada proceso tenga la "sensación" de que cuenta con un espacio de memoria mayor del que físicamente posee la memoria RAM del computador. Cuando un hilo de un proceso quiere acceder a una posición de memoria virtual se produce un "mapeo" de la dirección que maneja el código del programa, a la dirección física. Si el hilo intenta trabajar con más memoria de la existente en la RAM, el S.O. juega con la memoria en disco, bajando y subiendo bloques de memoria de la RAM en un proceso llamado *swapping*.

El rango de valores para las direcciones de memoria que los hilos de un proceso pueden usar se le llama *espacio de direccionamiento*. Dado que dichas direcciones son virtuales y que los procesos deben ejecutarse independientemente unos de otros, la misma dirección para hilos de procesos distintos se referirá a posiciones de memoria distintas.

Dado que cada proceso maneja su propio espacio de direccionamiento, se evita que un hilo de un proceso pueda acceder inadvertidamente a la memoria de otro hilo, produciendo errores. Esto permite una gran seguridad al trabajar con varios procesos a la vez, pero dificulta la comunicación entre procesos.

Ciclo de vida de un Hilo

Desde el momento en que la información utilizada por el computador para poder ejecutar un hilo es creada por el S.O., el hilo pasa por una secuencia de etapas hasta que es eliminado. La figura 8.1 muestra un diagrama de estados general para un hilo. Los cambios de estado en el ciclo de vida de un hilo son producidos por el sistema operativo o por una llamada a algún método.

Cuando un hilo se crea no se ejecuta en "ese" momento, su estado es *creado* y deberá pasar al estado *listo* para ser elegible por el planificador y ejecutarse. El estado "*listo*" significa *listo para ser ejecutado*. En todo momento del trabajo de un computador, el planificador de tareas mantiene una *lista de los hilos listos*, de forma que cuando el hilo en ejecución actual termine de ejecutar, el planificador escoja de la lista de "listos" un nuevo hilo para ser ejecutado. Cuando un hilo entra a ejecutarse, su estado pasa a "*en ejecución*".

Note que un hilo puede pasar del estado "*listo*" al de "*en ejecución*" y nuevamente al estado "*listo*" repetidas veces. Estos cambios se basan en la forma en que el S.O. planifica la ejecución de los hilos listos.

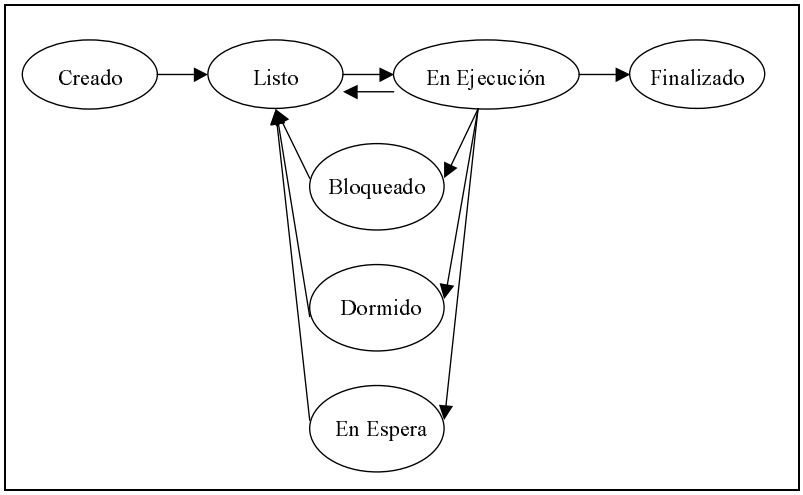

Figura 8.1. Ciclo de vida de un hilo

Note que un hilo en ejecución puede pasar a un estado de latencia en el que no se le asignará el CPU hasta que ocurra el evento adecuado que lo saque de dicho estado, pasándolo al estado de *"listo"*. Estos estados de latencia son: *Bloqueado, Dormido, En Espera.*

Cuando un hilo ha finalizado (o muerto), la información que el S.O. guarda de él no ha sido aún eliminada. Esto permite que otro hilo en ejecución pueda verificar dicho estado y pueda tomar alguna acción, lo que puede servir para sincronizar las acciones de los hilos.

Planificación

Un hilo se ejecuta hasta que muere, le cede la ejecución a otros hilos o es interrumpido por el planificador. Cuando el planificador tiene la capacidad de interrumpir un hilo, cuando su tiempo de ejecución vence u otro hilo de mayor prioridad está listo para ser ejecutado, se le llama *preentivo*. Cuando debe esperar a que dicho hilo ceda voluntariamente el CPU a otros hilos, se le llama *no preentivo*. Éste último esquema ofrece mayor control al programador sobre la planificación de su programa, por consiguiente es más propenso a las fallas, por lo que los S.O. multitarea modernos son *preentivos*. En los S.O. *preentivos*, una forma de determinar cuándo un hilo debe salir de ejecución es dándole un tiempo límite, al que se le llama *quantum*. Windows 95 y sus predecesores, así como Windows NT y sus predecesores, son sistemas operativos *preentivos* por *quantums*.

Cuando el *quantum* de un hilo vence, el planificador debe elegir cuál será el nuevo hilo a ejecutar. Uno de los elementos que se utiliza para esto es la prioridad del hilo. Todo hilo tiene una prioridad relacionada, la que está relacionada con la prioridad de su proceso. El planificador maneja una lista con prioridad de "listas de hilos listos", como se muestra en la figura 8.2. El planificador suele escoger un hilo dentro de la lista de mayor prioridad.

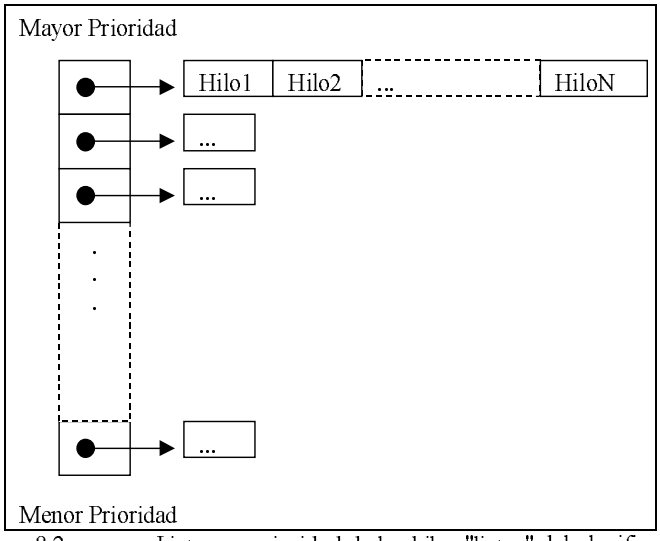

Figura 8.2. Lista con prioridad de los hilos "listos" del planificador

El trabajo con prioridades permite ejecutar más continuamente los hilos cuyos trabajos se consideran más prioritarios. Sin embargo, esto trae consigo dos problemas:

- **Bloqueos muertos** (*deadlocks*), ocurren cuando un hilo espera a que otro hilo, con menor prioridad, realice algo para poder continuar. Dado que el hilo esperado tiene menor prioridad que quien espera, nunca será elegido para ejecutarse, por lo que la condición nunca se cumplirá.

- **Condiciones de carrera** (*race conditions*), ocurren cuando un hilo acaba antes que otro, del cual depende para modificar los recursos compartidos, por lo que accede a valores errados.

Existen técnicas que los S.O. y las aplicaciones utilizan para determinar estas condiciones y evitarlas o solucionarlas.

Soporte para la Programación Concurrente

C y C++ no ofrecen soporte nativo (como parte de la especificación del lenguaje) para la programación concurrente, por lo que requieren hacer uso directo de las librerías que el sistema operativo subyacente provea para dicho fin.

Java y C# proveen soporte nativo para la programación concurrente, con librerías de clases para dicho fin que forman parte de la especificación del lenguaje. Ambos hacen uso de un conjunto reducido de características básicas de las librerías del S.O. subyacente. En el caso de Java, esto origina que sólo se permita la programación multihilos pero como contraparte se pueda utilizar las mismas librerías de clases y un comportamiento "bastante" similar desde distintos S.O. C# hace una implementación interna de la mayor parte de las características de la programación concurrente, como una forma de solucionar muchos de los problemas más comunes de esta programación sin modificar o depender del S.O.

Manejo de Hilos

En esta sección revisaremos las técnicas más simples de manejo de hilos vistas desde C/C++, Java y C#. No se verá el tema de la programación con procesos.

Creación

Una aplicación crea un hilo en Windows haciendo uso de la función CreateThread. El siguiente código en **C++** muestra el uso de esta función:

```
#include <windows.h>
#include <stdio.h>

DWORD __stdcall FuncionHilo(void* dwData) {
    char* DatosHilo = (char*) dwData;
    printf("Ejecutado hilo con mensaje %s\n", DatosHilo);
}

int main(int argc, char *argv[]) {
    for(int i = 0; i < 10; i++) {
        char* DatosHilo = new char[20];
        sprintf(DatosHilo, "Hilo-%d", i);
        DWORD dwID;
        HANDLE hHilo = CreateThread(
            NULL,          // dirección de una estructura SECURITY_ATTRIBUTES
            0,      // tamaño inicial de la pila del hilo
            FuncionHilo,   // punto de entrada del hilo
            (void*) DatosHilo,// parámetro del punto de entrada
            0,  // banderas con opciones. CREATE_SUSPENDED => ResumeThread
            &dwID   // recibe el identificador del hilo
            );
    }
    getchar();
    return 0;
}
```

Al igual que la función *main* o *WinMain* que son los puntos de entrada de nuestros programas, cuando se crea un hilo es necesario especificar una función de entrada, de forma que el S.O. sepa que cuando el hilo termina de ejecutar dicha función, finaliza. El ejemplo anterior crea 10 hilos con la misma función de entrada, *FuncionHilo*, pasándole como

parámetro una cadena de texto, cuarto parámetro de *CreateThread*, que es impresa por cada hilo al ejecutar *FuncionHilo*. Si bien en este ejemplo pasamos un único dato a la función de entrada, utilizando un puntero a una estructura o clase es posible pasar toda la información requerida a la función de entrada de un hilo. Esta función de entrada podrá tener cualquier nombre pero la misma firma (tipos del valor de retorno y los parámetros) que *FuncionHilo*. El significado de la palabra reservada __stdcall está relacionado con un requisito en la declaración de las funciones de entrada de los hilos del API de Windows. Su significado se verá más adelante.

El ejemplo anterior termina con un *getchar* al final de *main* debido a que, si *main* termina, el proceso termina. Eso se debe a que, si bien técnicamente un proceso no finaliza hasta que todos sus hilos finalizan, el hilo primario tiene un tratamiento especial en el sentido de que, si éste finaliza, se fuerza la finalización de los hilos secundarios y el proceso termina. Podemos probar esto quitando el *getchar* de main y corriendo el programa.

Un ejemplo igualmente sencillo en **Java** sería el siguiente:

```
class Hilo extends Thread {
    public Hilo( String nombre ) {
        super( nombre );
    }
    public void run() {
        System.out.println( "Hilo " + getName() + " ejecutado" );
    }
}

class Hilos0 {
    public static void main(String args[]) {
        for(int i = 0; i < 10; i++) {
            Hilo hilo = new Hilo("Hilo" + i);
            hilo.start();
        }
    }
}
```

En Java se hace uso de la clase *Thread*. Para crear un hilo se suele heredar de dicha clase y sobrescribir el método *run*, colocando dentro de él todo el código que deseamos que el hilo ejecute. El hilo se crea en estado "creado" y pasa al estado "listo" al llamar al método "start". El método *run* es el equivalente a la función de entrada del hilo en API de Windows, con la ventaja de ser un método, por lo que todos los datos que se hubiera deseado pasar a su equivalente en API de Windows, se pasan al crear el objeto de la clase derivada de *Thread* o bien llamando a sus métodos y guardándolos como datos miembros del objeto. Esto ofrece un enfoque más *orientado a objetos* para el manejo de hilos. Note que se llama a *super* en el constructor pasándole una cadena, lo que establece un nombre al objeto, lo que es útil para efectos de depuración. Dicho nombre se puede modificar y obtener con los métodos *setName* y *getName* respectivamente.

A diferencia del ejemplo de API de Windows, no se requiere de ningún artificio al final de *main* para evitar que el hilo primario finalice sin darle oportunidad a los secundarios a terminar su trabajo. Esto se debe a que cuando se retorna de *main* el intérprete de Java, quien es el que llama a *main*, se encarga de esperar a todos los hilos que fueron creados hasta que finalicen y recién allí finalizar el hilo primario y con él, el proceso.

En conclusión, el intérprete de Java realiza el siguiente algoritmo:

```
main(...);
if(hay_hilos_pendientes)
    joinAll();
return;
```

Un ejemplo igualmente sencillo en **C#** sería el siguiente:

```
using System;
using System.Threading;

class Hilos0 {
    public static void Ejecutar() {
        Thread hiloActual = Thread.CurrentThread;
        Console.WriteLine("Hilo " + hiloActual.Name + " ejecutado");
    }
    public static void Main() {
        for(int i = 0; i < 10; i++) {
            Thread hilo = new Thread(new ThreadStart(Ejecutar));
            hilo.Name = "Hilo" + i;
            hilo.Start();
        }
    }
}
```

En C# se utiliza la clase *Thread*, de la cual **no se puede heredar**, dado que ha sido declarada como *sealed*. Lo que se hace con ella es instanciarla pasándole como parámetro un objeto delegado que será utilizado para llamar a su método cuando el hilo se ejecute. El método relacionado con el objeto delegado hace el papel de la función de entrada del hilo. Este esquema ofrece la ventaja de evitar una herencia y poder hacer que cualquier método, estático o no, pertenecientes a la misma clase o a distintas, sean ejecutados por hilos distintos, todo esto sin perder el enfoque orientado a objetos.

Al igual que Java, existe código que se ejecuta después de finalizado el método Main y que realiza una espera hasta que todos los hilos secundarios finalicen, para luego finalizar el primario.

Clases para el Manejo de Hilos

C#

Las clases relacionadas con el manejo de hilos se encuentran en el espacio de nombres System.Threading. Dentro de ésta existen dos clases importantes: Thread y Monitor. La siguiente tabla muestra los principales miembros de estas clases:

Tabla 8.1. Métodos de la clase Thread

Clase Thread	
Prototipo	**Función**
Thread(ThreadStart)	Constructor que crea el hilo en estado no-iniciado, denominado "Unstarted". La documentación no indica si este estado corresponde a un hilo creado en estado "suspendido", o bien significa que el hilo aún no se ha creado, tan solo el objeto Thread que permitirá manejarlo.
Start()	Coloca el hilo en estado "listo", denominado "Running". Tanto un hilo "listo" como uno "en-ejecución" tienen en .NET el estado Running.
Abort()	Causa un ThreadAbortException en el hilo. La excepción puede ser capturada por un bloque catch en el código ejecutado por el hilo, pero al finalizar dicho bloque catch la excepción sera automáticamente relanzada. Debido a esto la ejecución inevitablemente terminará saliendo del método que es punto de entrada del hilo, por lo que el hilo finalizará, con estado "Stopped".Si nadie refiere al objeto Thread, será recolectado en algún momento.
Sleep(mlseg)	Coloca al hilo en estado de bloqueo, denominado "WaitSleepJoin". Es el S.O. quien lo saca luego de un tiempo. El parámetro *mlseg* especifica dicho tiempo en milisegundos.

Interrupt	Saca al hilo de un estado "WaitSleepJoin" y lo pone en estado "Running".
Join	Permiten que un hilo espere a que otro hilo finalice su ejecución.
Suspend	Permite que un hilo suspenda a otro hilo.
Resume	Permite que un hilo saque del estado de suspensión a otro hilo.

(*) El estado "WaitSleepJoin" agrupa a varias condiciones de bloqueo: La espera de un objeto de bloqueo que no sea un hilo, la espera a que un tiempo transcurra y la espera a que un hilo finalice.

Tabla 8.2. Métodos de la clase Monitor

Clase Monitor	
Prototipo	**Función**
Wait()	Coloca al hilo en estado "WaitSleepJoin"
Pulse() PulseAll()	Sacan del estado "WaitSleepJoin" a los hilos inactivos por dicho monitores.

Java

Todos los programas que se han mostrado como ejemplos hasta ahora han utilizado más de un hilo, dado que, en paralelo con el hilo principal que corre nuestro programa, el intérprete de Java dispara un hilo adicional, el *garbage collector*.

Sin embargo, a pesar de que uno de los objetivos principales del lenguaje Java es la portabilidad, la implementación de hilos en Java no llega a ser, hasta ahora, independiente de la plataforma. Esto se debe a las significativas diferencias entre los diversos sistemas operativos en la forma en que implementan la multitarea.

La Clase Thread

Es la clase base para la creación de hilos. Cuando se crea un hilo se define una nueva clase que deriva de *Thread*. Una vez que un nuevo objeto hilo ha sido creado, se utilizan los métodos heredados de *Thread* para administrarlo.

Cuando se instancia un objeto de una clase de hilo su estado es "creado" (llamado también "nacido"). Esto es equivalente a crear en C para Windows un hilo en estado suspendido.

Para que el hilo comience a correr, se debe de llamar a su método *start*, heredado de *Thread*. El método *start* arranca el hilo, lo que en su momento llama al método *run* de *Thread*, cuya implementación no hace nada dado que se espera que la clase derivada, en base a la que se creó el objeto hilo, la sobrescriba. Es en *run* donde se debe de colocar el código que debe de ejecutar las tareas del hilo. El método *run* es el equivalente a la función del hilo que se implementa en C para Windows.

Al igual que en C para Windows, mientras se ejecuta el método *run*, el hilo pasa del estado *listo* al estado *en ejecución* y nuevamente al estado *listo* repetidamente, hasta que finaliza este método, lo que coloca al hilo en estado *finalizado* (también llamado *muerto*).

Tabla 8.3. Métodos de la clase Thread

Métodos	**Descripción**
Thread(); Thread(String name);	Constructores. El parámetro *name* le da un nombre al hilo. El constructor por defecto asigna un nombre de forma automática.
String getName(); void setName(String name);	Permiten obtener y modificar el nombre de un objeto hilo.
int getPriority(); void setPriority(int newPriority);	Permiten obtener y modificar la prioridad de ejecución de un objeto hilo.

void start();	Coloca el estado del hilo en *listo* para ejecutarse.
void run();	Este método debe contener el código que ejecute las tareas del hilo.
static void sleep(long millis); static void sleep(long millis, int nanos);	Pone a dormir a un hilo. El parámetro *millis* especifica un tiempo en milisegundos. El parámetro *nanos* especifica un tiempo en nanosegundos.
static void yield();	Permite que un hilo voluntariamente renuncie a seguir siendo ejecutado si es que existe algún otro hilo en estado *listo*.

void interrupt();
static boolean interrupted();
boolean isInterrupted();
	Permiten manejar la interrupción de un hilo.

void join();
void join(long millis);
void join(long millis, int nanos);
	Permiten que un hilo espere a que otro hilo finalice su ejecución. Al igual que *sleep*, los parámetros permiten especificar un tiempo, en este caso, un tiempo máximo de espera. Si no se especifica un tiempo o éste es cero, se espera indefinidamente.

static Thread currentThread();	Permite obtener una referencia al hilo actual.
static void dumpStack();	Permite imprimir, en la salida estándar, un reporte de la pila de llamadas a métodos hasta el punto de ejecución actual.
String toString();	Permite obtener una descripción textual del hilo: Su nombre, su prioridad y el grupo al que pertenece.

Los métodos estáticos están pensados para ser llamados desde dentro de la ejecución del hilo, esto es, para realizar acciones sobre el hilo actual *en ejecución*. Estos métodos se pueden llamar desde *run* o desde alguno de los métodos (de la misma clase o de otras clases) que éste llame. Tome en cuenta que el método *main* es llamado por el método *run* del hilo primario.

El nombre de un hilo permite identificarlo dentro de un grupo de hilos, aunque dicha característica puede no ser utilizada. Java maneja el concepto de *grupo* de hilos. El manejo de los grupos de hilos así como sus semejanzas y diferencias con el concepto de proceso va más allá de los alcances del curso.

Todo hilo tiene una **prioridad**. Java maneja prioridades en el rango de *Thread.MIN_PRIORITY* (constante igual a 1) a *Thread.MAX_PRIORITY* (constante igual a 10). Por defecto, el hilo primario de un programa (el que llama al método *main* de nuestra clase ejecutable) se crea con la prioridad *Thread.MIN_NORMAL* (constante igual a 5). Cada nuevo objeto hilo que se cree hereda la prioridad del objeto hilo desde donde se instanció. Ahora bien, es importante recordar que Java se basa en las capacidades multitarea de la plataforma actual de ejecución, por lo que el efecto de la modificación de las prioridades sobre el orden de ejecución de los hilos es dependiente de la plataforma.

Un hilo *en ejecución* puede pasar a estar *bloqueado*, *dormido* o *en espera*. Estos estados pueden ser interrumpidos por el mismo sistema operativo o por otro hilo (método *interrupt*). Cuando estos estados se interrumpen, el hilo pasa al estado listo, de forma que pueda tomar alguna acción, como consecuencia de la interrupción, cuando el sistema operativo lo pase al estado *en ejecución*. El hilo guarda en un dato miembro interno el estado *interrumpido*, de forma que se pueda averiguar este hecho. Un hilo puede averiguar si fue interrumpido llamando el método *isInterrupted*. Este método no modifica el valor de este *flag*, a diferencia del método *interrupted*, que sí lo hace, colocando el *flag* a *false*. Esto significa que dos llamadas consecutivas a *interrupted*, luego de que el hilo fue interrumpido, devolverán *true* y *false* respectivamente, a menos que entre llamada y llamada se haya interrumpido nuevamente al hilo. Este *flag* puede servir como una forma de sincronizar el trabajo de un hilo con otro hilo.

Otra forma de sincronización de hilos es hacer que uno quede en estado de *espera* hasta que otro haya finalizado su trabajo, esto es, pase al estado *finalizado*. Esto se consigue mediante los métodos *join*.

Por último, el acceso a recursos compartidos y sincronización del trabajo entre los hilos se realiza mediante los objetos *monitores*. El uso de estos se verá mediante ejemplos.

Aspectos Generales de la Ejecución de un Hilo

Un hilo *en ejecución*, puede dejar de ejecutarse por diversos motivos:

- Si el hilo finaliza, entonces pasa al estado *finalizado*.
- El sistema operativo concede intervalos de tiempo a cada hilo en estado *listo* para que se ejecute, por lo tanto, si el intervalo de tiempo del hilo actual *en ejecución* vence, el hilo pasa a estado *listo*.
- Si el sistema operativo es *preentivo*, y determina que un hilo de mayor prioridad ha pasado al estado *listo*, entonces el hilo actual pasa al estado *listo* y el de mayor prioridad al estado *en ejecución*.
- El hilo queda en un estado de bloqueo. Casos comunes (y por lo mismo caracterizados con un nombre especial) son los estados *dormido* (sleep) y *en espera* (wait). Para el primer caso, cuando el sistema operativo advierte que el tiempo de dormir venció, pasa al hilo al estado *listo*. Para el segundo, un ejemplo es una operación de E/S a disco: Dado que no es el hilo quien realiza la lectura del disco, se bloquea esperando a que el sistema operativo complete la lectura solicitada. Cuando eso sucede, el sistema operativo desbloque al hilo.

Trabajo con Hilos en C++

Se debe seguir los siguientes pasos:

1. Definir una función de entrada del hilo.
2. Crear el hilo invocando a la función CreateThread.
3. Arrancar la ejecución de los hilos.

A continuación se muestra la forma de invocar a la función CreateThread y la especificación de sus parámetros:

```
HANDLE hHilo = CreateThread(NULL, 0, FuncionHilo, (void*)Nombre, 0, &dwID);
```

Tabla 8.4. Parámetros de la función CreateThread

Parámetro	Descripción
1er parámetro	Dirección de una estructura SECURITY_ATTRIBUTES
2do parámetro	Tamaño inicial de la pila del hilo
3er parámetro	Punto de entrada del hilo
4to parámetro	Parámetro del punto de entrada
5to parámetro	Banderas que determinan opciones. CREATE_SUSPENDED => ResumeThread
6to parámetro	Recibe el identificador del hilo

El siguiente programa muestra un ejemplo sencillo de la creación y uso de un hilo.

```
#include <windows.h>
#include <stdio.h>
#include <time.h>
```

```
struct DatosHilo {
    char Nombre[20];
    DatosHilo(int i) {
        sprintf(Nombre, "Hilo%d", i);
    }
};

DWORD __stdcall FuncionHilo(void* dwData) {
    DatosHilo* Datos = (DatosHilo*) dwData;
    printf("Hilo %s se va a dormir\n", Datos->Nombre);
    Sleep(rand() % 5000);
    printf("Hilo %s se desperto\n", Datos->Nombre);
    delete Datos;
    return 0;
}

int main(int argc, char *argv[]) {
    srand( (unsigned)time( NULL ) );
    DWORD dwID;
    HANDLE hHilo1, hHilo2, hHilo3, hHilo4;

    hHilo1= CreateThread(NULL, 0, FuncionHilo, new DatosHilo(1), CREATE_SUSPENDED, &dwID);
    hHilo2= CreateThread(NULL, 0, FuncionHilo, new DatosHilo(2), CREATE_SUSPENDED, &dwID);
    hHilo3= CreateThread(NULL, 0, FuncionHilo, new DatosHilo(3), CREATE_SUSPENDED, &dwID);
    hHilo4= CreateThread(NULL, 0, FuncionHilo, new DatosHilo(4), CREATE_SUSPENDED, &dwID);

    printf("Arrancando los hilos ...\n");
    ResumeThread(hHilo1);
    ResumeThread(hHilo2);
    ResumeThread(hHilo3);
    ResumeThread(hHilo4);

    printf("Hilos arrancados. Fin de main.\n");
    getchar();
    return 0;
}
```

Al correr el programa la salida será parecida a de la figura 8.3. Note que el hilo primario, correspondiente al método *main*, termina su trabajo antes de que cualquier hilo comience a ejecutarse. Sin embargo, esto no necesariamente es así.

Note que, a diferencia de Java y C# (como se verá más adelante) la "aparente" finalización del hilo primario sí finaliza el programa. Esto se debe a que en C++ los programas generados no agregan ningún código especial que permita al hilo primario esperar hasta que todos los secundarios hayan finalizado. Si se desea hacer esto tendrá que agregar a la función main (o a funciones llamadas desde ésta) código explicito para esta tarea.

Si bien en la figura 8.3 se muestra que los hilos inician su trabajo con cierto orden, mostrando su mensaje, esto no necesariamente es así, por lo que no podemos asumir que el orden en que se arrancan los hilos será el orden en que comiencen a ejecutarse. En general, el orden de ejecución de los hilos es una decisión del sistema operativo en base a las políticas que implemente su *planificador*.

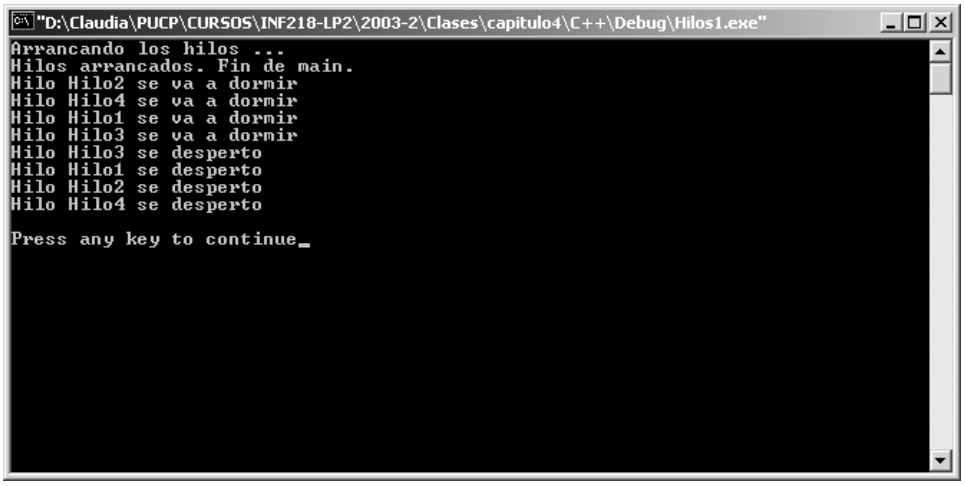

Figura 8.3. Salida del Ejemplo de Hilos en C++

Sincronización

En C++ utilizaremos *secciones críticas* para la sincronización de hilos. La *sección crítica* se define dentro de una función delimitando las instrucciones que se quiere que solamente un hilo a la vez las ejecute.

Cuando un grupo de instrucciones están delimitadas por una *sección crítica* y son ejecutadas desde un hilo, ningún otro hilo puede acceder a éstas. Cuando un hilo entra a ejecutar una *sección crítica*, ésta queda bloqueada. Esto implica que si un hilo intenta ejecutar esta *sección crítica*, quedará en estado de *espera* hasta que ésta se desbloquee. Cuando un hilo sale de la *sección crítica*, ésta se desbloquea, permitiendo que otros hilos accedan a ella. Si hubiese algún hilo *esperando* una llamada pendiente a la *sección crítica*, el hilo pasa al estado *listo*, de forma que cuando entre al estado *en ejecución* pueda ejecutar el código de la *sección crítica*.

El siguiente ejemplo muestra el uso de una sección crítica

```
#include <windows.h>
#include <stdio.h>
#include <time.h>

class Productor {
    int Trabajo;
    CRITICAL_SECTION cs;
public:
    Productor() {
        Trabajo = 0;
        InitializeCriticalSection(&cs);
    }
    ~Productor() {
        DeleteCriticalSection(&cs);
    }
    int SiguienteTrabajo() {
        EnterCriticalSection(&cs);
        int Nuevo = Trabajo++;
        Sleep( rand() % 1000 );
```

```
        LeaveCriticalSection(&cs);
        return Nuevo;
    }
};

class DatosConsumidor {
    char Nombre[20];
    Productor* p;
public:
    DatosConsumidor(char* Nombre, Productor* p) {
        strcpy(this->Nombre, Nombre);
        this->p = p;
    }
    void Ejecutar() {
        for( int i = 0; i < 5; i++ ) {
            int iTrabajo = p->SiguienteTrabajo();
            printf("%s: Obtenido trabajo %d\n", Nombre, iTrabajo);
            Sleep(rand() % 1000);
            printf("%s: Trabajo %d completado\n", Nombre, iTrabajo);
        }
    }
};

DWORD __stdcall FuncionHilo(void* dwData) {
    DatosConsumidor* Datos = (DatosConsumidor*) dwData;
    Datos->Ejecutar();
    delete Datos;
    return 0;
}
int main(int argc, char *argv[]) {
    srand( (unsigned)time( NULL ) );

    DWORD dwID;
    Productor* p = new Productor();
    CreateThread(NULL, 0, FuncionHilo, new DatosConsumidor("Hilo1", p), 0, &dwID);
    CreateThread(NULL, 0, FuncionHilo, new DatosConsumidor("Hilo2", p), 0, &dwID);

    getchar();
    //delete p;// este delete es riesgoso, hay un problema de sincronización
    return 0;
}
```

Este programa, cuya ejecución se muestra en la figura 8.4, simula un proceso "productor-consumidor". El productor devuelve un trabajo a demanda del consumidor que se lo solicita. Cuando el productor calcula un nuevo trabajo le toma un tiempo aleatorio entregarlo. Sin embargo, dado que la función de entrega de un trabajo, *SiguienteTrabajo*, cuenta con una sección crítica delimitando sus instrucciones, sólo un hilo podrá obtener un trabajo a la vez, por lo que no habrá pérdida de trabajos.

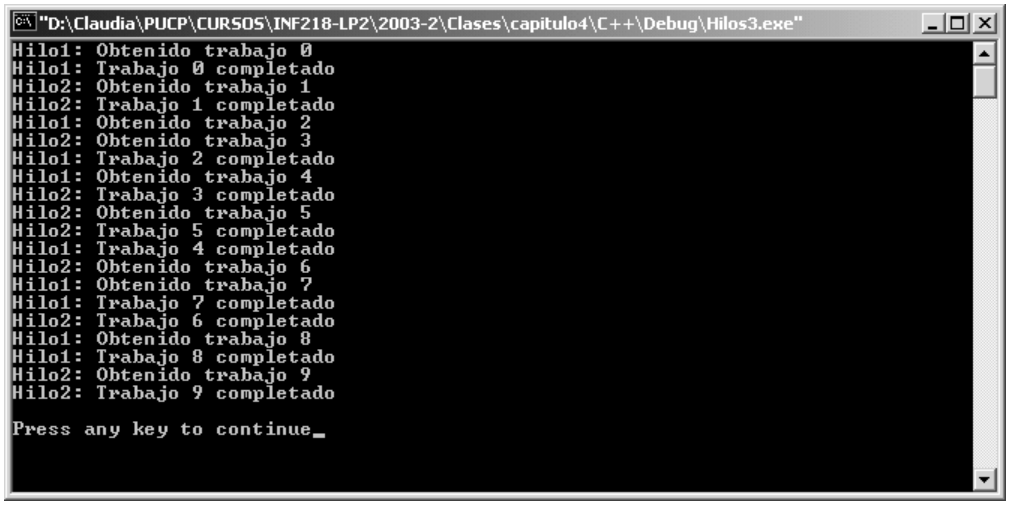

Figura 8.4. Salida del Ejemplo de Sincronización de Hilos en C++

Para ejemplificar mejor esta sincronización, la salida de la figura 8.5 muestra lo que podría ocurrir si la función *SiguienteTrabajo* no se sincronizara.

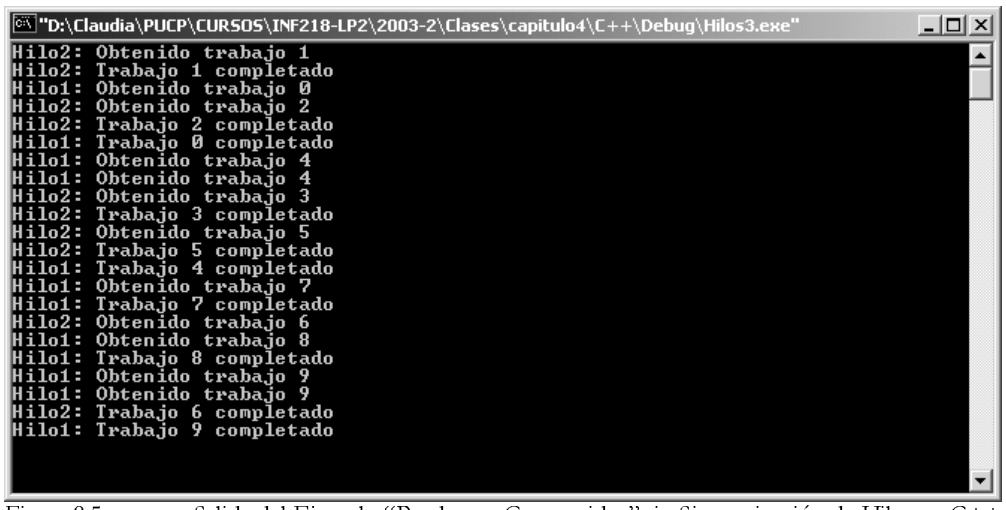

Figura 8.5. Salida del Ejemplo "Productor-Consumidor" sin Sincronización de Hilos en C++

El siguiente ejemplo muestra el uso de la función "WaitForSingleObject".

```
#include <windows.h>
#include <stdio.h>
#include <time.h>

class DatosHilo {
    bool Finalizar;
public:
    DatosHilo() {
        Finalizar = false;
    }
```

```
    void Finaliza() {
        Finalizar = true;
    }
    void Ejecutar() {
        int Trabajo = 0;
        while( !Finalizar ) {
            Trabajo++;
            printf("Hilo: Inicio de trabajo %d\n", Trabajo );
            Sleep(rand() % 3000);
            printf("Hilo: Finalizo trabajo %d\n", Trabajo );
        }
    }
};

DWORD __stdcall FuncionHilo(void* dwData) {
    DatosHilo* Datos = (DatosHilo*) dwData;
    Datos->Ejecutar();
    delete Datos;
    return 0;
}

int main(int argc, char *argv[]) {
    srand( (unsigned)time( NULL ) );

    DWORD dwID;
    DatosHilo* Datos = new DatosHilo();
    HANDLE hHilo = CreateThread(NULL, 0, FuncionHilo, Datos, 0, &dwID);

    printf("Hilo primario se va a dormir ...\n" );
    Sleep(rand() % 3000);
    printf("Hilo primario se despierta. Esperando a que finalice el hilo "
        "secundario ...\n");
    Datos->Finaliza();
    WaitForSingleObject(hHilo, INFINITE);
    printf("Función main finalizó\n");
    getchar();
    return 0;
}
```

A continuación en la figura 8.6 se muestra la salida del programa. Note que el hilo primario, el que ejecuta al método main, espera a que el nuevo hilo creado finalice para continuar.

Figura 8.6. Salida del Ejemplo del uso de la función WaitForSingleObject en C++

Trabajo con Hilos en C#

Las clases para hilos están en el espacio de nombres **System.Threading**. La clase *Thread* es *sealed*, es decir, nadie puede heredar de ella.

El procedimiento para crear un hilo es:

1. Crear un objeto delegado ThreadStart, pasándole como parámetro el método que ejecutará el hilo.
2. Crear un objeto Thread, pasándole como parámetro el objeto delegado.
3. Iniciar la ejecución del hilo llamando al método Start.

El siguiente programa muestra un ejemplo sencillo de creación y uso de un hilo.

```
using System;
using System.Threading;

class DatosHilo {
    private static Random r = new Random();
    public void Ejecutar() {
        Thread HiloActual = Thread.CurrentThread;
        int tiempo = r.Next(5000);
        Console.WriteLine("Hilo " + HiloActual.Name + " se va a dormir");
        Thread.Sleep(tiempo);
        Console.WriteLine("Hilo " + HiloActual.Name + " se desperto" );
    }
}

class Hilos1 {
    public static void Main() {
```

```
            DatosHilo datos1, datos2, datos3, datos4;
            Thread hilo1, hilo2, hilo3, hilo4;
            datos1 = new DatosHilo();
            datos2 = new DatosHilo();
            datos3 = new DatosHilo();
            datos4 = new DatosHilo();
            hilo1 = new Thread(new ThreadStart(datos1.Ejecutar));
            hilo2 = new Thread(new ThreadStart(datos2.Ejecutar));
            hilo3 = new Thread(new ThreadStart(datos3.Ejecutar));
            hilo4 = new Thread(new ThreadStart(datos4.Ejecutar));
            hilo1.Name = "Hilo1";
            hilo2.Name = "Hilo2";
            hilo3.Name = "Hilo3";
            hilo4.Name = "Hilo4";
            Console.WriteLine( "Arrancando los hilos ..." );
            hilo1.Start();
            hilo2.Start();
            hilo3.Start();
            hilo4.Start();
            Console.WriteLine( "Hilos arrancados. Fin de main.\n" );
        }
    }
```

Al correr el programa la salida será parecida a de la figura 8.7. Note que el hilo primario, correspondiente al método *main*, termina su trabajo antes de que cualquier hilo comience a ejecutarse. Sin embargo, esto no necesariamente es así.

Note que la finalización del hilo primario no finaliza el programa. El programa finaliza cuando todos los hilos creados finalizan.

Si bien en la figura 8.7 se muestra que los hilos inician su trabajo con cierto orden, mostrando su mensaje, esto no necesariamente es así, por lo que no podemos asumir que el orden en que se arrancan los hilos será el orden en que comiencen a ejecutarse. En general, el orden de ejecución de los hilos es una decisión del sistema operativo en base a las políticas que implemente su *planificador*.

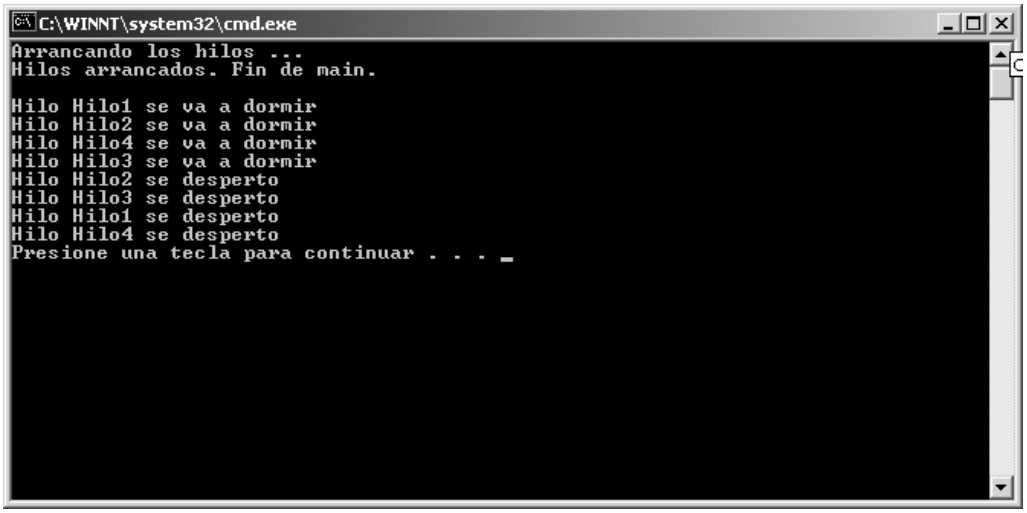

Figura 8.7. Salida del Ejemplo de Hilos en C#

Como se explicó, un hilo *bloqueado*, *dormido* o *esperando* puede ser interrumpido y sacado de este estado. Debido a esto, las llamadas a los métodos que producen estos estados disparan excepciones adecuadas para los casos en que estos estados son interrumpidos. El manejo de estos cambios de estado y otras acciones con hilos utilizan ampliamente las excepciones para su control. En el caso del método Interrupt se produce la excepción System.Threading.ThreadInterruptedException.

El siguiente programa muestra la interrupción de los hilos.

```
using System;
using System.Threading;

class DatosHilo {
    private static Random r = new Random();
    private Thread hilo;
    public Thread Hilo {
        get { return hilo; }
    }
    public DatosHilo(string Nombre) {
        hilo = new Thread(new ThreadStart(Ejecutar));
        hilo.Name = Nombre;
        Console.WriteLine( "Hilo " + hilo.Name + " creado" );
    }
    public void Ejecutar() {
        int Tiempo = r.Next(5000);
        Console.WriteLine("Hilo " + hilo.Name + " se va a dormir");
        Thread.Sleep(Tiempo);
        Console.WriteLine("Hilo " + hilo.Name + " se desperto");
    }
}

class Hilos2 {
    public static void Main() {
        DatosHilo datos1, datos2, datos3, datos4;
        datos1 = new DatosHilo( "Hilo1" );
        datos2 = new DatosHilo( "Hilo2" );
        datos3 = new DatosHilo( "Hilo3" );
        datos4 = new DatosHilo( "Hilo4" );
        Console.WriteLine("Arrancando los hilos ...");
        datos1.Hilo.Start();
        datos2.Hilo.Start();
        datos3.Hilo.Start();
        datos4.Hilo.Start();
        Console.WriteLine("Hilos arrancados. El hilo primario se va a dormir.\n");
        Thread.Sleep( 2500 );
        Console.WriteLine("Interrunpiendo los hilos ...");
        datos1.Hilo.Interrupt();
        datos2.Hilo.Interrupt();
        datos3.Hilo.Interrupt();
        datos4.Hilo.Interrupt();
        Console.WriteLine("\nHilos interrumpidos. Fin de main.\n");
        Console.ReadLine();
    }
}
```

Al correr el programa la salida será parecida a la mostrada en la figura 8.8.

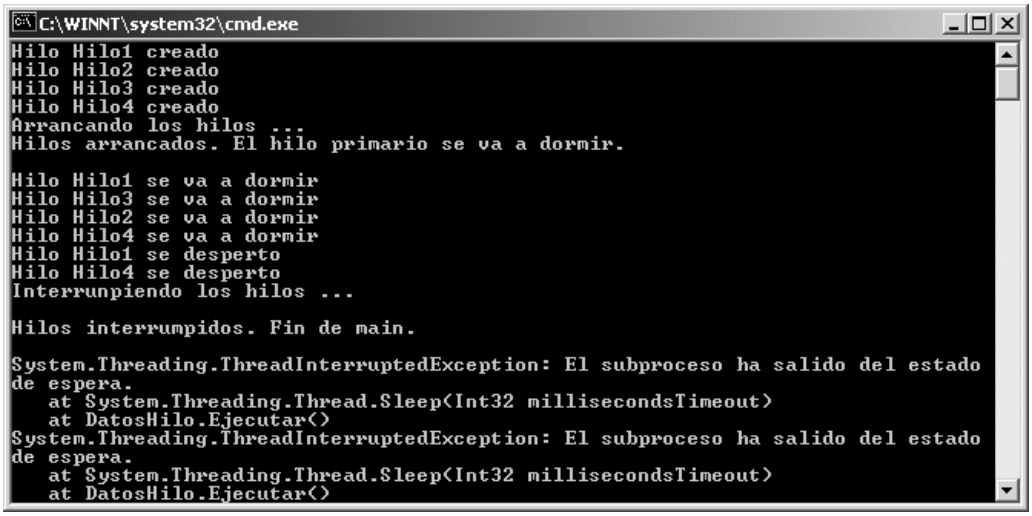

Figura 8.8. Salida del Ejemplo de Interrupción de Hilos en C#

Sincronización

C# utiliza la clase Monitor para la sincronización de hilos. La clase Monitor mediante sus métodos estáticos delimita una sección crítica. Esto implica que en cualquier método se puede definir una zona crítica.

Cuando un grupo de instrucciones están delimitadas por una *sección crítica* y son ejecutadas desde un hilo, ningún otro hilo puede acceder a éstas. Cuando un hilo entra a ejecutar una *sección crítica*, ésta queda bloqueada. Esto implica que si un hilo intenta ejecutar esta *sección crítica*, quedará en estado de *espera* hasta que ésta se desbloquee. Cuando un hilo sale de la *sección crítica*, ésta se desbloquea, permitiendo que otros hilos accedan a ella. Si hubiese algún hilo *esperando* una llamada pendiente a la *sección crítica*, el hilo pasa al estado *listo*, de forma que cuando entre al estado *en ejecución* pueda ejecutar el código de la *sección crítica*.

El siguiente ejemplo muestra el uso de una sección crítica

```
using System;
using System.Threading;

class Productor {
    public static Random Rand = new Random();
    private int Trabajo = 0;

    public int SiguienteTrabajo() {
        Monitor.Enter(this);
        Trabajo++;
        Thread.Sleep(Rand.Next(1000));
        Monitor.Exit(this);
        return Trabajo;
    }
}

class DatosConsumidor {
    public readonly Thread Hilo;
    private Productor P;
```

```
    public DatosConsumidor(string Nombre, Productor P) {
        this.P = P;
        Hilo = new Thread(new ThreadStart(Ejecutar));
        Hilo.Name = Nombre;
    }
    public void Ejecutar() {
        for( int i = 0; i < 5; i++ ) {
            int Trabajo = P.SiguienteTrabajo();
            Console.WriteLine(Hilo.Name + ": Obtenido trabajo " + Trabajo);
            Thread.Sleep(Productor.Rand.Next(1000));
            Console.WriteLine(Hilo.Name + ": Trabajo "  + Trabajo + " completado" );
        }
    }
}

public class Hilos3 {
    public static void Main() {
        Productor p = new Productor();
        DatosConsumidor datos1 = new DatosConsumidor("Hilo1", p);
        DatosConsumidor datos2 = new DatosConsumidor("Hilo2", p);
        datos1.Hilo.Start();
        datos2.Hilo.Start();
    }
}
```

Este programa, cuya ejecución se muestra en la figura 8.9, simula un proceso "productor-consumidor". El productor devuelve un trabajo a demanda al consumidor que se lo solicita. Cuando el productor calcula un nuevo trabajo le toma un tiempo aleatorio entregarlo. Sin embargo, dado que la función de entrega de un trabajo, *SiguienteTrabajo*, cuenta con una sección crítica delimitando sus instrucciones, sólo un hilo podrá obtener un trabajo a la vez, por lo que no habrá pérdida de trabajos.

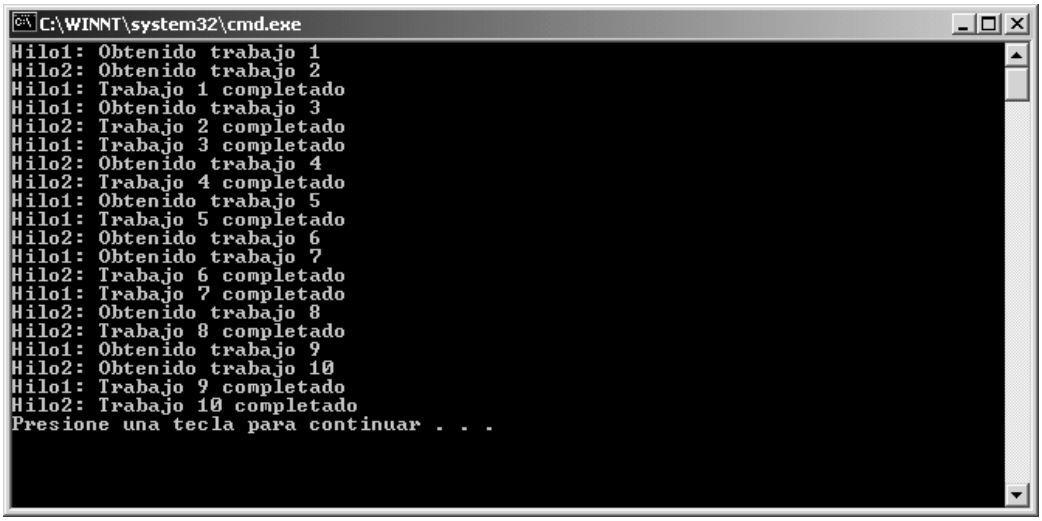

Figura 8.9. Salida del Ejemplo de Sincronización de Hilos en C#

Ahora bien, que pasaría en el caso de tener un productor que produce indiferentemente de que existan consumidores que consuman dichos trabajos, y a la vez consumidores que consuman indiferentemente de que exista algo que consumir. Si el

productor elimina un trabajo para crear uno nuevo antes de que el anterior sea consumido, se perderá dicho trabajo. Si el consumidor consume un trabajo antes de que se haya producido uno nuevo, un trabajo será realizado dos o más veces. En este caso, ambas labores, la de producir y la de consumir deben de sincronizarse una con otra, esto es, el productor no debe seguir produciendo mientras que no se haya consumido el trabajo anterior y el consumidor debe esperar a que se produzca un nuevo trabajo antes de consumirlo.

En este tipo de situación, la sincronización utilizando la clase Monitor por sí sola no es suficiente. Se necesita que el hilo productor espere y notifique al consumidor, y éste a su vez espere y notifique al productor.

Para esto, la clase Monitor provee los método *Wait*, *Pulse* y *PulseAll*. El método *Wait* coloca al hilo en estado de *espera* y desbloquea al monitor. El hilo saldrá de dicho estado cuando otro hilo acceda y llame a *Pulse* o *PulseAll*. Es ese momento el hilo será colocado en estado *listo* y competirá con el resto de hilos que intentan acceder a algún sección crítica de algún método (o que también hayan salido del estado de *espera*) para bloquearlo y ejecutar su código.

El método *Pulse* saca al primero de los hilos que esté en la lista de espera, del estado *esperando* y lo coloca en estado *listo*. De esta forma el hilo vuelve a entrar a competir por el acceso al objeto monitor.

El método *PulseAll* saca a todos los hilos que esté en la lista de espera, del estado *esperando* y los coloca en estado *listo*.

El siguiente ejemplo muestra el uso de los métodos *Wait* y *Pulse*.

```
using System;
using System.Threading;

class Trabajo {
    private int NroTrabajo = -1;
    private bool PuedoCrear = true;  // determina si se puede crear o consumir

    public void CrearNuevo( int NroTrabajo ) {
        Monitor.Enter(this);
        while ( !PuedoCrear ) {  // aún no se puede producir
            try {
                Monitor.Wait(this);
            }
            catch ( Exception ) {
            }
        }

        Console.WriteLine(Thread.CurrentThread.Name+" creo el trabajo " + NroTrabajo );
        this.NroTrabajo = NroTrabajo;
        PuedoCrear = false;
        Monitor.Pulse(this);// notifico que existe un trabajo listo
        Monitor.Exit(this);
    }

    public int ObtenerTrabajo() {
        Monitor.Enter(this);
        while ( PuedoCrear ) {   // aún no se puede obtener
            try {
                Monitor.Wait(this);
            }
            catch ( Exception) {
            }
        }
```

```
            PuedoCrear = true;
            Monitor.Pulse(this);  // notifico que ya se obtuvo el trabajo actual
            Console.WriteLine( Thread.CurrentThread.Name + " obtuvo el trabajo " +
                NroTrabajo );
            Monitor.Exit(this);
            return NroTrabajo;
        }
    }
    class Productor{
        private Trabajo t;
        public Thread Hilo;
        public Productor( Trabajo t ) {
            Console.WriteLine("Productor");
            this.t = t;
            Hilo = new Thread(new ThreadStart(run));
            Hilo.Name = "Productor";
        }

        public void run() {
            for ( int Contador = 1; Contador <= 5; Contador++ ) {
                // simulo un tiempo de demora aleatorio
                // en la creación del nuevo trabajo
                Random random = new Random();
                try {
                    Thread.Sleep(random.Next(1000));
                }
                catch( Exception ) {
                }

                t.CrearNuevo( Contador );
            }

            Console.WriteLine( Hilo.Name + " finalizo la produccion de trabajos" );
        }
    }

    class Consumidor {
        private Trabajo t;
        public Thread Hilo;

        public Consumidor ( Trabajo t ) {
            this.t = t;
            Hilo = new Thread(new ThreadStart(run));
            Hilo.Name = "Consumidor";
            Console.WriteLine("Consumidor");
        }

        public void run() {
            int iTrabajo;
            do {
                iTrabajo = t.ObtenerTrabajo();

                // simulo un tiempo de demora aleatorio
                // en el procesamiento del trabajo
                Random random = new Random();
                try {
                    Thread.Sleep(random.Next(1000));
                }
                catch( Exception) {
                }
            } while ( iTrabajo != 5 );
```

```
            Console.WriteLine( Hilo.Name + " termino de procesar los trabajos" );
        }
    }
    public class Hilos4 {
        public static void Main( String []args) {
            Trabajo t = new Trabajo();
            Productor p = new Productor( t );
            Consumidor c = new Consumidor( t );

            p.Hilo.Start();
            c.Hilo.Start();
            Console.ReadLine();
        }
    }
```

Al correr el programa la salida mostrada en la figura 8.10

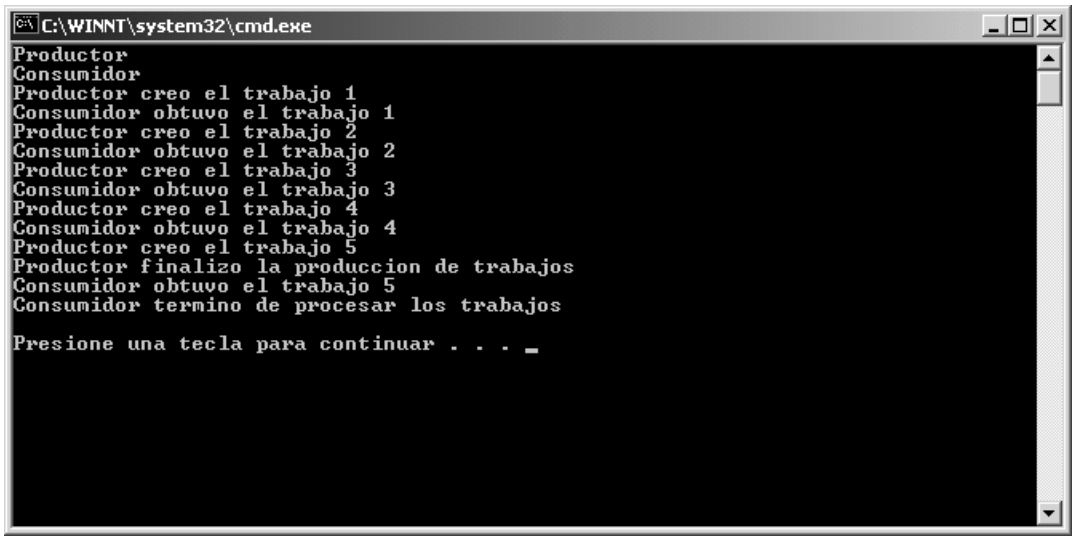

Figura 8.10. Salida del Ejemplo de Sincronización con Wait y Pulse en C#

También es posible hacer que un hilo quede bloqueado hasta que otro hilo pase a estado "finalizado". Para esta labor se utiliza el método *join*. El siguiente ejemplo muestra el uso de dicho método.

```
using System;
using System.Threading;

class DatosHilo {
    public static Random Rand = new Random();
    public readonly Thread Hilo;
    private bool Finalizar = false;

    public DatosHilo(string Nombre) {
        Hilo = new Thread(new ThreadStart(Ejecutar));
        Hilo.Name = Nombre;
    }
    public void Finaliza() {
        Finalizar = true;
```

```
        }
        public void Ejecutar() {
            int Trabajo = 0;
            while( !Finalizar ) {
                Trabajo++;
                Console.WriteLine(Hilo.Name + ": Inicio de trabajo " + Trabajo );
                Thread.Sleep(Rand.Next(3000));
                Console.WriteLine(Hilo.Name + ": Finalizo trabajo " + Trabajo );
            }
            Console.WriteLine(Hilo.Name + ": Finalizo el hilo");
        }
    }

    public class Hilos5 {
        public static void Main() {
            DatosHilo datos = new DatosHilo("Hilo2");
            datos.Hilo.Start();
            Console.WriteLine("Hilo primario se va a dormir ...");
            Thread.Sleep(DatosHilo.Rand.Next(3000));

            Console.WriteLine(
                "Hilo primario se despierta. Esperando a que finalice el hilo secundario ...");
            datos.Finaliza();
            datos.Hilo.Join();

            Console.WriteLine("Metodo main finalizo");
        }
    }
```

Al ejecutar este programa se obtendrá una salida como la que se muestra en la figura 8.11

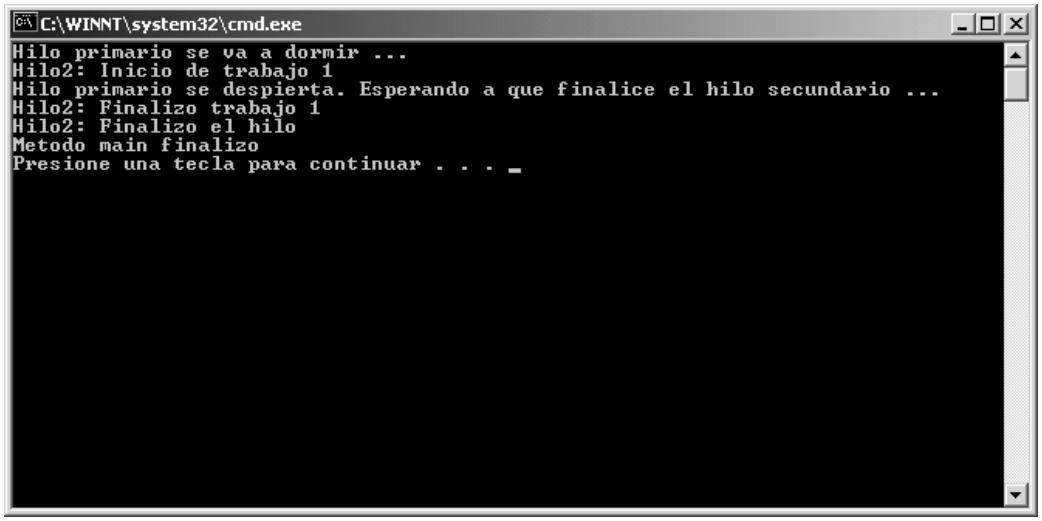

Figura 8.11. Salida del Ejemplo de Join en C#

Trabajo con Hilos en Java

Se debe seguir los siguientes pasos:

1. Definir una nueva clase que derive de *Thread*.
2. Sobrescribir el método *run*.
3. Crear e inicializar variables de la nueva clase de hilo.
4. Arrancar la ejecución de los hilos llamando al método *start*.

El siguiente programa muestra un ejemplo sencillo de creación y uso de un hilo.

```
class MiHilo extends Thread
{
    private int sleepTime;

    // El constructor asigna un nombre al hilo
    // llamando al constructor apropiado de Thread
    public MiHilo( String name )
    {
        super( name );
        // Calculo un tiempo aleatorio para dormir entre 0 y 5 segundos
        sleepTime = (int) ( Math.random() * 5000 );
        // Imprimo en la salida estándar los datos
        // del nuevo hilo creado
        System.out.println( "Nombre: " + getName() +
                  ";  tiempo a dormir: " + sleepTime );
    }

    // Código de ejecución del hilo
    public void run()
    {
        System.out.println( getName() + " going to sleep" );

        // pongo a dormir el hilo
        try
        {
            sleep( sleepTime );
        }
        catch ( InterruptedException exception )
        {
            System.out.println( exception.toString() );
        }

        // Indico que el hilo finalizo su trabajo
        System.out.println( getName() + " done sleeping" );
    }
}

public class Ejemplo1
{
```

```
    public static void main( String args[] )
    {
       MiHilo hilo1, hilo2, hilo3, hilo4;

       hilo1 = new MiHilo( "Hilo1" );
       hilo2 = new MiHilo( "Hilo2" );
       hilo3 = new MiHilo( "Hilo3" );
       hilo4 = new MiHilo( "Hilo4" );

       System.out.println( "Arrancando los hilos ..." );

       hilo1.start();
       hilo2.start();
       hilo3.start();
       hilo4.start();

       System.out.println( "Hilos arrancados\n" );
    }
}
```

Al correr el programa la salida será parecida a la mostrada en la figura 8.12.

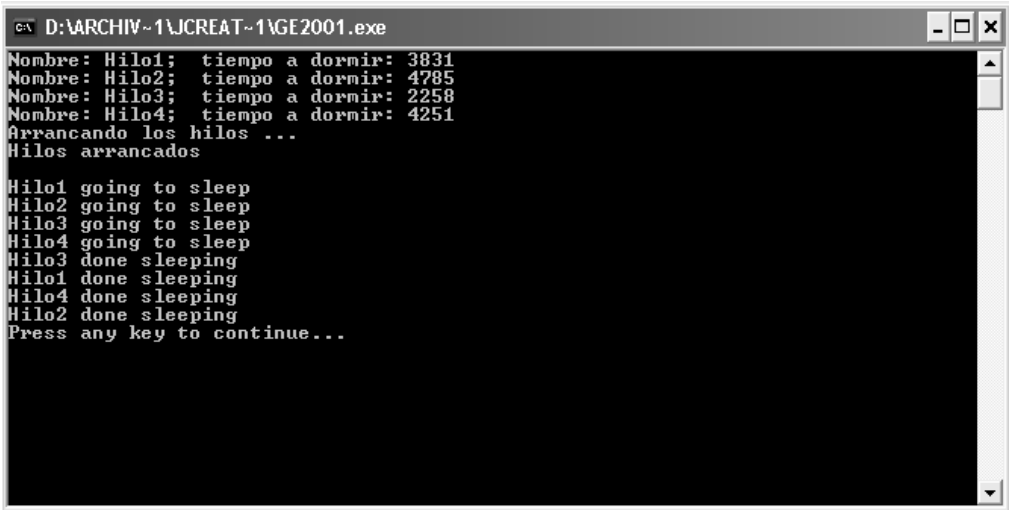

Figura 8.12. Salida del Ejemplo de Hilos en Java

Note que el hilo primario, correspondiente al método *main*, termina su trabajo antes de que cualquier hilo comience a ejecutarse. Sin embargo, esto no necesariamente es así. Otra ejecución podría arrojar lo mostrado en la figura 8.13.

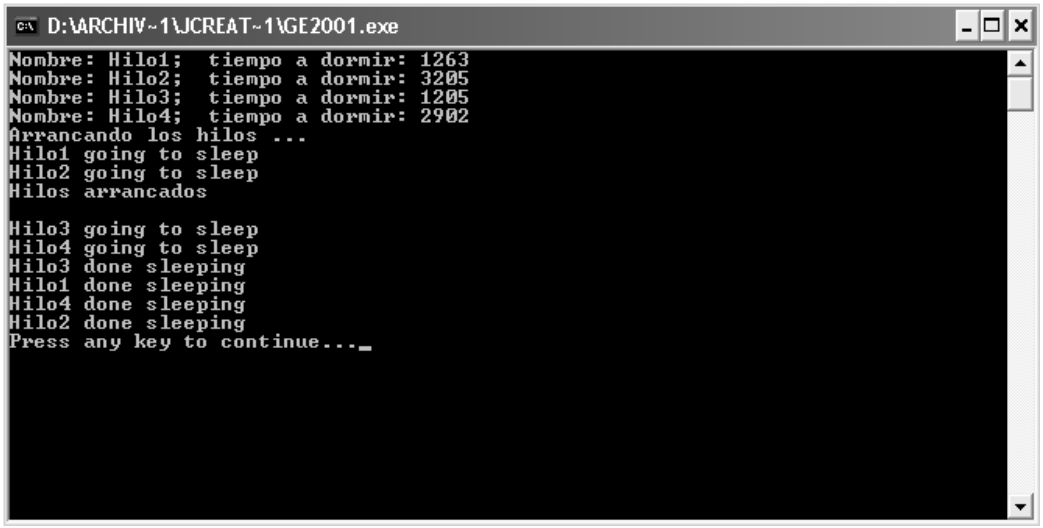

Figura 8.13. Segunda Salida del Ejemplo de Hilos en Java

Note que la finalización del hilo primario no finaliza el programa. El programa finaliza cuando todos los hilos creados finalizan.

Si bien en ambos ejemplos los hilos inician su trabajo en orden, mostrando su mensaje *"going to sleep"*, esto no necesariamente es así, por lo que no podemos asumir que el orden en que se arrancan los hilos será el orden en que comiencen a ejecutarse. En general, el orden de ejecución de los hilos es una decisión del sistema operativo en base a las políticas que implemente su *planificador*.

Note que la llamada al método *sleep* se coloca dentro de un bloque *try*, dado que este método puede disparar la excepción *InterruptedException*. Como se explicó en la sección anterior, un hilo *bloqueado*, *dormido* o *esperando* puede ser interrumpido y sacado de este estado. Debido a esto, las llamadas a los métodos que producen estos estados disparan excepciones adecuadas para los casos en que estos estados son interrumpidos. El manejo de estos cambios de estado y otras acciones con hilos utilizan ampliamente las interrupciones para su control. Estas excepciones son *No-Runtime*, por lo que el no manejarlas (con bloques *try* o indicándolas en la lista *throws* del método) provocarían un error en tiempo de compilación.

El siguiente programa muestra la interrupción de los hilos.

```
class MiHilo extends Thread
{
    private int sleepTime;

    // El constructor asigna un nombre al hilo
    // llamando al constructor apropiado de Thread
    public MiHilo( String name )
    {
        super( name );

        // Calculo un tiempo aleatorio para dormir entre 0 y 5 segundos
        sleepTime = (int) ( Math.random() * 5000 );
        // Imprimo en la salida estándar los datos
        // del nuevo hilo creado
```

```
            System.out.println( "Nombre: " + getName() +
                      ";  tiempo a dormir: " + sleepTime );
        }

        // Código de ejecución del hilo
        public void run()
        {
            // pongo a dormir el hilo
            try
            {
                System.out.println( getName() + " going to sleep" );

                Thread.sleep( sleepTime );

                // Indico que el hilo finalizo su trabajo satisfactoriamente
                System.out.println( getName() + " done sleeping" );
            }
            catch ( InterruptedException exception )
            {
                System.out.println( exception.toString() );
            }

        }
    }

    public class Ejemplo2
    {
        public static void main( String args[] ) throws InterruptedException
        {
            MiHilo hilo1, hilo2, hilo3, hilo4;

            hilo1 = new MiHilo( "Hilo1" );
            hilo2 = new MiHilo( "Hilo2" );
            hilo3 = new MiHilo( "Hilo3" );
            hilo4 = new MiHilo( "Hilo4" );

            System.out.println( "\nArrancando los hilos ...\n" );

            hilo1.start();
            hilo2.start();
            hilo3.start();
            hilo4.start();

            System.out.println( "\nHilos arrancados\n" );
            // Les doy un plazo de tiempo para que completen su trabajo
            Thread hiloPrimario = Thread.currentThread();
            hiloPrimario.sleep( 2500 );

            // Interrumpo los hilos
            hilo1.interrupt();
            hilo2.interrupt();
            hilo3.interrupt();
            hilo4.interrupt();
            System.out.println( "\nHilos interrumpidos\n" );
        }
    }
```

Al correr el programa la salida será la mostrada en la figura 8.14.

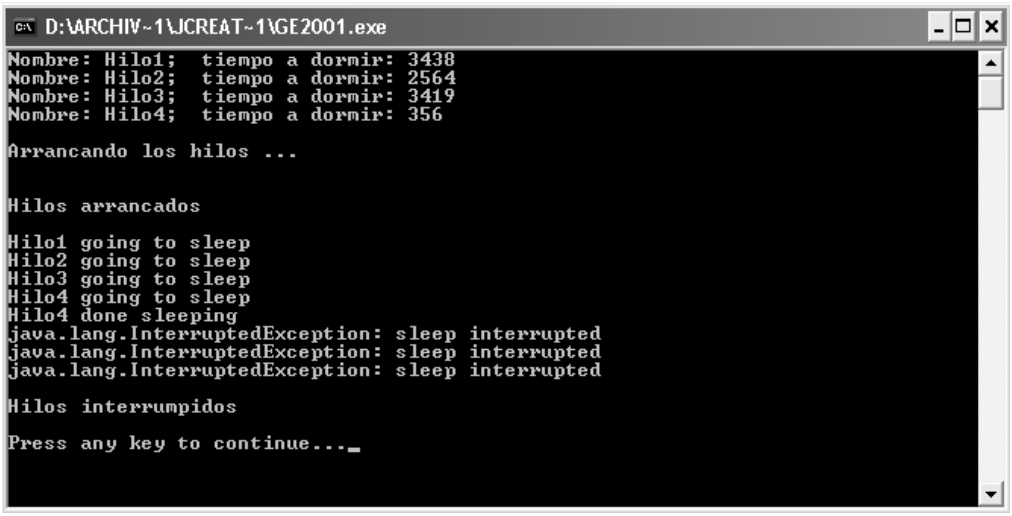

Figura 8.14. Segunda Salida del Ejemplo de Hilos en Java

Note que sólo el hilo "Hilo4" logra finalizar su trabajo antes de que sea interrumpido, los demás hilos son interrumpidos antes. En este caso en particular, dado que no se crea ningún recurso dentro del bloque *try* que no pueda ser liberado automáticamente por el intérprete (o alguna otra acción importante), no es necesario definir un bloque *finally* que se encargue de ello.

Note que el método *main* obtiene una referencia al hilo que lo ejecuta utilizando el método estático *currentThread* de la clase *Thread*. Con esta referencia llama al método *sleep*. Sin embargo, dado que este método es estático, también se pudo utilizar:

```
Thread.sleep( 2500 );
```

El tiempo que se pasa como parámetro al método *sleep*, y a otros métodos de *Thread* que requieran una especificación de tiempo, se dan en milisegundos.

Note que el método *main* no desea manejar la interrupción que pueda disparar *sleep*, por lo que declara dicha interrupción en la lista *throws* de su declaración.

Sincronización

Java utiliza objetos *monitores* para la sincronización de hilos. Un objeto *monitor* es aquel que contiene métodos declarados con el modificador *synchronized*, sin importar de qué clase sea el objeto. Esto implica que cualquier objeto es susceptible de ser utilizado como un *monitor*.

Cuando un método *synchronized* de un objeto es llamado desde un hilo, ningún otro hilo puede acceder a éste u otro método *synchronized* del mismo objeto. Cuando un hilo entra a ejecutar un método *synchronized* de un objeto, el objeto queda bloqueado. Esto implica que si un hilo llama a un método *synchronized* de un objeto bloqueado, quedará en estado de *espera* hasta que éste se desbloquee. Cuando un hilo sale de la ejecución de un método *synchronized* de un objeto, éste se desbloquea, permitiendo que otros hilos accedan a él. Si hubiese algún hilo *esperando* una llamada pendiente a un método

synchronized de este objeto, el hilo pasa al estado *listo*, de forma que cuando entre al estado *en ejecución* pueda ejecutar el código del método.

En resumen, un hilo que accede a ejecutar un método *synchronized* de un objeto, bloquea a dicho objeto para el resto de los hilos. Sólo un método *synchronized* de un objeto puede ser ejecutado a la vez por un hilo. Sin embargo, el resto de hilos sí pueden acceder a los métodos de dicho objeto que no sean *synchronized*. Esto permite una fácil sincronización en el acceso a los recursos del objeto monitor. Esta técnica es equivalente al uso de secciones críticas en la programación en C para Windows.

El siguiente ejemplo muestra el uso de un monitor.

```
class Productor
{
    private int iTrabajo = 0;

    public synchronized int SiguienteTrabajo()
    {
        iTrabajo++;

        // simulo un tiempo de demora en la entrega del trabajo
        try
        {
            Thread.sleep( ( int )( Math.random() * 1000 ) );
        }
        catch( InterruptedException e )
        {
        }

        return iTrabajo;
    }
}

class HiloConsumidor extends Thread
{
    Productor p;

    HiloConsumidor( Productor p )
    {
        this.p = p;
    }

    public void run()
    {
        for( int i = 0; i < 5; i++ )
        {
            int iTrabajo = p.SiguienteTrabajo();
            System.out.println( getName() + ": Obtenido trabajo " + iTrabajo );

            try
            {
                // simulo un tiempo de trabajo
                sleep( ( int )( Math.random() * 1000 ) );

                // reporto que se finalizo el trabajo
                System.out.println( getName() + ": Trabajo "  + iTrabajo +
                    " completado" );
            }
            catch( InterruptedException e )
            {
```

```
                System.out.println( getName() + ": Trabajo "  + iTrabajo +
                    " interrumpido" );
            }
        }
    }
}

public class Ejemplo3
{
    public static void main( String args[] )
    {
        Productor p = new Productor();
        HiloConsumidor hilo1 = new HiloConsumidor( p );
        HiloConsumidor hilo2 = new HiloConsumidor( p );
        hilo1.start();
        hilo2.start();
    }
}
```

Al correr el programa la salida será la mostrada en la figura 8.15.

Este programa simula un proceso "productor-consumidor". El productor devuelve un trabajo a demanda al consumidor que se lo solicita. Cuando el productor calcula un nuevo trabajo le toma un tiempo aleatorio entregarlo. Sin embargo, dado que el método de entrega de un trabajo, SiguienteTrabajo, es declarado synchronized, sólo un hilo podrá obtener un trabajo a la vez, por lo que no habrá pérdida de trabajos.

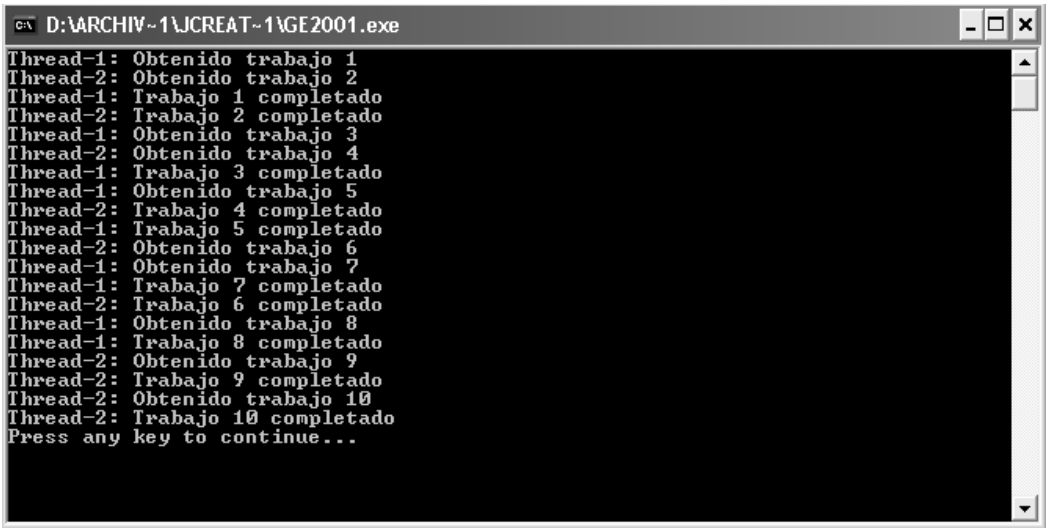

Figura 8.15. Salida del Ejemplo de Sincronización en Java

Para ejemplificar mejor esta sincronización, la salida en la figura 8.16 muestra lo que podría ocurrir si el método *SiguienteTrabajo* no se sincronizara:

Figura 8.16. Salida del Ejemplo sin Sincronización en Java

Note como algunos trabajos se pierden y otros se realizan dos veces. Esto se debe a que la instrucción:

```
iTrabajo++;
```

Es ejecutada, en ciertas ocasiones, por más de un hilo antes de que el método *SiguienteTrabajo* retorne.

Note el nombre que asigna el constructor por defecto de *Thread* a los hilos, dado que no se ha llamado a algún constructor con parámetros de *Thread* para darle uno.

Ahora bien, que pasaría en el caso de tener un productor que produce indiferentemente de que existan consumidores que consuman dichos trabajos, y a la vez consumidores que consuman indiferentemente de que exista algo que consumir. Si el productor elimina un trabajo para crear uno nuevo antes de que el anterior sea consumido, se perderá dicho trabajo. Si el consumidor consume un trabajo antes de que se haya producido uno nuevo, un trabajo será realizado dos o más veces. En este caso, ambas labores, la de producir y la de consumir deben de sincronizarse una con otra, esto es, el productor no debe de seguir produciendo mientras que no se haya consumido el trabajo anterior y el consumidor debe de esperar a que se produzca un nuevo trabajo antes de consumirlo.

En este tipo de situación, la sincronización utilizando métodos *synchronized* por sí sola no es suficiente. Se necesita que el hilo productor espere y notifique al consumidor, y éste a su vez espere y notifique al productor.

Para esto, la clase *Object*, de la que hereda cualquier objeto que se utilice como monitor, provee los método *wait*, *notify* y *notifyAll*. Estos métodos sólo pueden ejecutarse dentro de un método declarado *synchronized*, o en algún método llamado desde uno declarado *synchronized*. El no hacerlo así provocaría un error en tiempo de ejecución. Esto significa que estos métodos sólo pueden ser ejecutados desde un objeto monitor, por el hilo que haya bloqueado a este objeto.

El método *wait* coloca al hilo en estado de *espera* y desbloquea al objeto monitor. El hilo saldrá de dicho estado cuando otro hilo acceda a algún método *synchronized* del mismo objeto monitor y llame a *notify* o *notifyAll*. En ese momento el hilo será colocado en estado *listo* y competirá con el resto de hilos que intentan acceder a algún método *synchronized* de dicho objeto monitor (o que también hayan salido del estado de *espera*) para bloquearlo y ejecutar su código.

El método *notify* saca al primero de los hilos que esté en la lista de espera del objeto monitor, del estado *esperando* y lo coloca en estado *listo*. De esta forma el hilo vuelve a entrar a competir por el acceso al objeto monitor.

El método *notifyAll* saca a todos los hilos que esté en la lista de espera del objeto monitor, del estado *esperando* y los coloca en estado *listo*.

Tome en cuenta que sólo un hilo puede acceder a ejecutar el código de un método *synchronized* de un objeto monitor. Luego, aún cuando exista más de un hilo esperando a ejecutar un método *synchronized* y otros tantos que hayan sido sacados del estado de *espera* para el mismo objeto monitor, el primero que entre a *en ejecución* bloqueará al objeto monitor, lo que colocará a todos los demás hilos en estado *esperando*. Cuando el hilo ganador termine la ejecución de dicho método, todos los hilos que están *esperando* volverán al estado *listo*, de forma que vuelvan a competir por el acceso al objeto monitor.

El siguiente ejemplo muestra el uso de los métodos *wait* y *notify*.

```
class Trabajo
{
    private int NroTrabajo = -1;
    private boolean PuedoCrear = true;  // determina si se puede crear o consumir

    public synchronized void CrearNuevo( int NroTrabajo )
    {
        while ( !PuedoCrear ) {  // aún no se puede producir
            try {
                wait();
            }
            catch ( InterruptedException e ) {
            }
        }

        System.out.println( Thread.currentThread().getName() +
            " creo el trabajo " + NroTrabajo );

        this.NroTrabajo = NroTrabajo;

        PuedoCrear = false;
        notify();  // notifico que existe un trabajo listo
    }

    public synchronized int ObtenerTrabajo()
    {
        while ( PuedoCrear ) {   // aún no se puede obtener
            try {
                wait();
            }
            catch ( InterruptedException e ) {
            }
        }

        PuedoCrear = true;
        notify();  // notifico que ya se obtuvo el trabajo actual

        System.out.println( Thread.currentThread().getName() +
            " obtuvo el trabajo " + NroTrabajo );

        return NroTrabajo;
    }
}
```

```
class Productor extends Thread
{
    private Trabajo t;

    public Productor( Trabajo t )
    {
        this.t = t;
    }

    public void run()
    {
        for ( int Contador = 1; Contador <= 5; Contador++ )
        {
            // simulo un tiempo de demora aleatorio
            // en la creación del nuevo trabajo
            try
            {
                sleep( (int) ( Math.random() * 1000 ) );
            }
            catch( InterruptedException e )
            {
            }

            t.CrearNuevo( Contador );
        }

        System.out.println( getName() + " finalizo la produccion de trabajos" );
    }
}

class Consumidor extends Thread
{
    private Trabajo t;

    public Consumidor ( Trabajo t )
    {
        this.t = t;
    }

    public void run()
    {
        int iTrabajo;
        do {
            iTrabajo = t.ObtenerTrabajo();

            // simulo un tiempo de demora aleatorio
            // en el procesamiento del trabajo
            try
            {
                sleep( (int) ( Math.random() * 1000 ) );
            }
            catch( InterruptedException e )
            {
            }
        } while ( iTrabajo != 5 );

        System.err.println( getName() + " termino de procesar los trabajos" );
    }
}

public class Ejemplo4
{
```

```
        public static void main( String args[] )
        {
            Trabajo t = new Trabajo();
            Productor p = new Productor( t );
            Consumidor c = new Consumidor( t );

            p.start();
            c.start();
        }
    }
```

Al correr el programa la salida será la mostrada en la figura 8.17.

Note que a pesar de que tanto el productor como el consumidor tratan de crear y obtener respectivamente un trabajo en distinto orden, éstos son producidos y consumidos en el orden en que son creados gracias a la sincronización mediante *wait* y *notify*.

Note además que en algún momento el productor, por ejemplo, podría utilizar su propia notificación en su llamada a *wait*. Sin embargo, dado que se utiliza un *flag*, *PuedoCrear*, el hilo del productor nunca saldrá del bucle de espera hasta que efectivamente el trabajo haya sido retirado por el consumidor.

En ocasiones es necesario que un hilo verifique que otro haya finalizado para poder continuar con su trabajo. En este caso se utiliza el método *join*. Un hilo que llame al método *join* utilizando la referencia de otro hilo, pasará al estado de *esperando* hasta que dicho hilo termine, esto es, pase al estado *finalizado*. Cuando esto sucede, el hilo que espera pasa al estado *listo*.

Figura 8.17. Salida del Ejemplo de wait y notify en Java

El siguiente ejemplo muestra el uso del método *join*.

```
class MiHilo extends Thread
{
    private boolean bFinalizar = false;

    public void Finaliza()
    {
```

```
        bFinalizar = true;
    }

    public void run()
    {
        int iTrabajo = 0;

        while( !bFinalizar )
        {
            iTrabajo++;

            // reporto que se inicio el trabajo
            System.out.println( getName() + ": Inicio de trabajo " + iTrabajo );

            // simulo un tiempo de trabajo
            try
            {
                sleep( ( int )( Math.random() * 3000 ) );
            }
            catch( InterruptedException e )
            {
            }

            // reporto que se finalizo el trabajo
            System.out.println( getName() + ": Finalizo trabajo " + iTrabajo );
        }
        // reporto que finalizo el hilo
        System.out.println( getName() + ": Finalizo el hilo" );
    }
}

public class Ejemplo5
{
    public static void main( String args[] )
    {
        MiHilo hilo = new MiHilo();
        hilo.start();

        try
        {   // simulo un tiempo de trabajo
            Thread.sleep( ( int )( Math.random() * 3000 ) );

            // le aviso al hilo que finalice y lo espero
            hilo.Finaliza();
            hilo.join();
        }
        catch( InterruptedException e )
        {
        }

        // reporto el fin
        System.out.println( "Metodo main finalizo" );
    }
}
```

Al correr el programa se tendrá una salida como la mostrada en la figura 8.18.

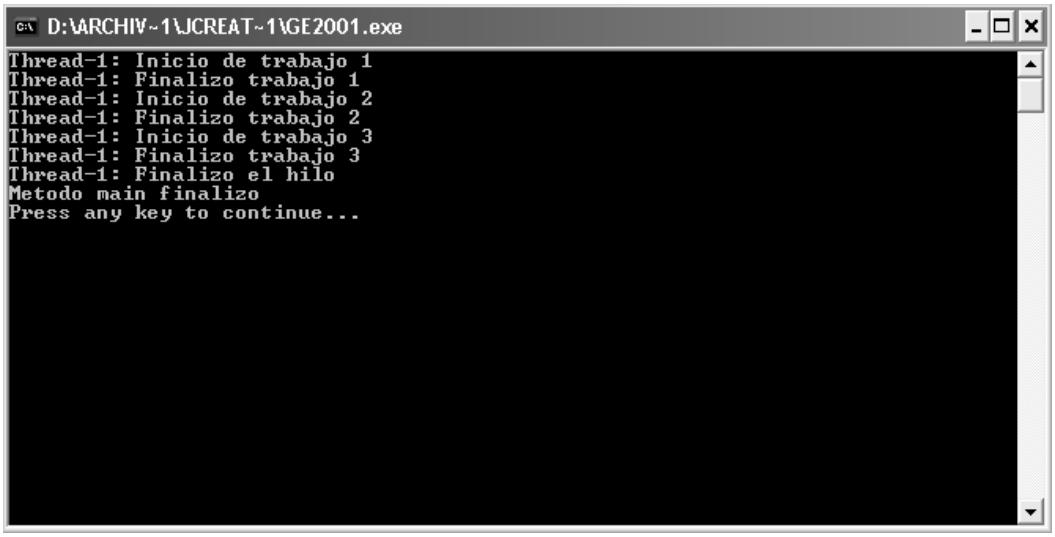

Figura 8.18. Salida del Ejemplo de join en Java

Note que el hilo primario, el que ejecuta al método *main*, espera a que el nuevo hilo creado finalice para continuar. Al igual que *sleep*, *join* también puede ser interrumpido y arrojar una excepción, por lo que se le ha colocado dentro del bloque *try*.

Note también que no es necesario definir un constructor en una clase hilo, dado que *Thread* posee un constructor por defecto. El único método que siempre debe de implementarse es *run*.

Aunque todos los ejemplos mostrados son aplicaciones de consola, el uso de hilos se aplica a todo tipo de programa en Java.